Dive into
Deep Learning
MATHEMATICS FOR DEEP LEARNING
Alexander J Smola · Rachel Hu · Brent Werness

Table of contents

Mathematics for Deep Learning 1

33 Linear Algebra 5
33.1 Vectors, Angles, and Projections . . . 6
33.1.1 Vectors and Their Geometry . . . 6
33.1.2 Subspaces and Least Squares . . . 11
33.1.3 Similarity in High Dimensions . . . 16
33.1.4 Hyperplanes and Decision Boundaries . . . 20
33.1.5 Summary . . . 25
33.1.6 Exercises . . . 25
33.2 Matrices as Linear Maps . . . 27
33.2.1 Linear Maps . . . 28
33.2.2 Rank and Invertibility . . . 35
33.2.3 The Determinant . . . 41
33.2.4 Summary . . . 50
33.2.5 Exercises . . . 50
33.3 Eigendecompositions . . . 52
33.3.1 Eigenvalues and Diagonalization . . . 53
33.3.2 Symmetric Matrices and Quadratic Forms . . . 64
33.3.3 Summary . . . 69
33.3.4 Exercises . . . 70
33.4 Spectra in Practice . . . 71
33.4.1 Locating and Computing Eigenvalues . . . 72
33.4.2 Iteration and Stability . . . 78
33.4.3 Summary . . . 83
33.4.4 Exercises . . . 84
33.5 Singular Value Decomposition and Low-Rank Approximation . . . 86
33.5.1 The Singular Value Decomposition . . . 87
33.5.2 Low-Rank Approximation . . . 94
33.5.3 Solving Linear Systems with the SVD . . . 102
33.5.4 The SVD in Deep Learning . . . 107
33.5.5 Summary . . . 112
33.5.6 Exercises . . . 112
33.6 Further Reading . . . 115
33.6.1 Geometry and Linear Maps . . . 115
33.6.2 Eigenvalues and Spectra . . . 116
33.6.3 Low Rank Beyond the Truncated SVD . . . 118

33.6.4 Books, Courses, and Notes . . . 119

34 Calculus and Automatic Differentiation 121
34.1 Single Variable Calculus . . . 122
34.1.1 The Derivative . . . 123
34.1.2 Computing Derivatives . . . 125
34.1.3 The Mean Value Theorem . . . 128
34.1.4 Gradient Descent in One Variable . . . 130
34.1.5 Curvature and Taylor Series . . . 133
34.1.6 Nondifferentiable Points . . . 143
34.1.7 Summary . . . 147
34.1.8 Exercises . . . 147
34.2 Multivariable Calculus . . . 150
34.2.1 Partial Derivatives and the Gradient . . . 151
34.2.2 The Geometry of Gradients . . . 153
34.2.3 The Hessian . . . 157
34.2.4 Constrained Optimization and Lagrange Multipliers . . . 165
34.2.5 Summary . . . 167
34.2.6 Exercises . . . 167
34.3 The Chain Rule on Computational Graphs . . . 169
34.3.1 The Chain Rule as a Sum over Paths . . . 170
34.3.2 Backpropagation . . . 173
34.3.3 Summary . . . 177
34.3.4 Exercises . . . 177
34.4 Matrix Calculus . . . 178
34.4.1 The Jacobian . . . 179
34.4.2 The Chain Rule for Jacobians . . . 182
34.4.3 Derivatives with Respect to Vectors and Matrices . . . 185
34.4.4 Summary . . . 191
34.4.5 Exercises . . . 191
34.5 Automatic Differentiation . . . 193
34.5.1 Forward Mode and Dual Numbers . . . 194
34.5.2 Reverse Mode and the Tape . . . 198
34.5.3 Composing Jacobian Products . . . 206
34.5.4 Summary . . . 210
34.5.5 Exercises . . . 210
34.6 Integral Calculus . . . 213
34.6.1 The Definite Integral . . . 214
34.6.2 The Fundamental Theorem . . . 216
34.6.3 Multiple Integrals . . . 223
34.6.4 Summary . . . 231
34.6.5 Exercises . . . 231
34.7 Expectations and Their Gradients . . . 233
34.7.1 Expectations as Integrals . . . 233
34.7.2 Differentiation under the Integral Sign . . . 234
34.7.3 Score-Function and Pathwise Gradients . . . 236
34.7.4 Summary . . . 239
34.7.5 Exercises . . . 239
34.8 Further Reading . . . 241
34.8.1 Derivatives at Kinks . . . 241

34.8.2 Derivatives of Vector and Matrix Maps . . . 242
34.8.3 Automatic Differentiation . . . 242
34.8.4 Integration . . . 243
34.8.5 Books, Courses, and Tutorials . . . 244

35 Optimization **247**
Notation . . . 248
35.1 Gradient Descent and Curvature . . . 248
35.1.1 Descent Directions and Smoothness . . . 249
35.1.2 The Quadratic Model and the Condition Number . . . 256
35.1.3 Momentum and Acceleration . . . 261
35.1.4 Summary . . . 265
35.1.5 Exercises . . . 266
35.2 Curvature Methods: Newton, Quasi-Newton, and Trust Regions . . . 269
35.2.1 Newton's Method . . . 269
35.2.2 Quasi-Newton Methods . . . 274
35.2.3 Trust Regions . . . 276
35.2.4 Summary . . . 280
35.2.5 Exercises . . . 280
35.3 Convexity . . . 281
35.3.1 Convex Sets . . . 282
35.3.2 Convex Functions . . . 283
35.3.3 Consequences of Convexity . . . 292
35.3.4 Recognizing Convexity . . . 299
35.3.5 Beyond Convexity . . . 301
35.3.6 Summary . . . 305
35.3.7 Exercises . . . 305
35.4 Stochastic Gradient Methods . . . 308
35.4.1 Minibatch Gradients . . . 309
35.4.2 Step-Size Decay . . . 314
35.4.3 Rates for a Fixed Budget . . . 320
35.4.4 Variance Reduction . . . 325
35.4.5 Summary . . . 328
35.4.6 Exercises . . . 329
35.5 Adaptive Methods . . . 331
35.5.1 Per-Coordinate Step Sizes . . . 332
35.5.2 Coordinate Scaling and Its Limits . . . 336
35.5.3 Decoupled Weight Decay . . . 342
35.5.4 Warmup . . . 345
35.5.5 Structured Preconditioners . . . 346
35.5.6 Summary . . . 348
35.5.7 Exercises . . . 349
35.6 Constrained Optimization . . . 351
35.6.1 Equality Constraints and Lagrange Multipliers . . . 352
35.6.2 Inequality Constraints and the KKT Conditions . . . 355
35.6.3 Projections and Projected Gradient Descent . . . 358
35.6.4 Proximal Operators and Coordinate Descent . . . 365
35.6.5 Summary . . . 371
35.6.6 Exercises . . . 371
35.7 Duality . . . 374

35.7.1 The Dual Problem . . . 375
35.7.2 Strong Duality . . . 379
35.7.3 Multipliers as Prices . . . 383
35.7.4 The Convex Conjugate . . . 385
35.7.5 Worked Examples . . . 389
35.7.6 Summary . . . 396
35.7.7 Exercises . . . 396
35.8 Numerical Stability and Conditioning . . . 399
35.8.1 Floating-Point Arithmetic . . . 400
35.8.2 Stable Softmax and Cross-Entropy . . . 405
35.8.3 Catastrophic Cancellation . . . 409
35.8.4 Conditioning . . . 415
35.8.5 Summary . . . 422
35.8.6 Exercises . . . 423
35.9 Further Reading . . . 426
35.9.1 Gradient Methods . . . 426
35.9.2 Stochastic and Adaptive Methods . . . 427
35.9.3 Constraints and Duality . . . 428
35.9.4 Floating Point . . . 429
35.9.5 General Optimization Resources . . . 430

36 Probability and Statistical Learning **433**
36.1 Random Variables . . . 434
36.1.1 Densities . . . 434
36.1.2 Mean and Variance . . . 440
36.1.3 Tails . . . 444
36.1.4 Summary . . . 449
36.1.5 Exercises . . . 449
36.2 Joint Distributions and Transformations . . . 451
36.2.1 Joint and Conditional Densities . . . 452
36.2.2 Covariance and Correlation . . . 459
36.2.3 Transformations of Random Variables . . . 464
36.2.4 Summary . . . 468
36.2.5 Exercises . . . 469
36.3 Distributions . . . 470
36.3.1 Discrete Distributions . . . 471
36.3.2 Continuous Distributions . . . 479
36.3.3 The Multivariate Gaussian . . . 487
36.3.4 Summary . . . 492
36.3.5 Exercises . . . 493
36.4 The Exponential Family . . . 495
36.4.1 One Form for Many Distributions . . . 496
36.4.2 Maximum Entropy . . . 499
36.4.3 The Log-Partition Function . . . 503
36.4.4 Origins of the Softmax . . . 507
36.4.5 Conjugate Priors . . . 510
36.4.6 Summary . . . 514
36.4.7 Exercises . . . 515
36.5 Statistics . . . 517
36.5.1 Estimators and Their Sampling Distribution . . . 518

36.5.2 Hypothesis Testing . . . 528
36.5.3 Confidence Intervals . . . 537
36.5.4 The Bootstrap . . . 541
36.5.5 Summary . . . 544
36.5.6 Exercises . . . 544
36.6 Maximum Likelihood . . . 549
36.6.1 The Maximum Likelihood Principle . . . 549
36.6.2 Losses as Negative Log-Likelihoods . . . 552
36.6.3 Accuracy of the Maximum Likelihood Estimate . . . 556
36.6.4 MAP Estimation and Regularization . . . 566
36.6.5 Summary . . . 569
36.6.6 Exercises . . . 569
36.7 Latent Variables, EM, and the ELBO . . . 572
36.7.1 Mixture Models . . . 573
36.7.2 The Evidence Lower Bound . . . 574
36.7.3 Expectation–Maximization . . . 575
36.7.4 Summary . . . 581
36.7.5 Exercises . . . 581
36.8 Bayesian Computation . . . 582
36.8.1 The Posterior of Bayesian Logistic Regression . . . 583
36.8.2 Rejection and Importance Sampling . . . 590
36.8.3 Markov Chain Monte Carlo . . . 594
36.8.4 Variational Inference . . . 601
36.8.5 Comparison of the Methods . . . 605
36.8.6 Summary . . . 607
36.8.7 Exercises . . . 607
36.9 Concentration Inequalities . . . 611
36.9.1 Exponential Tail Bounds . . . 611
36.9.2 Probability in High Dimension . . . 621
36.9.3 Summary . . . 626
36.9.4 Exercises . . . 627
36.10 Uniform Convergence and Generalization . . . 629
36.10.1 Uniform Convergence . . . 630
36.10.2 Rademacher Complexity . . . 633
36.10.3 Interpolation and Double Descent . . . 642
36.10.4 Summary . . . 648
36.10.5 Exercises . . . 648
36.11 Naive Bayes . . . 651
36.11.1 Bayes' Rule for Classification . . . 652
36.11.2 Learning by Counting . . . 655
36.11.3 Fashion-MNIST by Counting . . . 657
36.11.4 Evaluating the Classifier . . . 662
36.11.5 Summary . . . 668
36.11.6 Exercises . . . 668
36.12 Further Reading . . . 672
36.12.1 Probability Models . . . 672
36.12.2 Estimation and Testing . . . 673
36.12.3 Computing with Posteriors . . . 674
36.12.4 Concentration and Learning Theory . . . 675
36.12.5 Classifiers from Counts . . . 676

36.12.6 General Probability and Statistics Resources 676

37 Information Theory and Divergences **679**
37.1 Entropy, Cross-Entropy, and KL Divergence 680
37.1.1 Information and Entropy 681
37.1.2 KL Divergence and Cross-Entropy 686
37.1.3 The Classification Loss 690
37.1.4 Entropy and KL Divergence for Densities 696
37.1.5 Summary 700
37.1.6 Exercises 701
37.2 Codes, Compression, and Perplexity 703
37.2.1 Symbol Codes 704
37.2.2 Stream Codes and Description Length 709
37.2.3 Entropy Rate and Typical Sequences 714
37.2.4 Perplexity 718
37.2.5 Summary 720
37.2.6 Exercises 721
37.3 f-Divergences 724
37.3.1 The f-Divergence Family 725
37.3.2 Forward and Reverse KL 732
37.3.3 Total Variation, Pinsker's Inequality, and Data Processing 735
37.3.4 The Fenchel Conjugate and the Critic Bound 740
37.3.5 Summary 744
37.3.6 Exercises 744
37.4 Test Functions and Transport 746
37.4.1 Integral Probability Metrics and MMD 748
37.4.2 Optimal Transport and the Wasserstein Distance 753
37.4.3 Fixed Kernels and Trained Critics 759
37.4.4 Summary 760
37.4.5 Exercises 760
37.5 Score-Based Comparisons 763
37.5.1 The Score and the Fisher Divergence 764
37.5.2 Stein's Identity 768
37.5.3 The Kernel Stein Discrepancy 770
37.5.4 Training Objectives and Their Divergences 773
37.5.5 Summary 775
37.5.6 Exercises 775
37.6 Mutual Information 777
37.6.1 Measures of Dependence 778
37.6.2 Processing and Prediction 785
37.6.3 Communication and Compression 791
37.6.4 The Information Bottleneck 795
37.6.5 Summary 800
37.6.6 Exercises 800
37.7 Estimating Mutual Information 803
37.7.1 A Pair with Known Information 804
37.7.2 Nonparametric Estimation 805
37.7.3 Variational Lower Bounds 812
37.7.4 InfoNCE 815
37.7.5 Summary 824

37.7.6 Exercises . . . 825
37.8 Further Reading . . . 828
37.8.1 Entropy and Losses . . . 828
37.8.2 Codes and Compression . . . 829
37.8.3 Comparing Distributions . . . 831
37.8.4 Mutual Information . . . 833
37.8.5 Estimating Mutual Information . . . 834
37.8.6 General Information Theory Resources . . . 835

38 Dynamics: Differential Equations and Generative Flows 837
Notation . . . 838
38.1 Ordinary Differential Equations . . . 839
38.1.1 Vector Fields and Flows . . . 840
38.1.2 Linear ODEs and Stability . . . 846
38.1.3 Summary . . . 854
38.1.4 Exercises . . . 854
38.2 Numerical Solvers . . . 856
38.2.1 Forward Euler . . . 857
38.2.2 Runge–Kutta Methods . . . 859
38.2.3 Stiffness and Implicit Methods . . . 863
38.2.4 Gradient Descent as a Solver . . . 865
38.2.5 Summary . . . 868
38.2.6 Exercises . . . 869
38.3 Neural ODEs, Adjoints, and Continuous Flows . . . 871
38.3.1 Neural ODEs . . . 872
38.3.2 The Adjoint Method . . . 877
38.3.3 Continuous Normalizing Flows . . . 883
38.3.4 Summary . . . 889
38.3.5 Exercises . . . 889
38.4 Stochastic Differential Equations . . . 892
38.4.1 Brownian Motion . . . 893
38.4.2 Itô Calculus . . . 897
38.4.3 The Ornstein–Uhlenbeck Process . . . 906
38.4.4 Simulation . . . 910
38.4.5 Summary . . . 915
38.4.6 Exercises . . . 915
38.5 The Fokker–Planck Equation and Probability Flow . . . 918
38.5.1 From Paths to Densities . . . 919
38.5.2 The Fokker–Planck Equation . . . 921
38.5.3 The Score Function . . . 927
38.5.4 The Continuity Equation and the Probability-Flow ODE . . . 931
38.5.5 Time Reversal . . . 937
38.5.6 Summary . . . 944
38.5.7 Exercises . . . 944
38.6 Score Matching and Diffusion Models . . . 948
38.6.1 Learning the Score . . . 949
38.6.2 Sampling with a Score . . . 957
38.6.3 Scores Across Noise Levels . . . 959
38.6.4 Deterministic Sampling and Guidance . . . 964
38.6.5 Summary . . . 970

38.6.6 Exercises . . . 970
38.7 Flow Matching and Optimal Transport . . . 974
38.7.1 Flow Matching . . . 975
38.7.2 Rectified Flow . . . 981
38.7.3 Optimal Transport and Straightness . . . 990
38.7.4 Sampling with a Learned Field . . . 994
38.7.5 Summary . . . 998
38.7.6 Exercises . . . 998
38.8 Further Reading . . . 1001
38.8.1 Solvers . . . 1001
38.8.2 Continuous-Time Models . . . 1002
38.8.3 Noise . . . 1003
38.8.4 Learned Generators . . . 1005
38.8.5 General Dynamics and Generative Flow Resources . . . 1006

References **1009**

Mathematics for Deep Learning

Throughout the main chapters of this book, we introduce mathematical tools *just in time*—stating each result alongside its geometric, statistical, or physical intuition so that you can build and train working models without losing momentum. Yet anyone who wants to move beyond applying existing recipes to diagnosing subtle training failures, reading frontier research papers critically, or inventing new methods eventually needs to look inside the mathematical engine. Why does the learning rate of gradient descent have a hard ceiling set by the largest eigenvalue of the Hessian? Why does minimizing cross-entropy estimate the true conditional distribution, and why does the Jensen–Shannon divergence saturate when two distributions fail to overlap? How does adding Gaussian noise to an image day by day define a reverse-time differential equation that turns pure noise back into a photograph?

This six-chapter appendix answers those questions in full. It is designed to serve two complementary purposes:

- **A companion reference for the main text.** Whenever a chapter in *Basics* or *Advanced* uses a mathematical theorem or identity and focuses on its practical consequences, it points to the corresponding section here for the complete derivation, proof, and boundary conditions. If you are already comfortable with a topic, you can skip its chapter here and dip in only when a cross-reference piques your curiosity.
- **A self-contained textbook for a course on the mathematics of deep learning.** Read from start to finish, these six chapters form a coherent, modern mathematical curriculum tailored specifically to deep learning—developing each subject from first principles as far as modern machine learning actually uses it, without getting bogged down in pathology for its own sake.

Just like the rest of the book, every section in this appendix pairs its mathematical derivations with runnable code and computational experiments (implemented in PyTorch, JAX, TensorFlow, and MXNet), turning abstract theorems into concrete numerical phenomena that you can inspect, plot, and test for yourself:

- **Linear Algebra (Chapter 33).** Deep networks represent data and parameters as vectors, matrices, and tensors, and every layer transforms the geometry of space. Beginning with lengths, angles, inner products, projections, and vector/matrix norms, this chapter treats matrices as active linear transformations that rotate, stretch, and collapse subspaces. We then develop the spectral decompositions that govern high-dimensional dynamics: *eigendecomposition* (showing how repeated matrix multiplication in recurrences or optimization is controlled by eigenvalues), *spectra in practice* (from Gershgorin circles and condition numbers to random matrix theory and the Marchenko–Pastur law), and the *singular value decomposition* (SVD), Moore–Penrose

pseudoinverse, and Eckart–Young low-rank approximation that underlie parameter-efficient fine-tuning (LoRA) and spectral normalization.

- **Calculus (Chapter 34).** Training a neural network with billions of parameters rests entirely on our ability to compute how a scalar loss changes when every parameter is perturbed simultaneously. Starting from single-variable derivatives, Taylor series, and numerical differentiation pitfalls, we move to multivariable gradients, directional derivatives, Jacobian matrices, and Hessian curvature. We then trace the multivariate chain rule through computation graphs to demystify *automatic differentiation*—contrasting forward-mode and reverse-mode (*backpropagation*) accumulation and showing why reverse-mode computes the exact gradient with respect to millions of parameters at the cost of a single forward pass.
- **Optimization (Chapter 35).** This chapter supplies the formal proofs behind the empirical phenomena observed in Chapter 9. We establish the geometry of convex sets and convex functions (Jensen's inequality, first- and second-order convexity characterizations), prove how L-smoothness and μ-strong convexity govern the convergence rate and step-size stability of gradient descent via the condition number $\kappa = L/\mu$, analyze momentum and Newton/quasi-Newton preconditioning on ill-conditioned quadratic models, and develop constrained optimization via Lagrange multipliers, Karush–Kuhn–Tucker (KKT) optimality conditions, and Lagrangian duality.
- **Probability and Statistics (Chapter 36).** Machine learning reasons under uncertainty at every stage, from noisy data generation and stochastic minibatch sampling to out-of-sample generalization. We build from probability spaces, conditional distributions, Bayes' rule, expectations, and key parametric families (Bernoulli, categorical, Gaussian, Laplace, exponential) to the classical *concentration inequalities* (Markov, Chebyshev, Hoeffding) that bound rare deviations of empirical averages. We then develop statistical inference and learning theory: maximum likelihood estimation (MLE), Fisher information, the Cramér–Rao lower bound, Bayesian *maximum a posteriori* (MAP) estimation, sufficient statistics, hypothesis testing, confidence intervals, and the theoretical foundations of *generalization* (empirical risk minimization, uniform convergence, Rademacher complexity, VC dimension, and the bias–variance trade-off).
- **Information Theory (Chapter 37).** Nearly every classification, language-modeling, and generative objective in deep learning descends from information theory. Starting from Claude Shannon's axioms of self-information and entropy, we show how *cross-entropy* and *Kullback–Leibler (KL) divergence* measure the coding inefficiency of using an approximate model distribution Q in place of the true data distribution P, and extend these concepts to continuous random variables via differential entropy and the maximum-entropy principle. We then broaden the lens to compare probability distributions when KL divergence fails: f-divergences, the variational dual representations behind GANs, and *Wasserstein optimal transport* distances that respect the underlying geometry of the sample space. The chapter closes with *mutual information*, the data processing inequality, the variational information bottleneck, and the InfoNCE lower bound that drives modern contrastive representation learning.
- **Dynamics, Flows, and Diffusion (Chapter 38).** Many of the most powerful ideas in modern deep learning—from continuous normalizing flows and state space models to denoising diffusion and flow matching—are most naturally understood in continuous time. This final chapter builds the dynamical toolkit from the ground up: systems of ordinary differential equations (ODEs), vector fields, phase portraits, linear stability, numerical integrators (Euler, Runge–Kutta, and symplectic leapfrog), and *Neural ODEs* trained via the continuous adjoint sensitivity method. We then track how

a velocity field transports an entire probability density via the *continuity equation* and Hutchinson's trace estimator, before introducing randomness: Brownian motion, *Itô stochastic differential equations* (SDEs), Itô's lemma, the Ornstein–Uhlenbeck process, the *Fokker–Planck equation*, and Anderson's reverse-time SDE and the *probability-flow ODE*, which reveal how learning the score function $\nabla_{\mathbf{x}} \log p_t(\mathbf{x})$ allows a neural network to turn pure Gaussian noise into samples from any target distribution.

Chapter 33

Linear Algebra

Every neural network in this book is built out of matrix products: a linear layer multiplies its input by a weight matrix, attention multiplies queries by keys, and a recurrent network multiplies its hidden state by the same matrix at every step. In Section 1.3, we computed these products without asking what a matrix actually *does* to the vectors it acts on. This optional chapter takes up that geometric question, proofs included; nothing in the main text depends on it, yet seeing the geometry behind matrix products changes how you read almost every model in the book.

Consider a 784-pixel Fashion-MNIST image mapped to 256 features. How similar are two images, how much information does the layer discard, and what happens if we apply the same linear map fifty times in a row? Because we cannot draw in 784 dimensions, our answers come from algebra that carries plane geometry into any dimension.

A complicated matrix becomes transparent once viewed along the right directions. Around 1810, Gauss fitted the orbit of the asteroid Pallas to more observations than unknowns via least squares, a problem our first section solves and our last revisits. In orthonormal axes, lengths and angles reduce to dot products; along eigenvectors, a square matrix scales each direction by a single scalar; and in a pair of singular bases, *any* matrix decomposes into independent stretches. These ideas explain why attention divides scores by $\sqrt{d}$ (Section 10.2), when recurrent gradients explode (Section 8.5), and what a low-rank adapter can represent.

The chapter moves in two stages. First, we apply a matrix once: Section 33.1 extends lengths, angles, and orthogonal projections (giving the closest point in a subspace), while Section 33.2 tracks which directions collapse to zero and when a map is invertible. Next, we choose coordinates tailored to the matrix: eigenvectors and the spectral theorem for symmetric matrices (Section 33.3), non-normal behavior that eigenvalues alone miss (Section 33.4), and singular vectors with optimal low-rank truncation for any shape (Section 33.5). Two small matrices keep us honest throughout: $\mathbf{A} = \left(\begin{smallmatrix} 1 & 2 \\ -1 & 3 \end{smallmatrix}\right)$, with complex eigenvalues, and the shear $\mathbf{T} = \left(\begin{smallmatrix} 1 & 1 \\ 0 & 1 \end{smallmatrix}\right)$, whose eigenvectors fail to span the plane.

On a first reading, feel free to skim n-dimensional determinants and the discussions of Gershgorin discs, PageRank, and pseudospectra in Section 33.4. Numerical eigensolvers and linear solvers are surveyed in Section 33.6.

33.1 Vectors, Angles, and Projections

Throughout the Basics chapters, we represented every input as a vector: an image became a list of pixel values, and a word became an embedding. At heart, machine learning models perform three geometric operations on these vectors. They *compare* them, as when we measure whether two embeddings are similar; they *fit* them, as when we seek the combination of features that best matches a set of targets; and they *separate* them, as when a classifier decides which side of a boundary an input falls on. In the two-dimensional plane, each of these operations is a familiar picture of lengths, angles, and shadows. Real inputs, however, live in spaces of hundreds or thousands of dimensions that nobody can draw. Do those geometric pictures still mean anything there?

Telling t-shirts from trousers offers a concrete test. Each 28×28 Fashion-MNIST image (Section 3.2) is a list of 784 pixel intensities, so it is a point in $\mathbb{R}^{784}$. Consider the simplest rule imaginable: compute the average t-shirt and the average pair of trousers, and assign each new image to whichever average lies closer. In the plane, "closer" is something we can check by eye; in $\mathbb{R}^{784}$, it becomes a calculation requiring a single dot product per image. On the $2{,}000$ test images from these two classes, this rule is right about 92% of the time. The fact that such a simple rule works so well tells us that the two clouds of points sit well apart, and that geometric notions like distance remain meaningful even where we cannot see them.

What carries the geometry of the plane into arbitrary dimensions is the dot product. Lengths are square roots of dot products, the shadow of one vector onto another is a ratio of dot products, and the angle between two vectors is defined directly from them. We develop this geometry in three stages. First, we study a pair of vectors: projection onto a line, the angle between them, and the Cauchy–Schwarz inequality, which Section 1.3 proved algebraically and which we prove again here from a right triangle. Second, we introduce the language of subspaces (span, independence, basis, and dimension) that the rest of the chapter relies on, and we solve least squares as a projection, with the QR factorization as our computational tool. Third, we take these ideas to high-dimensional images, asking how similar unrelated vectors are in $\mathbb{R}^d$ and reading a linear classifier as a hyperplane.

Three results stand out above the rest: the residual of a projection is orthogonal to the line or subspace we project onto, which is what makes the projection the closest point; Cauchy–Schwarz, which ensures that angles remain well defined in any dimension; and the typical scale $1/\sqrt{d}$ of the cosine between random directions in $\mathbb{R}^d$. Aside from loading the dataset, all code in this section is plain NumPy and runs identically across frameworks.

33.1.1 Vectors and Their Geometry

33.1.1.1 Points and Directions

We can picture a vector in two complementary ways. Viewed as a *point*, $\mathbf{v} = (3, 2)$ marks the location 3 units to the right of the *origin* and 2 units up. This viewpoint turns an entire dataset into a cloud of points, turning a classification task such as separating t-shirts from trousers into a geometric problem of separating two clouds. Viewed as a *direction* (or displacement), the same vector instructs us to move 3 steps right and 2 steps up. Because a displacement can start anywhere, all of the parallel arrows in Figure 33.1 represent the same vector.

The direction viewpoint makes vector addition immediate: follow the first arrow, then fol-

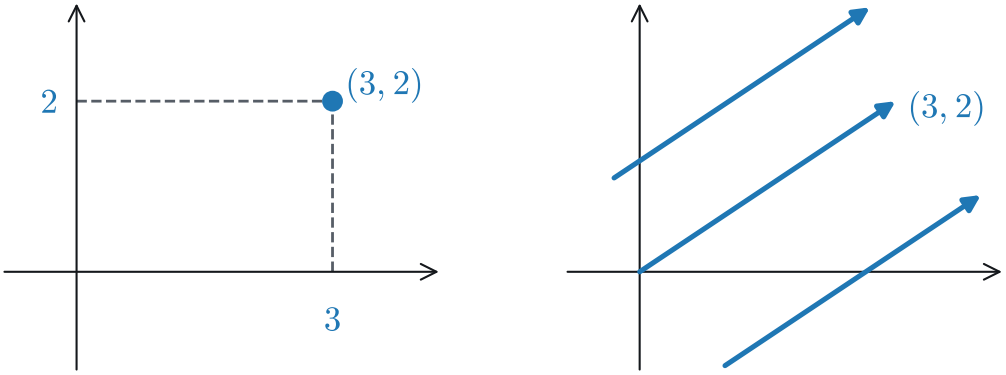


Figure 33.1: Two ways to read a vector: as a *point* with coordinates (x, y) (left), or as a *direction*, an arrow that can start anywhere, so every arrow shown represents the same vector $(3, 2)$ (right).

low the second from its tip, as shown in Figure 33.2.

Subtraction works the same way. Because $\mathbf{u} = \mathbf{v} + (\mathbf{u} - \mathbf{v})$, the difference $\mathbf{u} - \mathbf{v}$ is simply the displacement arrow that carries us from the point $\mathbf{v}$ to the point $\mathbf{u}$.

Everything else in this section grows out of a single operation, the *dot product* introduced in Section 1.3. Given two vectors $\mathbf{u}, \mathbf{v} \in \mathbb{R}^d$, it multiplies their corresponding entries and sums the results:

$$\mathbf{u}^\top \mathbf{v} = \sum_{i=1}^{d} u_i v_i. \tag{33.1}$$

Since the dot product is symmetric, $\mathbf{u}^\top \mathbf{v} = \mathbf{v}^\top \mathbf{u}$, we often write it as $\mathbf{u} \cdot \mathbf{v}$: using the dot when we picture two arrows and the transpose when the product sits inside a matrix expression. For $\mathbf{u} = (1, 3)$ and $\mathbf{v} = (2, 1)$, we have $\mathbf{u} \cdot \mathbf{v} = 2 + 3 = 5$, and the squared length of $\mathbf{u}$ is $\mathbf{u} \cdot \mathbf{u} = 1 + 9 = 10$.

33.1.1.2 Projection onto a Line

How much of a vector $\mathbf{v}$ points along another vector $\mathbf{w}$? Equivalently, which scalar multiple $t\mathbf{w}$ comes closest to $\mathbf{v}$? This is the simplest fitting problem in linear algebra: we approximate $\mathbf{v}$ using a single direction and seek the best coefficient t.

Let us test this on our pair $\mathbf{v} = (1, 3)$ and $\mathbf{w} = (2, 1)$ by trying a few values of t. At $t = 1$, the error $\mathbf{v} - t\mathbf{w} = (-1, 2)$ has squared length 5. At $t = 0.5$, the error is $(0, 2.5)$ with squared length 6.25; at $t = 1.5$, it is $(-2, 1.5)$, again with squared length 6.25. The minimum occurs at $t = 1$, giving the closest point $(2, 1)$. We call the remaining error the *residual*. Here the residual is $\mathbf{r} = (-1, 2)$, and notice its dot product with $\mathbf{w}$: $\mathbf{r} \cdot \mathbf{w} = -2 + 2 = 0$. The shortest error meets the line at a right angle. Even better, the squared lengths satisfy a right-triangle relation: $\|\mathbf{v}\|^2 = 10$, the closest point has squared length 5, the residual has squared length 5, and $10 = 5 + 5$.

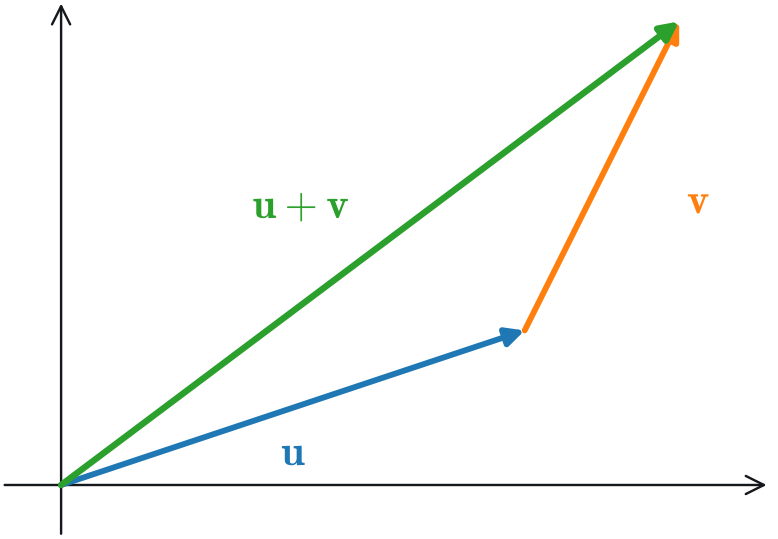


Figure 33.2: Vector addition visualized tip to tail: follow the first arrow, then follow the second from where the first ends.

This pattern holds in every dimension.

Proposition (orthogonal projection). *Let* $\mathbf{w} \neq \mathbf{0}$. *The multiple of* $\mathbf{w}$ *closest to* $\mathbf{v}$ *is*

$$\operatorname{proj}_{\mathbf{w}} \mathbf{v} = \frac{\mathbf{v} \cdot \mathbf{w}}{\mathbf{w} \cdot \mathbf{w}} \, \mathbf{w}, \tag{33.2}$$

and the residual $\mathbf{r} = \mathbf{v} - \operatorname{proj}_{\mathbf{w}} \mathbf{v}$ *satisfies* $\mathbf{r} \cdot \mathbf{w} = 0$.

Proof. We minimize the squared error as a function of t. Expanding the dot product of $\mathbf{v} - t\mathbf{w}$ with itself yields a quadratic in t,

$$q(t) = \|\mathbf{v} - t\mathbf{w}\|^2 = \|\mathbf{w}\|^2 \, t^2 - 2\,(\mathbf{v} \cdot \mathbf{w})\, t + \|\mathbf{v}\|^2.$$

Because its leading coefficient $\|\mathbf{w}\|^2$ is positive, $q(t)$ has a unique minimum where its derivative $q'(t) = 2\|\mathbf{w}\|^2 t - 2\,\mathbf{v} \cdot \mathbf{w}$ vanishes, namely at $t^\star = (\mathbf{v} \cdot \mathbf{w})/(\mathbf{w} \cdot \mathbf{w})$. Substituting $t^\star$ gives Equation 33.2. Checking the residual, we find $\mathbf{r} \cdot \mathbf{w} = \mathbf{v} \cdot \mathbf{w} - t^\star \, \mathbf{w} \cdot \mathbf{w} = 0$. $\blacksquare$

For our example, $q(t) = 5t^2 - 10t + 10$, which reaches its minimum value of 5 at $t^\star = 5/5 = 1$, confirming our trial values. We call $\operatorname{proj}_{\mathbf{w}} \mathbf{v}$ the *projection*, or *shadow*, of $\mathbf{v}$ onto $\mathbf{w}$. The projection captures the part of $\mathbf{v}$ aligned with $\mathbf{w}$, while the residual captures the part at right angles to it. Notice why we required $\mathbf{w} \neq \mathbf{0}$: the zero vector defines no direction, and setting $\mathbf{w} = \mathbf{0}$ would divide by zero.

That right angle is the reason squared lengths add cleanly. Two vectors $\mathbf{a}$ and $\mathbf{b}$ are *orthogonal* whenever $\mathbf{a} \cdot \mathbf{b} = 0$. Expanding the squared length of their sum then gives *Pythagoras' theorem*,

$$\|\mathbf{a}+\mathbf{b}\|^2 = \|\mathbf{a}\|^2 + 2\,\mathbf{a}\cdot\mathbf{b} + \|\mathbf{b}\|^2 = \|\mathbf{a}\|^2 + \|\mathbf{b}\|^2. \tag{33.3}$$

In any dimension, orthogonal components add in squared length with no cross term. Applying this to the decomposition $\mathbf{v} = \operatorname{proj}_{\mathbf{w}}\mathbf{v} + \mathbf{r}$ gives $\|\mathbf{v}\|^2 = \|\operatorname{proj}_{\mathbf{w}}\mathbf{v}\|^2 + \|\mathbf{r}\|^2$, which is $10 = 5 + 5$ in our example.

We will also need the length of the shadow together with a sign indicating whether it points in the same direction as $\mathbf{w}$ or opposite to it. Multiplying the coefficient $t^\star$ by $\|\mathbf{w}\|$ gives this *signed length*:

$$\frac{\mathbf{v}\cdot\mathbf{w}}{\|\mathbf{w}\|}. \tag{33.4}$$

In our example it equals $5/\sqrt{5} = \sqrt{5}$, the length of $(2,1)$. This signed length will reappear at the end of the section as the signed distance from a point to a hyperplane. Figure 33.3 illustrates the entire geometry.

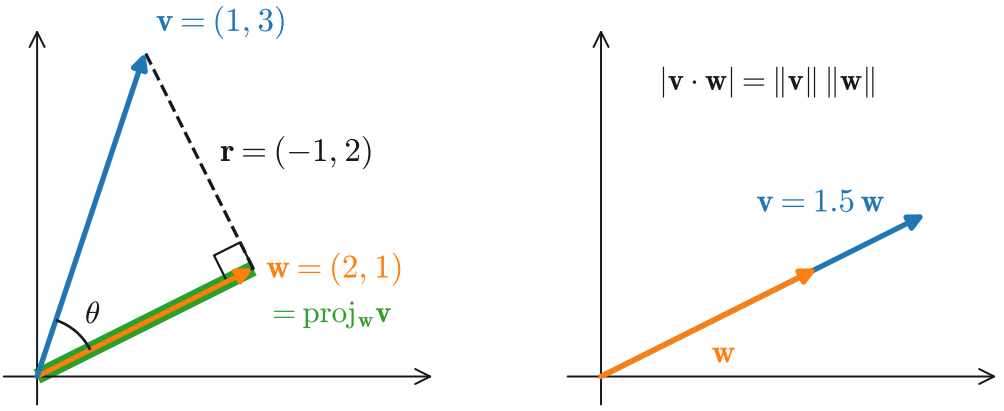


Figure 33.3: (a) The projection (shadow) of $\mathbf{v} = (1,3)$ onto $\mathbf{w} = (2,1)$ is $(2,1)$; the residual $(-1,2)$ meets $\mathbf{w}$ at a right angle, so $\|\mathbf{v}\|^2 = 10 = 5+5$. (b) The Cauchy–Schwarz equality case: $\mathbf{v}$ is collinear with $\mathbf{w}$ and the residual vanishes.

33.1.1.3 Cauchy–Schwarz and the Angle

In 784 dimensions there is no protractor. Can "the angle between two images" still mean anything? Surprisingly, the answer comes from a much simpler question: can the shadow of a vector ever be longer than the vector itself? Panel (a) of Figure 33.3 says no, because the shadow is a leg of a right triangle whose hypotenuse is $\mathbf{v}$. Pythagoras' theorem turns that picture into a proof that works in every dimension.

Proposition (Cauchy–Schwarz). *For all vectors* $\mathbf{v}, \mathbf{w}$*,*

$$|\mathbf{v}\cdot\mathbf{w}| \le \|\mathbf{v}\|\,\|\mathbf{w}\|, \tag{33.5}$$

with equality if and only if $\mathbf{v}$ *and* $\mathbf{w}$ *are collinear (one is a scalar multiple of the other).*

Proof. If $\mathbf{w} = \mathbf{0}$, both sides vanish and the vectors are collinear. Otherwise, split $\mathbf{v}$ into its projection onto $\mathbf{w}$ and its orthogonal residual $\mathbf{r}$. By Pythagoras' theorem Equation 33.3,

$$\|\mathbf{v}\|^2 = \|\operatorname{proj}_{\mathbf{w}} \mathbf{v}\|^2 + \|\mathbf{r}\|^2 \geq \|\operatorname{proj}_{\mathbf{w}} \mathbf{v}\|^2 = \frac{(\mathbf{v}\cdot\mathbf{w})^2}{\|\mathbf{w}\|^2}.$$

Multiplying both sides by $\|\mathbf{w}\|^2$ and taking square roots yields the inequality. Equality holds precisely when $\mathbf{r} = \mathbf{0}$, meaning $\mathbf{v} = t^\star\mathbf{w}$. Conversely, if $\mathbf{v} = t\mathbf{w}$, then $|\mathbf{v}\cdot\mathbf{w}| = |t|\,\|\mathbf{w}\|^2 = \|\mathbf{v}\|\,\|\mathbf{w}\|$. ■

In plain terms, a vector's shadow is never longer than the vector itself, and matches its length only when nothing is left over in the residual. Where Section 1.3 proved Cauchy–Schwarz algebraically, here it falls straight out of panel (a) of Figure 33.3, with panel (b) showing the equality case.

Now we can define angles in any dimension. In the plane, the law of cosines gives $\mathbf{v}\cdot\mathbf{w} = \|\mathbf{v}\|\|\mathbf{w}\|\cos\theta$ (Section 1.3). In $\mathbb{R}^d$ we have no physical triangle to measure, so we turn this identity into a definition. For nonzero vectors $\mathbf{v}, \mathbf{w} \in \mathbb{R}^d$, the *angle* between them is

$$\theta = \arccos\left(\frac{\mathbf{v}\cdot\mathbf{w}}{\|\mathbf{v}\|\|\mathbf{w}\|}\right) \in [0, \pi]. \tag{33.6}$$

Because arccos is defined only on $[-1, 1]$, this formula makes sense only if the ratio lies inside that interval, and Cauchy–Schwarz guarantees that it always does. Thus the angle is well defined in every dimension, and the planar dot-product formula now holds everywhere as an immediate consequence:

$$\mathbf{v}\cdot\mathbf{w} = \|\mathbf{v}\|\|\mathbf{w}\|\cos\theta. \tag{33.7}$$

The two equality cases of Cauchy–Schwarz correspond to $\theta = 0$ (vectors pointing in the same direction) and $\theta = \pi$ (vectors pointing in opposite directions).

A natural objection arises: does this algebraic definition agree with the angle a protractor would actually measure? It does. Any two vectors lie inside a single plane (or a line, if they are collinear), and inside that plane ordinary geometry applies. Projection gives us rectangular coordinates for that plane: set $\mathbf{q}_1 = \mathbf{w}/\|\mathbf{w}\|$ and, provided $\mathbf{r} \neq \mathbf{0}$, set $\mathbf{q}_2 = \mathbf{r}/\|\mathbf{r}\|$. These two vectors have unit length and are orthogonal, so dot products of their combinations behave just like dot products in $\mathbb{R}^2$. In this coordinate system, $\mathbf{w} = (\|\mathbf{w}\|, 0)$ and $\mathbf{v} = (\mathbf{v}\cdot\mathbf{w}/\|\mathbf{w}\|,\ \|\mathbf{r}\|)$. The angle a protractor measures between these two planar arrows has cosine $(\mathbf{v}\cdot\mathbf{w}/\|\mathbf{w}\|)/\|\mathbf{v}\|$, matching Equation 33.6. For our worked pair $\mathbf{v} = (1, 3)$ and $\mathbf{w} = (2, 1)$, we get $\cos\theta = 5/(\sqrt{10}\sqrt{5}) = 1/\sqrt{2}$, so $\theta = 45^\circ$.

For nonzero vectors, orthogonality means $\theta = \pi/2$. Defining orthogonality via $\mathbf{v}\cdot\mathbf{w} = 0$ has the minor convenience of including the zero vector as well: it is orthogonal to every vector, even though no angle is defined for it.

Turning Equation 33.6 into code takes only two lines, with one practical safeguard worth noticing. Let us test it on our worked pair and on a deliberately tricky collinear pair, $\mathbf{v} = (10^8 + 1, 3, 7)$ and $3\mathbf{v}$, whose true cosine is 1 and whose true angle is 0. For the collinear pair, we also print the raw cosine minus 1 and the unclipped `arccos`.

```
def angle(v, w):
    cos = v @ w / (np.linalg.norm(v) * np.linalg.norm(w))
    return np.arccos(np.clip(cos, -1, 1))  # roundoff can push |cos| past 1

v = np.array([1e8 + 1, 3, 7])
cos = v @ (3 * v) / (np.linalg.norm(v) * np.linalg.norm(3 * v))
with np.errstate(invalid='ignore'):        # silence the warning for nan
    print(angle(np.array([1., 3]), np.array([2., 1])), cos - 1,
          np.arccos(cos), angle(v, 3 * v))
```

```
0.7853981633974484 2.220446049250313e-16 nan 0.0
```

The first number is $0.785 = \pi/4$, matching the 45° angle of our worked pair. The remaining three values show why `np.clip` is essential. In exact arithmetic, Cauchy–Schwarz keeps the ratio inside $[-1, 1]$. In floating-point arithmetic, rounding can nudge it past the boundary by one unit in the last place: here the collinear pair yields $1 + 2.2 \times 10^{-16}$, causing `np.arccos` to return `nan`. Clipping to $[-1, 1]$ restores the right answer, 0.

Cauchy–Schwarz also shows that the length of a sum never exceeds the sum of the lengths. Expanding $\|\mathbf{v}+\mathbf{w}\|^2$ and bounding the cross term $2\,\mathbf{v}\cdot\mathbf{w}$ by $2\,\|\mathbf{v}\|\|\mathbf{w}\|$ via Equation 33.5 yields the *triangle inequality*, $\|\mathbf{v}+\mathbf{w}\| \le \|\mathbf{v}\| + \|\mathbf{w}\|$, just as in Section 1.3.

33.1.2 Subspaces and Least Squares

33.1.2.1 Span, Bases, and Subspaces

Suppose we measure a quantity three times, at $t = 0, 1, 2$, and want to fit the best straight line $C + Dt$ to our data. Stack the measurements into a vector $\mathbf{y} \in \mathbb{R}^3$, collect the times in $\mathbf{t} = (0, 1, 2)$, and let $\mathbf{1} = (1, 1, 1)$. Evaluating the line at the three times produces the vector $C\mathbf{1} + D\mathbf{t}$. Finding the best line therefore means finding the vector of the form $C\mathbf{1} + D\mathbf{t}$ that lies closest to $\mathbf{y}$. As C and D vary over all real numbers, these candidate vectors sweep out a plane through the origin in $\mathbb{R}^3$. Projection onto a line handled a single direction; to handle planes and their higher-dimensional counterparts, we need a small set of standard terms. Readers already comfortable with span, bases, and subspaces can skip straight to the next subsection, where we finish the line fit.

A weighted sum $a_1\mathbf{v}_1 + \cdots + a_k\mathbf{v}_k$ is called a *linear combination* of $\mathbf{v}_1, \dots, \mathbf{v}_k$. The collection of all such combinations is their *span*,

$$\operatorname{span}(\mathbf{v}_1, \dots, \mathbf{v}_k) = \{a_1\mathbf{v}_1 + \cdots + a_k\mathbf{v}_k : a_1, \dots, a_k \in \mathbb{R}\}.$$

In $\mathbb{R}^2$ (Figure 33.4), the span of a single nonzero vector is the line through the origin along its direction, while the span of two vectors not lying on a common line through the origin is the entire plane. In our line-fitting problem, the set of all candidate fits is $\operatorname{span}(\mathbf{1}, \mathbf{t})$.

Notice that a span is always closed under addition and scalar multiplication: adding two linear combinations, or scaling one by a real number, produces another linear combination. Any nonempty set of vectors with these two closure properties is called a *subspace*. Every subspace must contain the zero vector $\mathbf{0}$ (just multiply any element by 0), which means a line or plane that misses the origin is not a subspace. In $\mathbb{R}^2$, the only subspaces are the origin alone, the lines through the origin, and $\mathbb{R}^2$ itself; in $\mathbb{R}^3$, planes through the origin join the list.

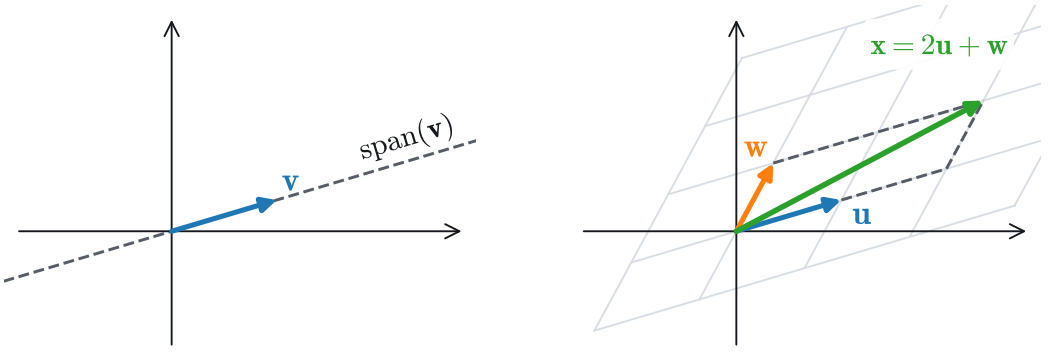


Figure 33.4: Left: the span of a single nonzero vector **v** is the set of all its scalar multiples, forming a line through the origin and thus a one-dimensional subspace. Right: two vectors not on a common line through the origin span the whole plane; the dashed parallelogram resolves **x** as $2\mathbf{u} + \mathbf{w}$, with unique coordinates because **u** and **w** are linearly independent.

A list of vectors that spans a subspace can contain redundancies. Whenever one vector already lies in the span of the others, we can drop it without shrinking the span. A list of vectors is *linearly independent* when no such redundancy exists, meaning that the only linear combination $a_1\mathbf{v}_1 + \cdots + a_k\mathbf{v}_k$ equal to $\mathbf{0}$ is the trivial one with every $a_i = 0$. Our vectors $\mathbf{1}$ and $\mathbf{t}$ are linearly independent: $a_1\mathbf{1} + a_2\mathbf{t} = (a_1,\, a_1 + a_2,\, a_1 + 2a_2)$ vanishes only when $a_1 = a_2 = 0$.

A *basis* of a subspace is a linearly independent list of vectors that spans it. The familiar vectors $\mathbf{e}_1 = (1, 0)$ and $\mathbf{e}_2 = (0, 1)$ form the *standard basis* of $\mathbb{R}^2$, though the slanted pair in Figure 33.4 is an equally good basis. A fundamental theorem of linear algebra (which we use without proof) states that every basis of a given subspace has the same number of vectors; that shared count is the subspace's *dimension*. This gives a crisp meaning to the d in "d-dimensional space": $\mathbb{R}^d$ has dimension d because $\mathbf{e}_1, \ldots, \mathbf{e}_d$ is a basis. Likewise, our candidate line fits form a two-dimensional subspace of $\mathbb{R}^3$.

Linear independence is what gives every point in a subspace unique coordinates. If $\mathbf{v}_1, \ldots, \mathbf{v}_k$ is a basis of a subspace S, then every $\mathbf{x} \in S$ can be written as $\mathbf{x} = a_1\mathbf{v}_1 + \cdots + a_k\mathbf{v}_k$ in only one way. At least one representation exists because the basis spans S. If two representations existed with coefficients a_i and b_i, subtracting them would give $\sum_i (a_i - b_i)\,\mathbf{v}_i = \mathbf{0}$, and independence forces $a_i = b_i$ for every i.

Once we fix a basis, a k-dimensional subspace behaves just like $\mathbb{R}^k$: each point is uniquely named by its coefficient tuple $(a_1, \ldots, a_k)$. For our line fit, those coordinates are simply the intercept C and the slope D.

33.1.2.2 Projection onto a Subspace

Let us return to the line fit with concrete measurements $\mathbf{y} = (6, 0, 0)$. We want the point $C\mathbf{1} + D\mathbf{t}$ in the plane $\operatorname{span}(\mathbf{1}, \mathbf{t})$ closest to $\mathbf{y}$. When we projected onto a single line, the closest point left a residual orthogonal to that line. Projecting onto a plane should leave a residual orthogonal to the entire plane, and therefore orthogonal to both $\mathbf{1}$ and $\mathbf{t}$ (a fact we prove in a moment). Setting both dot products to zero gives two linear equations in two unknowns:

$$\mathbf{1} \cdot (\mathbf{y} - C\mathbf{1} - D\mathbf{t}) = 6 - 3C - 3D = 0,$$
$$\mathbf{t} \cdot (\mathbf{y} - C\mathbf{1} - D\mathbf{t}) = 0 - 3C - 5D = 0.$$

Subtracting the first equation from the second yields $2D = -6$, so $D = -3$ and $C = 5$. The best-fitting line is $5 - 3t$. Its predicted values at $t = 0, 1, 2$ are $(5, 2, -1)$, leaving the residual $\mathbf{r} = \mathbf{y} - (5, 2, -1) = (1, -2, 1)$. Both orthogonality checks hold: $\mathbf{r} \cdot \mathbf{1} = 1 - 2 + 1 = 0$ and $\mathbf{r} \cdot \mathbf{t} = 0 - 2 + 2 = 0$. Pythagoras' theorem again splits the squared length of $\mathbf{y}$ cleanly: $36 = 30 + 6$. By comparison, projecting $\mathbf{y}$ onto $\mathbf{1}$ alone yields the constant fit $\mathbf{y} \cdot \mathbf{1}/\mathbf{1} \cdot \mathbf{1} = 2$ (the sample mean), whose residual $(4, -2, -2)$ has squared length 24. Allowing a slope along $\mathbf{t}$ reduces the squared error from 24 down to 6. Figure 33.5 shows this projection in $\mathbb{R}^3$.

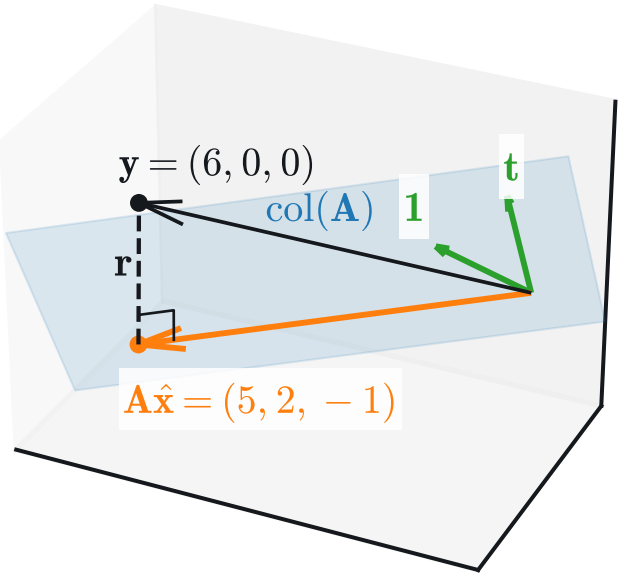


Figure 33.5: Least squares as projection: the measurements $\mathbf{y} = (6, 0, 0)$ lie outside the plane of candidate lines $C\mathbf{1} + D\mathbf{t}$; the closest point in the plane is $(5, 2, -1)$ (the line $5 - 3t$), and the residual $(1, -2, 1)$ is perpendicular to both $\mathbf{1}$ and $\mathbf{t}$.

To state and prove the general result cleanly, it helps to choose a basis whose vectors are mutually orthogonal and have unit length. A basis $\mathbf{q}_1, \dots, \mathbf{q}_k$ of a subspace $S \subseteq \mathbb{R}^n$ is *orthonormal* when every vector satisfies $\|\mathbf{q}_i\| = 1$ and distinct vectors satisfy $\mathbf{q}_i \cdot \mathbf{q}_j = 0$. Every finite spanning list admits such a basis: Section 33.1.2.3 constructs one using nothing more than projection onto a line. If we place these basis vectors as the columns of an $n \times k$ matrix $\mathbf{Q}$, orthonormality takes the compact form $\mathbf{Q}^\top \mathbf{Q} = \mathbf{I}_k$, because entry (i, j) of $\mathbf{Q}^\top \mathbf{Q}$ is $\mathbf{q}_i \cdot \mathbf{q}_j$. Throughout this chapter, we reserve the letter $\mathbf{Q}$ for a matrix with orthonormal columns. The following proposition is the workhorse result that the rest of the chapter draws on most.

Proposition (projection onto a subspace). *Let* $\mathbf{Q}$ *have orthonormal columns spanning* S*, and set* $\mathbf{P} = \mathbf{Q}\mathbf{Q}^\top$*. Then for every* $\mathbf{x} \in \mathbb{R}^n$*:* (i) $\mathbf{P}\mathbf{x} \in S$*, and the residual* $\mathbf{x} - \mathbf{P}\mathbf{x}$ *is orthogonal to every vector of* S*;* (ii) $\mathbf{P}\mathbf{x}$ *is the unique closest point of* S *to* $\mathbf{x}$*;* (iii) $\mathbf{P}^2 = \mathbf{P}$ *and* $\mathbf{P}^\top = \mathbf{P}$*.*

Proof. For (i), $\mathbf{P}\mathbf{x} = \mathbf{Q}(\mathbf{Q}^\top\mathbf{x})$ is a linear combination of the columns of $\mathbf{Q}$, so it lies in S. Multiplying the residual on the left by $\mathbf{Q}^\top$ gives

$$\mathbf{Q}^\top(\mathbf{x} - \mathbf{P}\mathbf{x}) = \mathbf{Q}^\top\mathbf{x} - (\mathbf{Q}^\top\mathbf{Q})\,\mathbf{Q}^\top\mathbf{x} = \mathbf{0},$$

so the residual is orthogonal to every column $\mathbf{q}_i$, and therefore to every linear combination of them. For (ii), take any $\mathbf{s} \in S$ and write $\mathbf{x} - \mathbf{s} = (\mathbf{x} - \mathbf{P}\mathbf{x}) + (\mathbf{P}\mathbf{x} - \mathbf{s})$. Since $\mathbf{P}\mathbf{x} - \mathbf{s}$ lies in S while $\mathbf{x} - \mathbf{P}\mathbf{x}$ is orthogonal to S, Pythagoras' theorem Equation 33.3 gives

$$\|\mathbf{x} - \mathbf{s}\|^2 = \|\mathbf{x} - \mathbf{P}\mathbf{x}\|^2 + \|\mathbf{P}\mathbf{x} - \mathbf{s}\|^2 \geq \|\mathbf{x} - \mathbf{P}\mathbf{x}\|^2,$$

with equality only when $\mathbf{s} = \mathbf{P}\mathbf{x}$. For (iii), $\mathbf{P}^2 = \mathbf{Q}(\mathbf{Q}^\top\mathbf{Q})\mathbf{Q}^\top = \mathbf{Q}\mathbf{Q}^\top = \mathbf{P}$, and $\mathbf{P}^\top = (\mathbf{Q}^\top)^\top\mathbf{Q}^\top = \mathbf{P}$. ■

Conversely, the orthogonality condition in (i) uniquely identifies the closest point. If some $\mathbf{s} \in S$ leaves a residual $\mathbf{x} - \mathbf{s}$ orthogonal to S, then the difference $\mathbf{P}\mathbf{x} - \mathbf{s} = (\mathbf{x} - \mathbf{s}) - (\mathbf{x} - \mathbf{P}\mathbf{x})$ both lies in S and is orthogonal to S, hence orthogonal to itself, which forces $\mathbf{s} = \mathbf{P}\mathbf{x}$. That is why solving the two orthogonality equations in our line fit was guaranteed to produce the closest point.

Each half of property (iii) has a clear geometric meaning. The identity $\mathbf{P}^2 = \mathbf{P}$ says that projecting a second time changes nothing, since the first projection already landed in S. Symmetry, $\mathbf{P}^\top = \mathbf{P}$, is what makes the projection *orthogonal*. A matrix satisfying $\mathbf{P}^2 = \mathbf{P}$ without symmetry is an *oblique* projector: it maps vectors into its column space along a slanted direction and generally misses the closest point (3). In the one-dimensional case $k = 1$, taking the single unit column $\mathbf{q} = \mathbf{w}/\|\mathbf{w}\|$ reduces $\mathbf{P} = \mathbf{q}\mathbf{q}^\top$ directly to our earlier line projection Equation 33.2.

Our line fit is one example of *least squares*, the optimization problem at the heart of linear regression. Let $\mathbf{A}$ be a data matrix whose rows are examples and whose columns are features, and let $\mathbf{y}$ be the vector of targets. Least squares seeks the coefficient vector $\mathbf{x}$ that minimizes the squared error:

$$\min_{\mathbf{x}} \|\mathbf{A}\mathbf{x} - \mathbf{y}\|^2. \tag{33.8}$$

As $\mathbf{x}$ varies over all coefficient vectors, $\mathbf{A}\mathbf{x}$ traces out the span of the columns of $\mathbf{A}$, known as its *column space* $\mathrm{col}(\mathbf{A})$. Least squares therefore asks for the point in $\mathrm{col}(\mathbf{A})$ closest to $\mathbf{y}$. By our projection proposition, that point is the one whose residual $\mathbf{y} - \mathbf{A}\hat{\mathbf{x}}$ is orthogonal to every column of $\mathbf{A}$. Writing this orthogonality condition as $\mathbf{A}^\top(\mathbf{y} - \mathbf{A}\hat{\mathbf{x}}) = \mathbf{0}$ produces the *normal equations*,

$$\mathbf{A}^\top\mathbf{A}\,\hat{\mathbf{x}} = \mathbf{A}^\top\mathbf{y}. \tag{33.9}$$

These are the matrix form of the 2×2 system we solved by hand: for the line fit, $\mathbf{A} = [\mathbf{1}\ \mathbf{t}]$ gives $\mathbf{A}^\top\mathbf{A} = \left(\begin{smallmatrix} 3 & 3 \\ 3 & 5 \end{smallmatrix}\right)$ and $\mathbf{A}^\top\mathbf{y} = (6, 0)$. Whenever the columns of $\mathbf{A}$ are linearly independent, $\mathbf{A}^\top\mathbf{A}$ is invertible and the optimal coefficients $\hat{\mathbf{x}}$ are unique. Even when the columns are

dependent, the fitted vector $\mathbf{A}\hat{\mathbf{x}}$ remains unique because it is still the closest point in $\mathrm{col}(\mathbf{A})$, though more than one coefficient vector $\hat{\mathbf{x}}$ can produce it.

In code, we therefore have two ways to compute the fit $5 - 3t$: solve the normal equations directly, or project using $\mathbf{P} = \mathbf{Q}\mathbf{Q}^\top$. To obtain $\mathbf{Q}$, we call `np.linalg.qr`, which builds an orthonormal basis for the column space of a matrix with independent columns (we will construct one by hand in the next subsection).

```
A = np.array([[1., 0], [1, 1], [1, 2]])     # columns 1 and t
y = np.array([6., 0, 0])
x_hat = np.linalg.solve(A.T @ A, A.T @ y)   # normal equations
Q, R = np.linalg.qr(A)                      # orthonormal basis of col(A)
P = Q @ Q.T
r = y - P @ y
print(x_hat.round(6), (P @ y).round(6), r.round(6))
print(np.linalg.norm(P @ P - P), np.linalg.norm(Q.T @ r))
```

```
[ 5. -3.] [ 5.  2. -1.] [ 1. -2.  1.]
1.962615573354719e-16 0.0
```

Both routes recover our hand calculation: the coefficients $C = 5$ and $D = -3$, the projected values $(5, 2, -1)$, and the residual $(1, -2, 1)$. The two check norms on the second line vanish up to floating-point roundoff, confirming $\mathbf{P}^2 = \mathbf{P}$ and $\mathbf{Q}^\top \mathbf{r} = \mathbf{0}$.

On a tiny, well-behaved problem like this, the two methods agree completely. When the columns of $\mathbf{A}$ are nearly dependent, however, forming $\mathbf{A}^\top \mathbf{A}$ squares the problem's condition number, its sensitivity to rounding errors (as we will show in Section 33.5.3.2). For that reason, production least-squares solvers avoid the normal equations and work with an orthonormal basis of the column space instead.

33.1.2.3 Gram–Schmidt and QR

How do we actually build an orthonormal basis? By repeated projection. Given two linearly independent vectors, keep the direction of the first and subtract from the second its shadow along the first. By our projection proposition, whatever remains is orthogonal to the first vector.

Let us carry this out on $\mathbf{a}_1 = (3, 4)$ and $\mathbf{a}_2 = (1, 0)$. Normalizing $\mathbf{a}_1$ gives $\mathbf{q}_1 = \mathbf{a}_1/5 = (0.6, 0.8)$. The shadow of $\mathbf{a}_2$ onto $\mathbf{q}_1$ is $(\mathbf{q}_1 \cdot \mathbf{a}_2)\,\mathbf{q}_1 = 0.6\,(0.6, 0.8) = (0.36, 0.48)$. Subtracting it leaves $(1, 0) - (0.36, 0.48) = (0.64, -0.48)$, a vector of length 0.8 that is indeed orthogonal to $\mathbf{q}_1$ ($0.384 - 0.384 = 0$). Normalizing this remainder gives $\mathbf{q}_2 = (0.8, -0.6)$. Reading the steps in reverse writes each original vector in terms of the new orthonormal basis:

$$\mathbf{a}_1 = 5\,\mathbf{q}_1, \qquad \mathbf{a}_2 = 0.6\,\mathbf{q}_1 + 0.8\,\mathbf{q}_2.$$

In matrix notation, this is $\mathbf{A} = \mathbf{Q}\mathbf{R}$, where $\mathbf{A} = [\mathbf{a}_1\ \mathbf{a}_2]$, $\mathbf{Q} = [\mathbf{q}_1\ \mathbf{q}_2]$, and $\mathbf{R} = \left(\begin{smallmatrix} 5 & 0.6 \\ 0 & 0.8 \end{smallmatrix}\right)$ collects the dot products $\mathbf{q}_i \cdot \mathbf{a}_j$. Notice why $\mathbf{R}$ is upper triangular: by construction, each $\mathbf{a}_j$ depends only on $\mathbf{q}_1, \dots, \mathbf{q}_j$.

Extending this idea to any linearly independent list $\mathbf{a}_1, \dots, \mathbf{a}_k$ gives the *Gram–Schmidt process*. We process the vectors in order: for each $\mathbf{a}_j$, subtract its projections Equation 33.2 onto

the unit vectors $\mathbf{q}_1, \ldots, \mathbf{q}_{j-1}$ already constructed, and normalize the remainder to obtain $\mathbf{q}_j$. The result is both an orthonormal basis for $\operatorname{span}(\mathbf{a}_1, \ldots, \mathbf{a}_k)$ and the *QR factorization* $\mathbf{A} = \mathbf{QR}$, where $\mathbf{Q}$ has orthonormal columns and $\mathbf{R}$ is upper triangular with positive diagonal entries. Linear independence is essential for the factorization, because a vector that already lies in the span of its predecessors leaves a zero remainder that cannot be normalized. If we only want an orthonormal basis for the span, we can simply skip any vector with a zero remainder and continue. Every finite spanning list therefore yields an orthonormal basis, justifying the existence claim we relied on earlier.

Numerical libraries compute the QR factorization using a different algorithm based on reflections, which behaves much better in floating-point arithmetic. Its output matches Gram–Schmidt only up to signs: negating a column of $\mathbf{Q}$ together with the corresponding row of $\mathbf{R}$ leaves the product $\mathbf{QR}$ untouched. When $\mathbf{A}$ has independent columns, those sign flips are the only ambiguity. Let us compare our two-step Gram–Schmidt calculation against `np.linalg.qr` and verify that they produce the same columns up to sign.

```
a1, a2 = np.array([3., 4]), np.array([1., 0])
q1 = a1 / np.linalg.norm(a1)
v = a2 - (q1 @ a2) * q1                    # remove the shadow on q1
q2 = v / np.linalg.norm(v)
Q_qr, R_qr = np.linalg.qr(np.stack([a1, a2], axis=1))
print(np.stack([q1, q2], axis=1).round(6))
print(Q_qr.round(6))
print(R_qr.round(6))
```

```
[[ 0.6  0.8]
 [ 0.8 -0.6]]
[[-0.6 -0.8]
 [-0.8  0.6]]
[[-5.  -0.6]
 [ 0.  -0.8]]
```

Gram–Schmidt produces $(0.6, 0.8)$ and $(0.8, -0.6)$, while `np.linalg.qr` returns the negatives of both columns along with diagonal entries -5 and -0.8 in $\mathbf{R}$: the same basis and, up to sign, the same dot products 5, 0.6, and 0.8 we found by hand. In floating point, the order in which we perform the subtractions also matters. On nearly dependent columns, the formulation above, known as *classical* Gram–Schmidt, gradually loses orthogonality (5).

33.1.3 Similarity in High Dimensions

Up to this point, every result came with a picture in $\mathbb{R}^2$ or $\mathbb{R}^3$. Now we move to spaces where no picture is possible, asking how similar two real images are and what similarity we should expect between unrelated vectors by pure chance. We load the Fashion-MNIST dataset (Section 3.2) through `d2l.FashionMNIST` as NumPy arrays, flattening each image into a 784-pixel row scaled to $[0, 1]$.

```
data = d2l.FashionMNIST()
X_train = data.train.data.numpy().reshape(-1, 784) / 255
y_train = data.train.targets.numpy()
X_test = data.val.data.numpy().reshape(-1, 784) / 255
y_test = data.val.targets.numpy()
```

```
X_train.shape, X_test.shape
```

```
((60000, 784), (10000, 784))
```

Each of these $70{,}000$ images is now a point in $\mathbb{R}^{784}$.

33.1.3.1 Cosine Similarity

The ratio inside Equation 33.6 is the *cosine similarity* (Section 1.3),

$$\cos\theta = \frac{\mathbf{v}\cdot\mathbf{w}}{\|\mathbf{v}\|\,\|\mathbf{w}\|} \;\in\; [-1, 1]. \tag{33.10}$$

Cosine similarity compares directions while ignoring lengths: an image and a copy of it dimmed to 10% brightness sit far apart in Euclidean distance, yet their cosine similarity is still 1.

That scale invariance makes cosine similarity the standard metric for nearest-neighbor retrieval over embeddings and for contrastive learning objectives such as SimCLR (Chen et al. 2020), which align the directions of embeddings produced from two views of the same image. The attention layer of the original transformer makes a different choice: it uses the *unnormalized* dot product scaled by $1/\sqrt{d}$ (for reasons we derive shortly), so vector lengths remain informative there. Many later models normalize queries and keys first (Section 11.1.2.5), which turns the score back into a scaled cosine.

33.1.3.2 Near-Orthogonality of Random Directions

Suppose we pick two unrelated images or two unrelated embeddings. What cosine similarity should we expect between them purely by chance? Without a baseline, we cannot judge whether an observed cosine is meaningfully large. The cleanest baseline is a direction chosen uniformly at random on the unit sphere, with no preference for any region of space.

Proposition (near-orthogonality). *Fix a unit vector* $\mathbf{u} \in \mathbb{R}^d$ *and let* $\mathbf{v}$ *be drawn uniformly from the unit sphere in* $\mathbb{R}^d$. *Then* $\cos\theta = \mathbf{u}\cdot\mathbf{v}$ *satisfies*

$$\mathbb{E}[\cos\theta] = 0, \qquad \mathrm{Var}(\cos\theta) = \frac{1}{d}, \qquad \Pr\big(|\cos\theta| \geq \varepsilon\big) \leq \frac{1}{d\,\varepsilon^2} \quad \text{for every } \varepsilon > 0.$$

Proof. Because the uniform distribution on the sphere is invariant under rotations, we may rotate coordinates so that $\mathbf{u} = \mathbf{e}_1$; then $\cos\theta = v_1$. The distribution is also unchanged by sign reversal $\mathbf{v} \mapsto -\mathbf{v}$, so $\mathbb{E}[v_1] = 0$. By symmetry, every coordinate has the same second moment $\mathbb{E}[v_i^2]$. Since $\sum_{i=1}^{d} v_i^2 = \|\mathbf{v}\|^2 = 1$, taking expectations yields $d\,\mathbb{E}[v_1^2] = \mathbb{E}[\sum_{i=1}^{d} v_i^2] = 1$, and therefore $\mathrm{Var}(\cos\theta) = \mathbb{E}[v_1^2] = 1/d$. The tail bound follows immediately from Chebyshev's inequality, $\Pr(|X - \mathbb{E}[X]| \geq \varepsilon) \leq \mathrm{Var}(X)/\varepsilon^2$ (Section 1.6). ■

The typical cosine between two random directions is therefore of order $1/\sqrt{d}$. In $d = 784$, the standard deviation is $1/28 \approx 0.036$, meaning two random directions in image space are almost always within a few degrees of orthogonal. Chebyshev's inequality bounds the

probability of seeing $|\cos\theta| \geq 0.1$ by $1/(784 \cdot 0.01) \approx 0.13$, whereas the true probability is under 0.01. Chebyshev is loose because it uses only the variance; sharper exponential bounds appear in Section 36.9.

Low dimensions look completely different (Figure 33.6). In the plane ($d = 2$), the angle between a fixed direction and a random one is uniform on $[0, \pi]$, so landing within 0.1 radians of orthogonal is just as likely as landing within 0.1 radians of collinear (about 6% each). Yet the *cosine* piles up near ± 1, because $\cos\theta$ flattens out near $\theta = 0$ and $\theta = \pi$: the probability that $|\cos\theta| > 0.9$ is about 0.29, compared with only 0.06 for $|\cos\theta| < 0.1$. In $d = 3$, the cosine is uniform on $[-1, 1]$, a classical result known to Archimedes. Starting at $d = 4$, the density peaks at 0; by $d = 10$ it forms a bell of width $1/\sqrt{10} \approx 0.32$, and by $d = 1000$ it has narrowed into a sharp spike of width roughly 0.03.

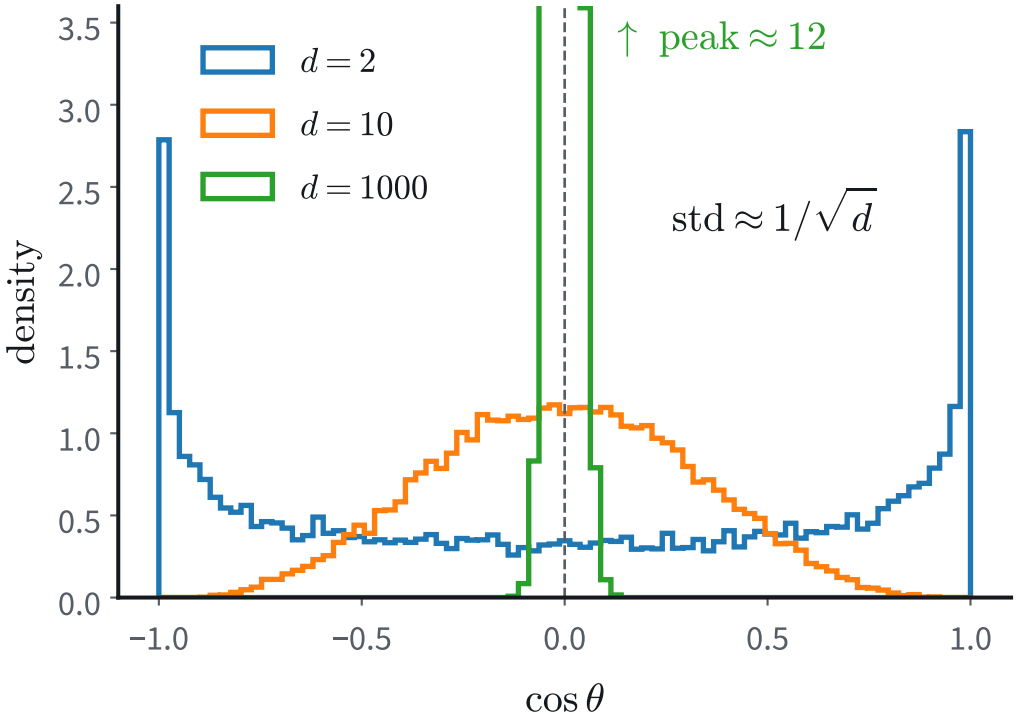


Figure 33.6: Histograms of the cosine between independent random unit vectors in dimensions $d = 2$, 10, and 1000. In $d = 2$, the cosine piles up near ± 1 even though the angle is uniform. By $d = 10$ it forms a bell around 0, and by $d = 1000$ it concentrates into a narrow spike of width $\approx 1/\sqrt{d}$, matching the proposition.

Let us check this isotropic baseline in $d = 784$ by simulation. Because a standard Gaussian vector's distribution is rotationally invariant, normalizing it produces a uniformly random unit vector. In the same cell, we also estimate a variance we will need for attention in a moment: the variance of $\mathbf{q} \cdot \mathbf{k}$ for independent standard normal vectors in 64 dimensions.

```
rng = np.random.default_rng()
d = 784
```

```
U = rng.standard_normal((10000, d))
U /= np.linalg.norm(U, axis=1, keepdims=True)
V = rng.standard_normal((10000, d))
V /= np.linalg.norm(V, axis=1, keepdims=True)
q, k = rng.standard_normal((10000, 64)), rng.standard_normal((10000, 64))
np.std((U * V).sum(axis=1)), 1 / np.sqrt(d), np.var((q * k).sum(axis=1))
```

```
(np.float64(0.03602372011587368),
 np.float64(0.03571428571428571),
 np.float64(65.51162136338216))
```

The empirical standard deviation matches $1/\sqrt{784} \approx 0.036$, and the variance of the unnormalized dot product is close to 64, the dimension.

Real data, however, rarely look isotropic. Do two unrelated Fashion-MNIST images behave like nearly orthogonal vectors? Let us sample $5{,}000$ random pairs of test images and measure their cosines, first on the raw pixels and then after subtracting the mean training image.

```
def cosines(A, B):
    """Cosine similarity between corresponding rows of A and B."""
    return (A * B).sum(axis=1) / (np.linalg.norm(A, axis=1)
                                  * np.linalg.norm(B, axis=1))

i, j = rng.integers(0, len(X_test), (2, 5000))
mean_image = X_train.mean(axis=0)
raw = cosines(X_test[i], X_test[j])
centered = cosines(X_test[i] - mean_image, X_test[j] - mean_image)
raw.mean(), raw.std(), centered.mean(), centered.std()
```

```
(np.float64(0.592208289545085),
 np.float64(0.17608271528452482),
 np.float64(-0.004695197414552555),
 np.float64(0.3390460995081151))
```

Not even close. Raw images have an average cosine around 0.6 with a standard deviation of roughly 0.18, nowhere near 0 ± 0.036. The reason is simple: pixel intensities are nonnegative, so every image lives in the same positive orthant of $\mathbb{R}^{784}$, making negative dot products impossible. Subtracting the mean image centers the cosines at 0, yet their standard deviation, about 0.34, is still nearly ten times the isotropic value. By our proposition, a spread of 0.34 corresponds to random directions in only eight or nine dimensions ($1/0.34^2$ lies between 8 and 9). Judged by their cosines, the centered images behave like a cloud living in fewer than ten dimensions rather than 784, an early glimpse of the low-rank structure we will study in Section 33.5. Practically, this means a cosine counts as large only when it stands out above the spread of unrelated pairs from the *same* dataset; training an embedding model is what pushes related pairs above that background.

33.1.3.3 The Scale of Attention Scores

Dot-product attention (Section 10.2) scores a query $\mathbf{q} \in \mathbb{R}^d$ against keys $\mathbf{k}_1, \mathbf{k}_2, \ldots$ using $\mathbf{q} \cdot \mathbf{k}_j / \sqrt{d}$ and passes those scores through a softmax (Vaswani et al. 2017). A brief variance

calculation explains where the $\sqrt{d}$ comes from. Suppose all $2d$ entries of $\mathbf{q}$ and $\mathbf{k}$ are mutually independent with mean 0 and variance 1. Each product $q_i k_i$ then has mean 0 and variance $\mathbb{E}[q_i^2]\,\mathbb{E}[k_i^2] = 1$, and distinct terms are uncorrelated because $\mathbb{E}[(q_i k_i)(q_j k_j)] = \mathbb{E}[q_i]\,\mathbb{E}[k_i]\,\mathbb{E}[q_j]\,\mathbb{E}[k_j] = 0$ for $i \neq j$. The variances of the d terms therefore add:

$$\operatorname{Var}(\mathbf{q} \cdot \mathbf{k}) = \sum_{i=1}^{d} \operatorname{Var}(q_i k_i) = d.$$

Thus the raw dot products have standard deviation $\sqrt{d}$, which grows with the head dimension; dividing by $\sqrt{d}$ restores unit variance. Without that scaling, the softmax saturates into near-one-hot weights. For $d = 64$, raw scores have standard deviation 8, and two scores separated by 8 receive softmax weights 0.9997 and 0.0003; dividing by $\sqrt{d} = 8$ brings the weights back to a healthy 0.73 and 0.27. Geometry gives the exact same scaling: each vector has length roughly $\sqrt{d}$, while the cosine between unrelated directions is roughly $1/\sqrt{d}$, so their dot product has typical magnitude $\sqrt{d} \cdot \sqrt{d} \cdot (1/\sqrt{d}) = \sqrt{d}$.

Near-orthogonality does not, however, make unrelated keys harmless when there are many of them. Although any single unrelated cosine has scale $1/\sqrt{d}$, the *largest* cosine among n independent keys grows, to leading order, like $\sqrt{2 \ln n / d}$. In a large retrieval index, that worst-case background match is several times larger than $1/\sqrt{d}$, so a relevant key must clear a much higher bar, which is what training arranges (1 puts concrete numbers on this effect).

33.1.4 Hyperplanes and Decision Boundaries

A linear classifier asks two geometric questions about an input $\mathbf{x}$: which side of a decision boundary does it fall on, and how far is it from that boundary? Throughout this section, we write the boundary as $\mathbf{w} \cdot \mathbf{x} + b = 0$.

33.1.4.1 Hyperplanes and Signed Distance

In the plane, this equation defines a straight line, and a quick example shows the roles of $\mathbf{w}$ and b. Take $\mathbf{w} = (2, 1)$ and $b = -1$. The points satisfying $2x + y - 1 = 0$ form the line $y = 1 - 2x$. Every point $\mathbf{x}$ on this line satisfies $\mathbf{w} \cdot \mathbf{x} = -b$, so its shadow onto $\mathbf{w}$ has the same signed length Equation 33.4:

$$\frac{\mathbf{w} \cdot \mathbf{x}}{\|\mathbf{w}\|} = \frac{-b}{\|\mathbf{w}\|} = \frac{1}{\sqrt{5}}.$$

Geometrically, the line runs perpendicular to $\mathbf{w}$ and crosses it at signed position $-b/\|\mathbf{w}\|$, at distance $|b|/\|\mathbf{w}\| = 1/\sqrt{5} \approx 0.45$ from the origin.

Nothing changes in higher dimensions. A *hyperplane* in $\mathbb{R}^d$ is a set $\{\mathbf{x} : \mathbf{w} \cdot \mathbf{x} + b = 0\}$ with $\mathbf{w} \neq \mathbf{0}$; we call $\mathbf{w}$ its *normal* vector and b its *offset*. It is simply a shifted copy of the subspace $\mathbf{w} \cdot \mathbf{x} = 0$, which has dimension $d - 1$ (Section 33.2.2.2): a line in $\mathbb{R}^2$, a plane in $\mathbb{R}^3$, and a 783-dimensional flat sheet inside image space. The hyperplane partitions $\mathbb{R}^d$ into two *half-spaces*, $\mathbf{w} \cdot \mathbf{x} + b > 0$ and $\mathbf{w} \cdot \mathbf{x} + b < 0$. Adjusting b translates the hyperplane parallel to itself along $\mathbf{w}$ without tilting it.

How far is an arbitrary point $\mathbf{x}$ from the hyperplane? Pick any point $\mathbf{x}_0$ lying on the hyperplane, so $\mathbf{w} \cdot \mathbf{x}_0 = -b$. The distance from $\mathbf{x}$ to the hyperplane is the signed length of the shadow of the displacement $\mathbf{x} - \mathbf{x}_0$ onto the normal $\mathbf{w}$, giving the *signed distance*

$$\frac{\mathbf{w} \cdot (\mathbf{x} - \mathbf{x}_0)}{\|\mathbf{w}\|} = \frac{\mathbf{w} \cdot \mathbf{x} + b}{\|\mathbf{w}\|}. \tag{33.11}$$

Notice that $\mathbf{x}_0$ drops out completely: the signed distance depends only on $\mathbf{x}$, $\mathbf{w}$, and b. It is positive on the side $\mathbf{w}$ points toward, negative on the opposite side, and zero on the hyperplane itself. In our planar example, the origin sits at signed distance $-1/\sqrt{5}$, while the point $(1, 1)$ has score $\mathbf{w} \cdot \mathbf{x} + b = 2$, placing it at signed distance $+2/\sqrt{5} \approx 0.89$ on the positive side; its closest point on the line is $(1, 1) - (2/5)(2, 1) = (0.2, 0.6)$. The point $(0, 1)$ scores 0 and lies directly on the line (Figure 33.7).

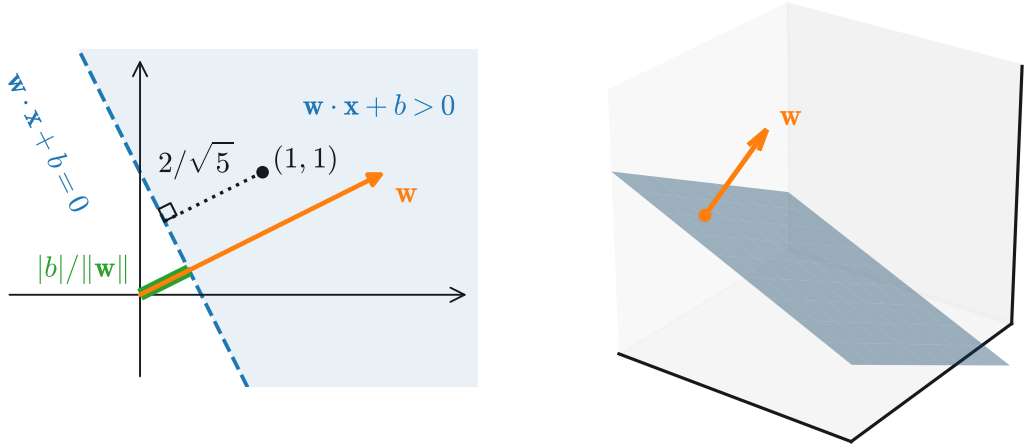


Figure 33.7: The hyperplane $\mathbf{w} \cdot \mathbf{x} + b = 0$ with normal $\mathbf{w}$. Left: for $\mathbf{w} = (2, 1)$ and $b = -1$, the hyperplane is the line $2x + y = 1$ at distance $1/\sqrt{5}$ from the origin; the point $(1, 1)$ lies at signed distance $2/\sqrt{5}$ on the side $\mathbf{w}$ points toward, and the shaded half-plane is $\mathbf{w} \cdot \mathbf{x} + b > 0$. Right: in three dimensions the same equation defines a plane.

For a labeled example $(\mathbf{x}, y)$ with $y \in \{-1, +1\}$, we also want to know whether it falls on the *correct* side. Multiplying the signed distance by the label defines the *margin*, $y\,(\mathbf{w} \cdot \mathbf{x} + b)/\|\mathbf{w}\|$, which is positive when the classification is right and grows larger the deeper the point sits inside its own half-space. Rescaling $(\mathbf{w}, b) \to (c\mathbf{w}, cb)$ for any $c > 0$ leaves both the hyperplane and the margin unchanged, while multiplying the raw score $\mathbf{w} \cdot \mathbf{x} + b$ by c. Any learning objective that maximizes the raw score must therefore constrain the scale of $\mathbf{w}$.

33.1.4.2 The Nearest-Mean Classifier

Which hyperplane should we pick to separate two classes? The simplest rule requires no iterative training at all: assign $\mathbf{x}$ to the class whose mean lies closer. Remarkably, this rule is itself a single hyperplane. Let $\bar{\mathbf{x}}_0$ and $\bar{\mathbf{x}}_1$ denote the two class means. Expanding both squared distances, the quadratic term $\|\mathbf{x}\|^2$ cancels on both sides:

$$\begin{aligned}\|\mathbf{x}-\bar{\mathbf{x}}_1\|^2 < \|\mathbf{x}-\bar{\mathbf{x}}_0\|^2 &\iff -2\,\mathbf{x}\cdot\bar{\mathbf{x}}_1 + \|\bar{\mathbf{x}}_1\|^2 < -2\,\mathbf{x}\cdot\bar{\mathbf{x}}_0 + \|\bar{\mathbf{x}}_0\|^2 \\ &\iff (\bar{\mathbf{x}}_1-\bar{\mathbf{x}}_0)\cdot\mathbf{x} - \tfrac{1}{2}(\|\bar{\mathbf{x}}_1\|^2 - \|\bar{\mathbf{x}}_0\|^2) > 0.\end{aligned}$$

Using the identity $\|\mathbf{a}\|^2 - \|\mathbf{c}\|^2 = (\mathbf{a}-\mathbf{c})\cdot(\mathbf{a}+\mathbf{c})$, the final inequality becomes $\mathbf{w}\cdot\mathbf{x}+b>0$ with

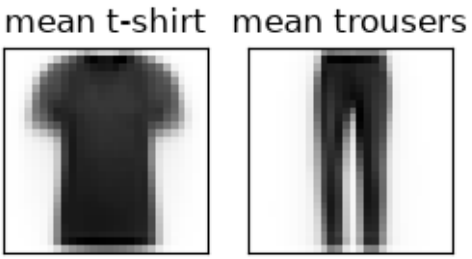
$$\mathbf{w} = \bar{\mathbf{x}}_1 - \bar{\mathbf{x}}_0, \qquad b = -\mathbf{w}\cdot\tfrac{1}{2}\left(\bar{\mathbf{x}}_0 + \bar{\mathbf{x}}_1\right).$$

So "nearest mean" is just the perpendicular bisector hyperplane: its normal points from one mean to the other, and it cuts directly through their midpoint. There is no threshold to tune by hand, and the rule is invariant to uniform pixel rescaling: multiplying every pixel by $c>0$ sends $\mathbf{w}\to c\mathbf{w}$ and $b\to c^2 b$, scaling every score by c^2 without changing a single prediction (2 extends this invariance to rotations and translations, and constructs a per-pixel rescaling that *does* change predictions).

Before testing this classifier on t-shirts and trousers, let us inspect the two class means and compute the angle between them.

```
X0, X1 = X_train[y_train == 0], X_train[y_train == 1]    # t-shirts, trousers
mean_0, mean_1 = X0.mean(axis=0), X1.mean(axis=0)
d2l.show_images([mean_0.reshape(28, 28), mean_1.reshape(28, 28)], 1, 2,
                titles=['mean t-shirt', 'mean trousers'], cmap='Greys')
np.degrees(angle(mean_0, mean_1))
```

```
np.float64(33.20917997376519)
```



Both mean images are blurry yet immediately recognizable. They sit only about 33° apart (a cosine of 0.84), far from orthogonal, because both garments occupy the bright center of the frame. Seeing the two means so close in angle might make us pessimistic, but the nearest-mean classifier does not require orthogonal means; it relies only on their *difference*. Let us evaluate it on all $2{,}000$ test images of t-shirts and trousers.

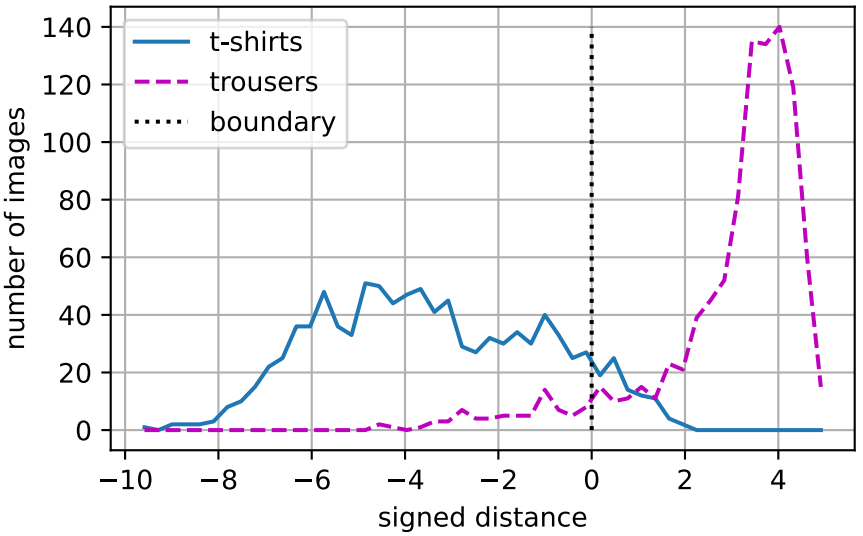
```
w = mean_1 - mean_0
b = -w @ (mean_0 + mean_1) / 2
mask = (y_test == 0) | (y_test == 1)
X2, y2 = X_test[mask], y_test[mask]
predictions = X2 @ w + b > 0                              # True: trousers
np.mean(predictions == (y2 == 1))
```

```
np.float64(0.9155)
```

The rule classifies about 92% of the test images correctly, putting only 169 of the $2{,}000$ on the wrong side. All we did was average each class and check which side of one hyperplane each test image falls on.

To understand where the remaining errors come from, we compress every test image to a single number, its signed distance Equation 33.11 to the boundary, and plot the histogram for each class.

```
dist = (X2 @ w + b) / np.linalg.norm(w)         # signed distance to boundary
bins = np.linspace(dist.min(), dist.max(), 51)
counts = [np.histogram(dist[y2 == c], bins)[0] for c in (0, 1)]
top = max(counts[0].max(), counts[1].max())
d2l.plot([bins[:-1]] * 2 + [[0, 0]], counts + [[0, top]], 'signed distance',
         'number of images', fmts=('-', 'm--', 'k:'),
         legend=['t-shirts', 'trousers', 'boundary'], figsize=(5, 3))
```



The two classes form distinct modes whose tails cross the boundary. T-shirts form a broad hump centered around -4, while trousers form a sharper peak near $+4$ with a long tail stretching toward zero. On average, each class sits near the signed distance of its training mean, $\mp\|\mathbf{w}\|/2 \approx \mp 3.2$. Let us quantify this picture by computing each class's standard deviation along $\mathbf{w}$ and counting how many examples land on the wrong side.

```
(dist[y2 == 0].std(), dist[y2 == 1].std(),
 int((dist[y2 == 0] > 0).sum()), int((dist[y2 == 1] < 0).sum()))
```

```
(np.float64(2.356225995048113), np.float64(1.6627004538100882), 101, 68)
```

Along $\mathbf{w}$, t-shirts have a standard deviation of about 2.4 and trousers about 1.7, with 101 t-shirts and 68 trousers misclassified. Because each class center lies less than two standard deviations from the boundary, an error rate of several percent is entirely expected. Notice the built-in limitation of the midpoint rule: it looks only at the two class means and has no way to account for the size or shape of either cloud. What happens when we apply it to

harder pairs that look much more alike, such as t-shirts versus shirts or pullovers versus coats?

```
# t-shirt/trouser, t-shirt/shirt, pullover/coat
for a, c in [(0, 1), (0, 6), (2, 4)]:
    m_a = X_train[y_train == a].mean(axis=0)
    m_c = X_train[y_train == c].mean(axis=0)
    w_ac, b_ac = m_c - m_a, -(m_c - m_a) @ (m_a + m_c) / 2
    pair = (y_test == a) | (y_test == c)
    acc = np.mean((X_test[pair] @ w_ac + b_ac > 0) == (y_test[pair] == c))
    dist_ac = (X_test[pair] @ w_ac + b_ac) / np.linalg.norm(w_ac)
    sd_a, sd_c = (dist_ac[y_test[pair] == k].std() for k in (a, c))
    names = data.text_labels([a, c])
    print(f'{names[0]:>8} vs {names[1]:<9}accuracy {acc:.3f}   '
          f'||w||/2 {np.linalg.norm(w_ac) / 2:.2f}   '
          f'std along w {sd_a:.2f}, {sd_c:.2f}')
```

```
 T-shirt vs trouser  accuracy 0.915   ||w||/2 3.23   std along w 2.36, 1.66
 T-shirt vs shirt    accuracy 0.786   ||w||/2 1.86   std along w 1.77, 2.38
pullover vs coat     accuracy 0.716   ||w||/2 1.24   std along w 1.88, 2.09
```

Accuracy drops sharply: t-shirts versus shirts falls to about 79%, and pullovers versus coats falls to about 72%. The remaining columns explain why. Each training mean sits at distance $\|\mathbf{w}\|/2$ from the boundary, and what governs overlap is that distance measured in units of each class's standard deviation along $\mathbf{w}$. For t-shirts and trousers, the gap is 1.4 to 1.9 standard deviations. For the two harder pairs, it shrinks to just 0.6 to 1.0 standard deviations, so a boundary placed at the midpoint slices straight through the bulk of both classes. The cure is to tilt $\mathbf{w}$ away from directions along which the classes vary widely, which is the idea behind Fisher's linear discriminant (4). Softmax regression goes a step further by learning both the tilt and the offset directly from data, while a deep network transforms the inputs themselves into a representation where the classes become nearly separable by a single hyperplane.

33.1.4.3 Layers as Stacked Hyperplanes

A linear neural network layer runs many hyperplane tests in parallel. Entry i of the matrix-vector product $\mathbf{Ax}$ is the dot product of row i of $\mathbf{A}$ with $\mathbf{x}$, a score $\mathbf{a}_i \cdot \mathbf{x}$ against the normal $\mathbf{a}_i$, and adding a bias vector turns each entry into a hyperplane score. For instance, the ten-class softmax layer of Section 3.1 on Fashion-MNIST stores ten normals $\mathbf{w}_1, \dots, \mathbf{w}_{10}$ and ten offsets. Because it predicts the class with the highest score, the decision boundary separating classes i and j is the hyperplane $(\mathbf{w}_i - \mathbf{w}_j) \cdot \mathbf{x} + (b_i - b_j) = 0$ where their two scores tie.

To see this concretely, take our chapter's running matrix $\mathbf{A} = \left(\begin{smallmatrix} 1 & 2 \\ -1 & 3 \end{smallmatrix}\right)$ and the input $\mathbf{x} = (2, -1)$. Row one yields $1 \cdot 2 + 2 \cdot (-1) = 0$, meaning $\mathbf{x}$ lies directly on the hyperplane through the origin with normal $(1, 2)$. Row two yields $-2 - 3 = -5$, placing $\mathbf{x}$ on the negative side of the hyperplane with normal $(-1, 3)$ at signed distance $-5/\sqrt{10} \approx -1.58$. Stacking both results gives $\mathbf{Ax} = (0, -5)$.

Viewing $\mathbf{Ax}$ row by row reveals a linear layer as a stack of hyperplane tests, yet it leaves two essential questions unanswered: which inputs does the transformation collapse to zero, and when can its action be inverted? Reading the very same product column by column answers

both questions in Section 33.2.

33.1.5 Summary

The dot product extends lengths, angles, and projections from the plane to any dimension using the exact same formulas. Projecting onto a line or subspace produces the unique closest point, characterized by a residual orthogonal to the target space; from this single geometric fact follow the Cauchy–Schwarz inequality (which makes high-dimensional angles well defined) and the normal equations of least squares. The Gram–Schmidt process converts any spanning list into an orthonormal basis, yielding the $\mathbf{Q}$ factor of the QR factorization. While random directions in $\mathbb{R}^d$ are nearly orthogonal with typical cosines of order $1/\sqrt{d}$, real image datasets are far from isotropic, so an observed cosine is meaningful only relative to unrelated pairs from the same distribution. Finally, the nearest-mean classifier defines a single hyperplane bisecting the two class means: it reaches about 92% accuracy on t-shirts versus trousers, but drops to 72%–79% on visually similar pairs where class spread along the boundary normal matters.

33.1.6 Exercises

1. code **Cosines between random vectors.**
    1. Sample $1{,}000$ pairs of independent random unit vectors in $d = 10{,}000$ dimensions by normalizing standard Gaussian vectors, and compute the cosine of each pair. Compare the empirical standard deviation with the prediction $1/\sqrt{d} = 0.01$ from the near-orthogonality proposition, and report the largest $|\cos\theta|$ observed.
    2. A retrieval system compares one query against n keys and returns the key with the largest cosine. Model the query and keys as independent random unit vectors in $d = 256$ dimensions. Compute the maximum of the n cosines for $n = 10$, 10^3, and 10^5, and compare it with $\sqrt{2\ln n/d}$, the leading-order estimate for the maximum of n independent Gaussian draws of variance $1/d$. How large a cosine does an unrelated key achieve as the top match among 10^5 keys? How should a threshold for declaring two embeddings related scale with n and d?
2. **Nearest means and invariance.** The nearest-mean rule assigns $\mathbf{x}$ to class 1 when $\mathbf{w}\cdot\mathbf{x}+b>0$, where $\mathbf{w}=\bar{\mathbf{x}}_1-\bar{\mathbf{x}}_0$ and $b=-\mathbf{w}\cdot\frac{1}{2}(\bar{\mathbf{x}}_0+\bar{\mathbf{x}}_1)$.
    1. Suppose every training and test image undergoes the same transformation $\mathbf{x}\mapsto c\mathbf{Q}\mathbf{x}+\mathbf{z}$, where $c>0$, $\mathbf{Q}$ is square with $\mathbf{Q}^\top\mathbf{Q}=\mathbf{I}$, and $\mathbf{z}$ is a fixed vector, and we recompute the class means and b. Show that no prediction changes, and connect this to the rescaling $\mathbf{w}\to c\mathbf{w}$, $b\to c^2 b$ discussed in the text.
    2. Standardizing each pixel to unit variance replaces $\mathbf{x}$ by $\mathbf{D}\mathbf{x}$ for a positive diagonal matrix $\mathbf{D}$. Construct two class means and a test point in $\mathbb{R}^2$ whose predicted class flips under such a $\mathbf{D}$.
    3. With q classes under the nearest-mean rule, show that the region assigned to class i is the intersection of the $q-1$ half-spaces $(\bar{\mathbf{x}}_i-\bar{\mathbf{x}}_j)\cdot\mathbf{x}+b_{ij}>0$ for $j\neq i$, and find b_{ij}. Compare this with the pairwise boundaries $(\mathbf{w}_i-\mathbf{w}_j)\cdot\mathbf{x}+(b_i-b_j)=0$ of a softmax layer: which choice of weights and biases makes a softmax layer reproduce the nearest-mean classifier?
3. **Orthogonal and oblique projectors.**

   1. Let $\mathbf{P}$ satisfy $\mathbf{P}^2 = \mathbf{P}$ and $\mathbf{P}^\top = \mathbf{P}$. Show that for every $\mathbf{x}$, $\mathbf{P}\mathbf{x}$ lies in the column space of $\mathbf{P}$ and $\mathbf{x} - \mathbf{P}\mathbf{x}$ is orthogonal to it. Deduce via Pythagoras' theorem that $\mathbf{P}\mathbf{x}$ is the closest point in the column space, without invoking an orthonormal basis.
   2. The matrix $\mathbf{P}_{\mathrm{o}} = \left(\begin{smallmatrix} 1 & 1 \\ 0 & 0 \end{smallmatrix}\right)$ satisfies $\mathbf{P}_{\mathrm{o}}^2 = \mathbf{P}_{\mathrm{o}}$ but is not symmetric. For $\mathbf{x} = (1, 1)$, compute $\mathbf{P}_{\mathrm{o}}\mathbf{x}$ and the orthogonal projection of $\mathbf{x}$ onto the same column space (the first coordinate axis), along with the distance of each from $\mathbf{x}$. Along what direction does $\mathbf{P}_{\mathrm{o}}$ project points?
   3. Show that any matrix $\mathbf{P}$ with $\mathbf{P}^2 = \mathbf{P}$ decomposes every $\mathbf{x}$ as $\mathbf{P}\mathbf{x} + (\mathbf{x} - \mathbf{P}\mathbf{x})$, with the first term in the column space of $\mathbf{P}$ and the second in its null space $\{\mathbf{z} : \mathbf{P}\mathbf{z} = \mathbf{0}\}$. Show that these two subspaces are orthogonal if and only if $\mathbf{P}^\top = \mathbf{P}$.

4. code **Tilting the boundary.** ● The mean-difference normal $\Delta = \bar{\mathbf{x}}_1 - \bar{\mathbf{x}}_0$ ignores how the images in each class spread around their mean. Let $\mathbf{C}$ be the average of the two within-class covariance matrices on the training set. Fisher's linear discriminant (Fisher 1936) selects the normal $\mathbf{w}$ maximizing

$$J(\mathbf{w}) = \frac{(\mathbf{w} \cdot \Delta)^2}{\mathbf{w}^\top \mathbf{C} \mathbf{w}},$$

   the squared separation between the projected class means divided by the within-class variance of the projections. When $\mathbf{C}$ is positive definite ($\mathbf{v}^\top \mathbf{C} \mathbf{v} > 0$ for all $\mathbf{v} \neq \mathbf{0}$), its inverse $\mathbf{C}^{-1}$ exists and the maximizer is $\mathbf{w} \propto \mathbf{C}^{-1}\Delta$. (This problem uses the matrix inverse from Section 33.2.2.3, positive definiteness from Section 33.3.2.2, and the condition number from Section 33.5.3.2.)

   1. Show that $J(c\mathbf{w}) = J(\mathbf{w})$ for every $c \neq 0$. When $\mathbf{C} = s^2\mathbf{I}$, apply Cauchy–Schwarz to show that Δ maximizes J.
   2. Using the arrays `X0` and `X1` of t-shirts and trousers from the text, compute $\mathbf{C}$ and let $\bar{s}^2 = \operatorname{tr}(\mathbf{C})/784$ be the average pixel variance. For these images, $\mathbf{C}$ has a condition number of about 10^9 because some border pixels are nearly constant. For $\gamma \in \{10^{-4}, 10^{-2}, 1, 10^2, 10^4\}$, form the regularized normal $\mathbf{w}_\gamma = (\mathbf{C} + \gamma \bar{s}^2 \mathbf{I})^{-1}\Delta$, place the boundary at the midpoint of the two means, and evaluate both training and test accuracy. Next, hold out $1{,}000$ training images per class, recompute $\mathbf{C}$ and the class means on the remaining images, select γ by validation accuracy on the held-out set, and replot the signed-distance histogram using that $\mathbf{w}_\gamma$.
   3. Why does $\mathbf{w}_\gamma$ align with Δ as $\gamma \to \infty$? Why scale the regularizer by $\bar{s}^2$ instead of adding a fixed multiple of $\mathbf{I}$?

5. code **Gram–Schmidt in floating point.** ● The Gram–Schmidt process can be executed in two algebraically equivalent orders. Classical Gram–Schmidt (CGS) computes $\mathbf{q}_j$ by normalizing $\mathbf{a}_j - \sum_{i<j} (\mathbf{q}_i \cdot \mathbf{a}_j)\,\mathbf{q}_i$, as in our two-vector example. Modified Gram–Schmidt (MGS) initializes $\mathbf{v} = \mathbf{a}_j$, updates $\mathbf{v} \leftarrow \mathbf{v} - (\mathbf{q}_i \cdot \mathbf{v})\,\mathbf{q}_i$ sequentially for $i = 1, \dots, j-1$, and then normalizes $\mathbf{v}$.

   1. Show that CGS and MGS produce identical vectors in exact arithmetic.
   2. Implement both algorithms and run them on the Hilbert matrices $\mathbf{H}_n$ with entries $1/(i + j - 1)$ for $n = 4, 6, 8, 10, 12$, whose columns are nearly linearly dependent. Plot the Frobenius norm $\|\mathbf{Q}^\top \mathbf{Q} - \mathbf{I}\|_F$ against n for CGS, MGS, and `np.linalg.qr`. For each method, at what n does $\mathbf{P} = \mathbf{Q}\mathbf{Q}^\top$ fail to act as a projector, as measured by $\|\mathbf{P}^2 - \mathbf{P}\|_F$?

3. Apply the CGS subtraction step twice to each column, running the second pass on the vector left by the first. How close to orthonormal are the resulting columns compared with a single pass of CGS and with MGS? Explain the improvement by comparing the size of the components along $\mathbf{q}_1, \dots, \mathbf{q}_{j-1}$ entering the first pass versus those entering the second.

6. [extended] **Packing nearly orthogonal directions.** ●● Although $\mathbb{R}^d$ accommodates at most d mutually orthogonal directions, it can pack vastly more directions whose pairwise cosines stay small. Elhage et al. (2022) use this geometry to explain how a layer of width d can represent more than d features in *superposition* as nearly orthogonal directions. Their features are *sparse*: each activates on only a small fraction of inputs, so only a few need to be read out at once. Let $\mathbf{u}_1, \dots, \mathbf{u}_n$ be unit vectors in $\mathbb{R}^d$, stacked as the rows of $\mathbf{U} \in \mathbb{R}^{n \times d}$, and let $\mu = \max_{i \neq j} |\mathbf{u}_i \cdot \mathbf{u}_j|$ denote their worst pairwise interference.

   1. Expand $\sum_{i,j} (\mathbf{u}_i \cdot \mathbf{u}_j)^2$ in coordinates and show that it equals $\sum_{k,l} \big(\sum_i u_{ik} u_{il}\big)^2$, the sum of squared entries of the $d \times d$ matrix $\mathbf{U}^\top \mathbf{U}$. Bound this sum below using its d diagonal terms together with Cauchy–Schwarz in the form $\sum_{k=1}^{d} a_k^2 \geq \big(\sum_k a_k\big)^2 / d$, noting that $\sum_k (\mathbf{U}^\top \mathbf{U})_{kk} = n$. Bound the original sum above by $n + n(n-1)\mu^2$ to derive the *Welch bound* (Welch 1974):

$$\mu^2 \geq \frac{n - d}{d\,(n-1)}.$$

      Evaluate this bound for $d = 32$ with $n = 64$ and $n = 512$.

   2. A union bound over all pairs shows that independent random directions keep $\mu \leq \varepsilon$ for n on the order of $e^{d\varepsilon^2/4}$ (Section 36.9). For $d = 32$ and $n \in \{64, 128, 256, 512\}$, compute μ for random unit vectors and compare it against the Welch bound.

   3. Reduce μ directly by optimization: minimize $\sum_{i \neq j} (\mathbf{u}_i \cdot \mathbf{u}_j)^p$ for an even power such as $p = 10$ via gradient steps on $\mathbf{U}$, renormalizing every row after each step. For $d \in \{16, 32, 48, 64\}$ and $n \in \{d, 2d, 4d, 8d, 16d\}$, plot μ against n for random vectors, optimized vectors, and the Welch bound.

   4. Use these curves to estimate, for each d, the largest n for which optimized directions keep $\mu \leq 0.2$. How does this capacity grow with d compared with the random-direction rate $e^{d\varepsilon^2/4}$? How would your conclusions change if some features were more important than others, or if many features activated simultaneously?

33.2 Matrices as Linear Maps

In Section 33.1 we read a matrix–vector product $\mathbf{Ax}$ one row at a time: entry i is the dot product of row i with $\mathbf{x}$, a single hyperplane test. That row view tells us what a matrix computes for one input vector, but it hides what the matrix does to the entire input space at once. In particular, it does not tell us whether the map preserves information or throws it away, and whether its effect can ever be undone.

This question comes up as soon as a neural network layer is narrower than its input. A fully connected layer mapping the 784 pixels of a Fashion-MNIST image to 256 hidden features is a 256×784 matrix. Because its output holds fewer numbers than its input, it must collapse space: by a count we will make precise shortly, at least $784 - 256 = 528$ independent input directions are sent to zero, so distinct images can produce identical features before any nonlinearity even sees them.

Even a square matrix can lose information. Consider the running matrix of this chapter, $\mathbf{A} = \left(\begin{smallmatrix} 1 & 2 \\ -1 & 3 \end{smallmatrix}\right)$. It sends $(2, -1)$ to $(0, -5)$, and there is another matrix that sends $(0, -5)$ right back to $(2, -1)$. In fact, that second matrix undoes $\mathbf{A}$ on every input, so $\mathbf{A}$ loses nothing. Contrast this with $\mathbf{B} = \left(\begin{smallmatrix} 2 & 1 \\ 4 & 2 \end{smallmatrix}\right)$, which sends both $(1, 1)$ and $(0, 3)$ to $(3, 6)$. Given only the output $(3, 6)$, we have no way to tell which input produced it, so no matrix can reverse $\mathbf{B}$.

Whether a linear map can be undone determines whether a layer discards information, whether a linear system has a unique solution, and how a probability density rescales under a change of variables (Section 36.2.3.1). To answer this question cleanly, we shift from rows to columns. Column j of a matrix is simply where it sends the j-th coordinate axis, and those images alone fix the entire map. Counting how many columns are linearly independent tells us how many input dimensions survive the map, while measuring the signed volume of the transformed unit cube gives a single number that detects whether any dimension was lost. We develop both ideas first in two dimensions, where every statement is a picture, and then extend them to n dimensions.

Three results organize the section: the balance between surviving and collapsed dimensions, the exact conditions under which a square matrix has an inverse, and the rule for how volume scaling behaves when two maps are composed. The general n-dimensional construction of the determinant (Section 33.2.3.2) is the most technical part and can be skimmed on a first reading.

```
import numpy as np
```

33.2.1 Linear Maps

33.2.1.1 Columns as Images of the Axes

A matrix acts on infinitely many input vectors, yet it is stored as a finite table of numbers. How can just four numbers fix the output for every vector in the plane? Take a 2×2 matrix $\mathbf{A}$ with entries p, q, r, s and apply it to $\mathbf{x} = (x, y)$. Collecting the terms that multiply x and those that multiply y gives

$$\mathbf{A}\mathbf{x} = \begin{bmatrix} p & q \\ r & s \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} px + qy \\ rx + sy \end{bmatrix} = x \begin{bmatrix} p \\ r \end{bmatrix} + y \begin{bmatrix} q \\ s \end{bmatrix} = x\,\mathbf{A}\mathbf{e}_1 + y\,\mathbf{A}\mathbf{e}_2.$$

Thus $\mathbf{A}\mathbf{x}$ is simply a weighted sum of the two columns of $\mathbf{A}$, using the coordinates of $\mathbf{x}$ as weights. Because $\mathbf{A}\mathbf{e}_1 = (p, r)$ and $\mathbf{A}\mathbf{e}_2 = (q, s)$, the columns are the images of the standard basis vectors $\mathbf{e}_1 = (1, 0)$ and $\mathbf{e}_2 = (0, 1)$. Once we know where those two axes land, the image of every other vector in the plane is completely determined.

For our running matrix ($p = 1$, $q = 2$, $r = -1$, $s = 3$), the two axes go to $\mathbf{A}\mathbf{e}_1 = (1, -1)$ and $\mathbf{A}\mathbf{e}_2 = (2, 3)$. Since the input $\mathbf{v} = (2, -1)$ is $2\mathbf{e}_1 - \mathbf{e}_2$, its image is the exact same combination of the columns:

$$\mathbf{Av} = 2\,(1,-1) - (2,3) = (0,-5).$$

Checking by rows confirms the result: $(1,2)\cdot(2,-1) = 0$ and $(-1,3)\cdot(2,-1) = -5$. Both views compute the same product, but the column view lets us see what $\mathbf{A}$ does to the entire plane at once. Figure 33.8 illustrates this geometry: the unit square spanned by $\mathbf{e}_1$ and $\mathbf{e}_2$ is carried to the parallelogram spanned by the two columns, and the grid point $\mathbf{v}$ lands on the corresponding point $\mathbf{Av}$ of the transformed grid.

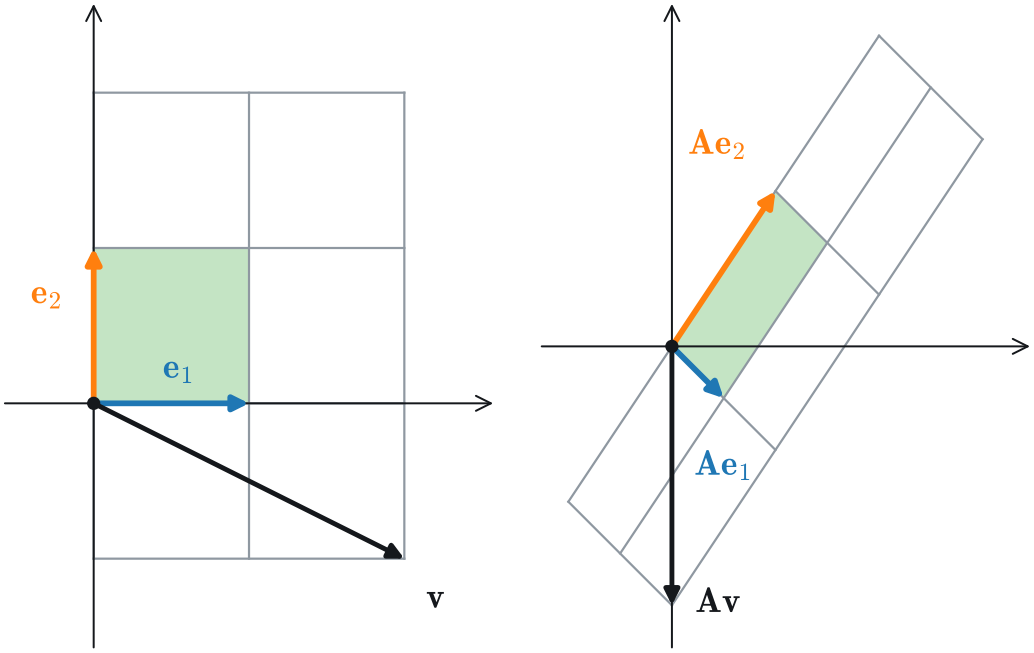


Figure 33.8: The running matrix $\mathbf{A} = \left(\begin{smallmatrix} 1 & 2 \\ -1 & 3 \end{smallmatrix}\right)$ acting on the plane. (a) The input grid, shaded unit square, basis vectors, and input $\mathbf{v} = (2,-1)$. (b) Its image: $\mathbf{e}_1$ goes to $(1,-1)$, $\mathbf{e}_2$ to $(2,3)$, and $\mathbf{v}$ to $(0,-5)$. Lines remain lines, the origin stays fixed, and equally spaced points stay equally spaced. The shaded square maps to the shaded parallelogram, whose area leads to the determinant. Panel (b) is drawn at one-third the scale of (a).

The same column picture holds in any dimension. An $m \times n$ matrix maps vectors in $\mathbb{R}^n$ to vectors in $\mathbb{R}^m$, and it is *linear*:

$$\mathbf{A}(\alpha\mathbf{x} + \beta\mathbf{y}) = \alpha\,\mathbf{Ax} + \beta\,\mathbf{Ay} \quad \text{for all } \mathbf{x},\mathbf{y} \in \mathbb{R}^n \text{ and } \alpha,\beta \in \mathbb{R}.$$

In words, the image of a weighted sum is the same weighted sum of the images. Writing any input as $\mathbf{x} = \sum_j x_j \mathbf{e}_j$ gives $\mathbf{Ax} = \sum_j x_j\,\mathbf{Ae}_j$, where multiplying by $\mathbf{e}_j$ picks out column j. Conversely, every map $f : \mathbb{R}^n \to \mathbb{R}^m$ that satisfies this linearity rule can be written as a matrix whose j-th column is $f(\mathbf{e}_j)$, because $f(\mathbf{x}) = f(\sum_j x_j \mathbf{e}_j) = \sum_j x_j f(\mathbf{e}_j)$. Linear maps from $\mathbb{R}^n$ to $\mathbb{R}^m$ and $m \times n$ matrices are therefore two ways of describing the same object.

Linearity is also what keeps the grid in Figure 33.8 looking like a grid. Every line $\mathbf{p} + t\mathbf{u}$ maps to $\mathbf{Ap} + t\,\mathbf{Au}$, which is again a line (or a single point if $\mathbf{Au} = \mathbf{0}$), and the origin never moves: $\mathbf{A0} = \mathbf{0}$.

Nothing here requires the matrix to be square. A 2×3 matrix maps $\mathbb{R}^3$ into the span of its three columns in $\mathbb{R}^2$, filling the whole output plane as long as at least two columns are linearly independent. Going the other way, a 3×2 matrix maps the plane into $\mathbb{R}^3$, where its outputs form a plane through the origin, a line, or the origin alone, depending on its two columns. Every fully connected layer in a neural network is a rectangular map of this kind, followed by a bias shift and a nonlinearity.

To build visual intuition, Figure 33.9 displays six 2×2 matrices through their effect on the unit square. A 90° counter-clockwise rotation sends $\mathbf{e}_1$ to $\mathbf{e}_2$ and $\mathbf{e}_2$ to $-\mathbf{e}_1$, so its columns are $(0, 1)$ and $(-1, 0)$. A horizontal shear leaves $\mathbf{e}_1$ untouched and tilts $\mathbf{e}_2$ to $(1, 1)$, sliding every horizontal line sideways in proportion to its height. The last two panels are qualitatively different: they lose a dimension, flattening the entire square, and with it the entire plane, onto a single line.

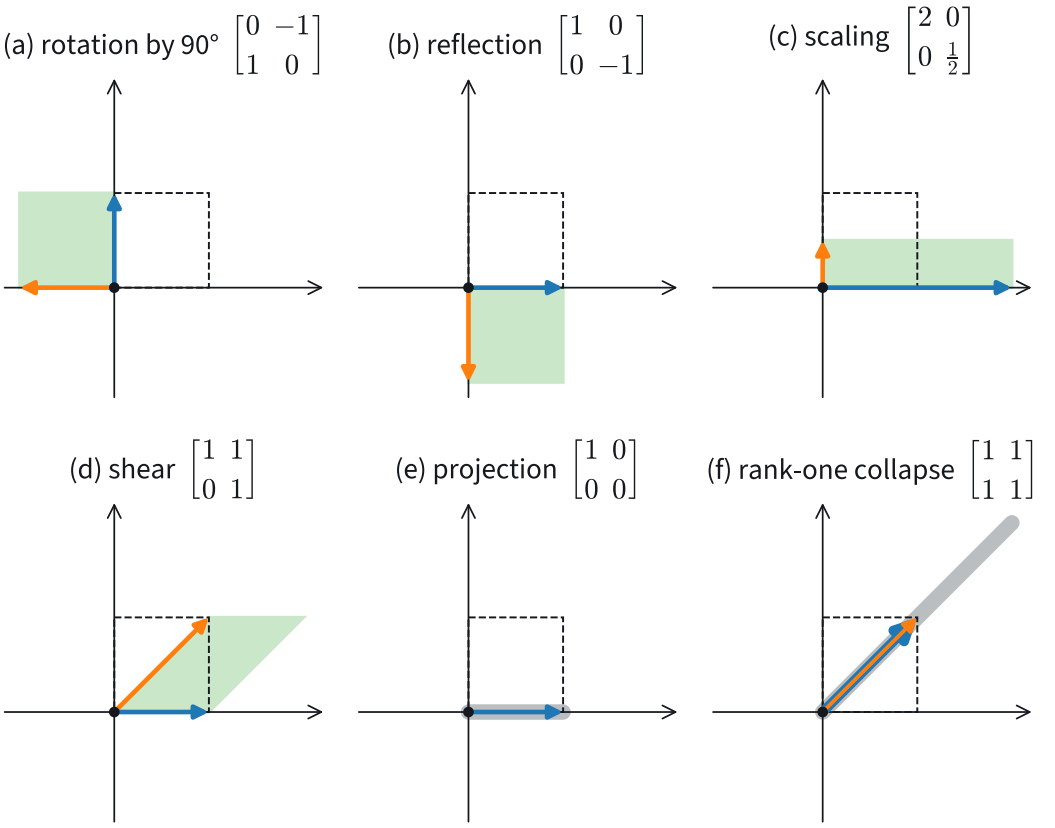


Figure 33.9: Six ways a 2×2 matrix can transform the unit square (dashed). In each panel, the arrows show the columns, which are the images of $\mathbf{e}_1$ (blue) and $\mathbf{e}_2$ (orange), and the shaded region is the image of the square. Panels (e) and (f) collapse the plane onto a line; in (f) both columns are $(1, 1)$.

One familiar operation is noticeably absent from this gallery: a translation $\mathbf{x} \mapsto \mathbf{x} + \mathbf{c}$ with

$\mathbf{c} \neq \mathbf{0}$. Because it moves the origin, it is not linear. An affine layer $\mathbf{Wx} + \mathbf{b}$ in a neural network is therefore a linear map followed by a translation.

33.2.1.2 Composition

Two linear layers placed back to back, with no nonlinearity between them, collapse into a single linear map. Which matrix represents that combined map? Let us test this on two concrete maps from Figure 33.9: the 90° rotation $\mathbf{R}_{90}$ and the shear $\mathbf{T}$ (named for how it tilts the vertical axis),

$$\mathbf{R}_{90} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}, \qquad \mathbf{T} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}.$$

Both matrices will reappear throughout the chapter. Let us track where the standard basis vectors go under each ordering. If we shear first, $\mathbf{e}_1$ stays at $(1, 0)$ and the rotation sends it to $(0, 1)$; meanwhile $\mathbf{e}_2$ shears to $(1, 1)$ and rotates to $(-1, 1)$. Thus "shear, then rotate" has columns $(0, 1)$ and $(-1, 1)$. In the opposite order, rotating sends $\mathbf{e}_1$ to $(0, 1)$, which the shear moves to $(1, 1)$, and sends $\mathbf{e}_2$ to $(-1, 0)$, which the shear leaves unchanged. The two composites are therefore

$$\text{shear, then rotate: } \begin{bmatrix} 0 & -1 \\ 1 & 1 \end{bmatrix}, \qquad \text{rotate, then shear: } \begin{bmatrix} 1 & -1 \\ 1 & 0 \end{bmatrix}.$$

They are not the same map, as Figure 33.10 makes clear.

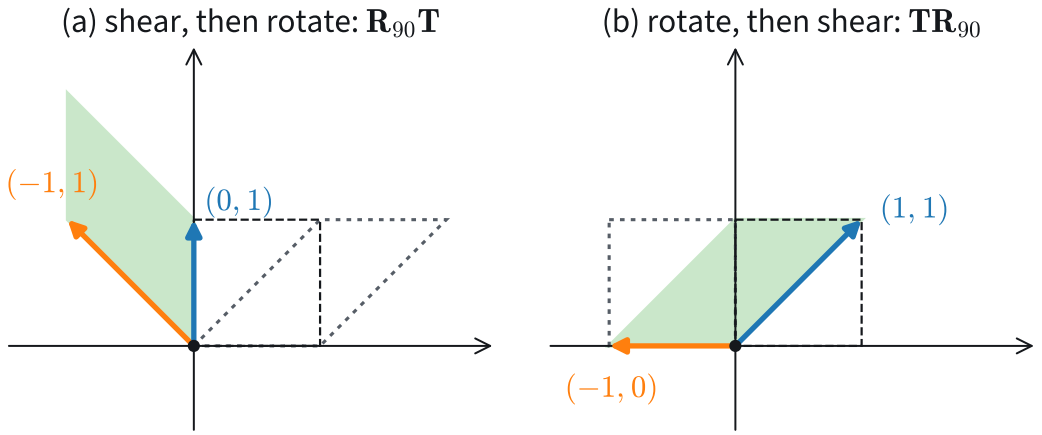


Figure 33.10: The order of composition matters. (a) Applying the shear $\mathbf{T}$ (dotted outline) followed by the rotation $\mathbf{R}_{90}$ produces $\mathbf{R}_{90}\mathbf{T}$. (b) Applying the rotation (dotted outline) followed by the shear produces $\mathbf{T}\mathbf{R}_{90}$. The labels show the final images of $\mathbf{e}_1$ (blue) and $\mathbf{e}_2$ (orange). Matrix products are always read from right to left.

Matrix multiplication is the algebraic rule that computes such composites. For any two matrices $\mathbf{A}$ and $\mathbf{B}$ whose dimensions match, apply $\mathbf{B}$ to $\mathbf{v}$, then $\mathbf{A}$ to the result, and inspect entry i:

$$\big(\mathbf{A}(\mathbf{B}\mathbf{v})\big)_i = \sum_j a_{ij}(\mathbf{B}\mathbf{v})_j = \sum_j \sum_k a_{ij} b_{jk} v_k = \sum_k \Big(\sum_j a_{ij} b_{jk}\Big) v_k = \big((\mathbf{A}\mathbf{B})\mathbf{v}\big)_i. \quad (33.12)$$

Applying the two maps in sequence is identical to applying the single matrix $\mathbf{AB}$, whose (i, k) entry is the dot product of row i of $\mathbf{A}$ with column k of $\mathbf{B}$. Notice the order: the product $\mathbf{AB}$ means "apply $\mathbf{B}$ first, then $\mathbf{A}$," read from right to left just like function composition $f(g(x))$. Setting $\mathbf{v} = \mathbf{e}_j$ in Equation 33.12 also gives a useful column rule: column j of $\mathbf{AB}$ is $\mathbf{A}\mathbf{b}_j$, the image under $\mathbf{A}$ of column j of $\mathbf{B}$. We will lean on this fact when we study the determinant.

Seeing multiplication as composition explains its seemingly odd definition. The row-times-column formula is not an arbitrary convention; it is simply what composing two linear maps produces. It also makes the algebraic properties immediate. Matrix multiplication is associative, $(\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC})$, because function composition is associative: both sides apply $\mathbf{C}$, then $\mathbf{B}$, then $\mathbf{A}$. And it is not commutative in general, because performing two geometric operations in opposite orders usually leads to different results. In NumPy, matrix multiplication uses the `@` operator. Let us check that it reproduces both composites of the rotation and the shear (stored as `R90` and `S` below).

```
R90 = np.array([[0.0, -1.0], [1.0, 0.0]])  # rotation by 90 degrees
S = np.array([[1.0, 1.0], [0.0, 1.0]])     # shear
print(R90 @ S)
print(S @ R90)
```

```
[[ 0. -1.]
 [ 1.  1.]]
[[ 1. -1.]
 [ 1.  0.]]
```

Of course, some pairs of matrices do commute. Two rotations in the plane commute, since rotating by ϕ and then by ψ yields a rotation by $\phi + \psi$ in either order. Two diagonal matrices commute as well, since each scales the coordinate axes independently. Exercise 3 characterizes every 2×2 matrix that commutes with $\mathbf{R}_{90}$.

33.2.1.3 Orthogonal Matrices

Some linear maps move space rigidly, leaving every length and every angle untouched. Which matrices have this property? Consider a counter-clockwise rotation by an angle ϕ in the plane. It sends $\mathbf{e}_1$ to $(\cos\phi, \sin\phi)$ and $\mathbf{e}_2$ to $(-\sin\phi, \cos\phi)$, so its matrix is

$$\mathbf{R}_\phi = \begin{bmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{bmatrix}.$$

Its columns have unit length, $\sqrt{\cos^2\phi + \sin^2\phi} = 1$, and zero dot product, $-\cos\phi\sin\phi + \sin\phi\cos\phi = 0$. They are orthonormal, a condition we wrote in Section 33.1 as $\mathbf{R}_\phi^\top \mathbf{R}_\phi = \mathbf{I}$. Setting $\phi = 90^\circ$ recovers $\mathbf{R}_{90}$, which sends $(3, 2)$ to $(-2, 3)$; both vectors have length $\sqrt{13}$. The reflection $\mathrm{diag}(1, -1)$ across the x-axis has orthonormal columns as well.

In any dimension, a square matrix $\mathbf{Q}$ is called *orthogonal* when its columns are orthonormal, that is,

$$\mathbf{Q}^\top \mathbf{Q} = \mathbf{I}.$$

Two subtleties in this definition are worth flagging. First, columns that are merely perpendicular are not enough: $\mathrm{diag}(1, 10)$ has perpendicular columns of lengths 1 and 10, yet it bends the 90° angle between $(1, 1)$ and $(1, -1)$ into $\arccos(-99/101) \approx 169^\circ$ (1). Second, the name "orthogonal" is reserved for square matrices. A tall $n \times k$ matrix with $k < n$ orthonormal columns still preserves lengths, but it embeds $\mathbb{R}^k$ into a larger space $\mathbb{R}^n$ and therefore has no two-sided inverse (Section 33.2.2.3).

An orthogonal map preserves every dot product:

$$(\mathbf{Q}\mathbf{x}) \cdot (\mathbf{Q}\mathbf{y}) = (\mathbf{Q}\mathbf{x})^\top (\mathbf{Q}\mathbf{y}) = \mathbf{x}^\top \mathbf{Q}^\top \mathbf{Q}\, \mathbf{y} = \mathbf{x}^\top \mathbf{y} = \mathbf{x} \cdot \mathbf{y}.$$

Setting $\mathbf{y} = \mathbf{x}$ gives $\|\mathbf{Q}\mathbf{x}\| = \|\mathbf{x}\|$, so lengths are unchanged; and because angles depend only on dot products and lengths, angles are unchanged as well. If you rotate an entire set of embedding vectors by the same orthogonal matrix, every pairwise cosine similarity stays identical.

The defining equation $\mathbf{Q}^\top \mathbf{Q} = \mathbf{I}$ says that applying $\mathbf{Q}^\top$ *after* $\mathbf{Q}$ undoes it. For a square matrix $\mathbf{Q}$, it turns out that $\mathbf{Q}^\top$ also undoes $\mathbf{Q}$ when applied *before* it, $\mathbf{Q}\mathbf{Q}^\top = \mathbf{I}$, meaning the rows of $\mathbf{Q}$ are orthonormal too and $\mathbf{Q}^{-1} = \mathbf{Q}^\top$. This is not obvious from the definition alone; it follows from a general theorem on square matrices that we prove in Section 33.2.2.3.

In two dimensions, rotations and reflections are the only orthogonal matrices. The first column must be a unit vector $(\cos\phi, \sin\phi)$, which leaves only two unit vectors perpendicular to it for the second column: $\pm(-\sin\phi, \cos\phi)$. The plus sign gives the rotation $\mathbf{R}_\phi$. The minus sign gives $\mathbf{R}_\phi\, \mathrm{diag}(1, -1)$, a reflection across the x-axis followed by a rotation, which is itself a reflection across a line through the origin. Later, the sign of the determinant will separate rotations from reflections at a glance (Section 33.2.3.3).

33.2.1.4 Tensors and Einstein Summation

In a transformer, self-attention compares every query with every key, across every attention head and every sequence in a batch. Labeling the batch by b, the head by h, the query position by i, the key position by t, and the feature dimension within a head by j, the attention scores are

$$s_{bhit} = \sum_j q_{bhij}\, k_{bhtj},$$

where q_{bhij} and k_{bhtj} are entries of the four-axis query and key tensors (Section 1.1). Each score is a dot product along the feature axis j, evaluated for every combination of the other four indices. Standard matrix notation has no clean way to write this four-index contraction; calling it "a batched matrix multiplication with a transpose on the last two axes" obscures which axis is actually being summed.

Step back, and notice that every product we have met so far follows the exact same pattern. The dot product, the matrix–vector product, the matrix product, and the *trace* $\mathrm{tr}(\mathbf{A})$ (the sum of the diagonal entries of a square matrix) all read

$$\mathbf{v}\cdot\mathbf{w}=\sum_i v_i w_i, \qquad (\mathbf{Av})_i=\sum_j a_{ij}v_j, \qquad (\mathbf{AB})_{ik}=\sum_j a_{ij}b_{jk}, \qquad \operatorname{tr}(\mathbf{A})=\sum_i a_{ii}.$$

The trace is new here and will return in Section 33.3 as the sum of a matrix's eigenvalues. Each of these four formulas multiplies entries together and sums over an index that appears twice. *Einstein notation* turns that pattern into a convention: drop the summation sign and automatically sum over every repeated index. Under this rule, $\mathbf{v}\cdot\mathbf{w}=v_i w_i$ and $(\mathbf{AB})_{ik}=a_{ij}b_{jk}$.

The attention scores reveal where the classical convention falls short: the batch and head indices b and h appear in both factors, yet we want to keep them rather than sum them out. Numerical libraries solve this in `einsum` by having you name the kept indices explicitly. You pass a string listing the input indices, an arrow `->`, and the output indices; any index omitted from the right-hand side is summed over. Let us express our four basic products this way and check them on our running matrices.

```
A = np.array([[1.0, 2.0], [-1.0, 3.0]])    # the running matrix
v = np.array([2.0, -1.0])
(np.einsum('i,i->', v, v),         # dot product: v.v
 np.einsum('ij,j->i', A, v),       # matrix-vector product: Av
 np.einsum('ij,jk->ik', R90, S),   # matrix product: R90 S
 np.einsum('ii->', A))             # trace: tr(A)
```

```
(np.float64(5.0),
 array([ 0., -5.]),
 array([[ 0., -1.],
        [ 1.,  1.]]),
 np.float64(4.0))
```

Every output matches our hand calculations from earlier, including $\operatorname{tr}\mathbf{A}=1+3=4$. The attention scores take just one line in the same notation. Below, we compute them with `einsum`, verify that they agree with a batched matrix product against the transposed keys, and count the total number of multiply–adds.

```
rng = np.random.default_rng()
queries = rng.standard_normal((2, 3, 4, 8))   # (batch, heads, queries, width)
keys = rng.standard_normal((2, 3, 5, 8))      # (batch, heads, keys, width)
scores = np.einsum('bhij,bhtj->bhit', queries, keys)
(scores.shape, np.allclose(scores, queries @ keys.swapaxes(-1, -2)),
 scores.size * queries.shape[-1])             # multiply-adds
```

```
((2, 3, 4, 5), True, 960)
```

The output has shape $(2,3,4,5)$, giving one 4×5 score table per sequence and head. Both expressions agree, and the contraction takes $2\cdot 3\cdot 4\cdot 5\cdot 8=960$ multiply–adds, which is simply the product of the sizes of all five indices.

Beyond code, index notation yields effortless one-line proofs. For any $\mathbf{A}\in\mathbb{R}^{m\times n}$ and $\mathbf{B}\in\mathbb{R}^{n\times m}$, both $\mathbf{AB}$ and $\mathbf{BA}$ are square, and their traces satisfy

$$\operatorname{tr}(\mathbf{AB})=a_{ij}b_{ji}=b_{ji}a_{ij}=\operatorname{tr}(\mathbf{BA}). \tag{33.13}$$

The middle step merely swaps two scalar numbers while leaving the indices alone. Surprisingly, the trace of a product does not depend on the order of the two factors, even when $\mathbf{AB} \neq \mathbf{BA}$ and even when the two products do not have the same dimensions. One immediate consequence will matter when we study eigenvalues: if $\mathbf{V}$ has an inverse $\mathbf{V}^{-1}$ (Section 33.2.2.3), then $\operatorname{tr}(\mathbf{VAV}^{-1}) = \operatorname{tr}(\mathbf{AV}^{-1}\mathbf{V}) = \operatorname{tr}(\mathbf{A})$. Because $\mathbf{VAV}^{-1}$ represents the same linear map $\mathbf{A}$ written in a different coordinate system, the trace is an intrinsic property of the map rather than an artifact of the coordinates we chose. The determinant will turn out to share this invariance (Section 33.2.3.3).

One practical warning: when an `einsum` expression combines three or more tensors, the string specifies the mathematical result but not the order in which pairwise contractions are carried out, and that order can make a huge difference in speed. For $n \times n$ matrices, computing $(\mathbf{AB})\mathbf{v}$ takes roughly n^3 multiply–adds, whereas computing $\mathbf{A}(\mathbf{Bv})$ takes only $2n^2$.

33.2.2 Rank and Invertibility

33.2.2.1 Column Space and Null Space

Which outputs can a matrix actually produce, and which inputs does it erase? To see both at once, examine our singular example

$$\mathbf{B} = \begin{bmatrix} 2 & 1 \\ 4 & 2 \end{bmatrix}.$$

Its columns are $\mathbf{b}_1 = (2, 4)$ and $\mathbf{b}_2 = (1, 2)$, so one is simply twice the other: $\mathbf{b}_1 = 2\mathbf{b}_2$. Because every output of $\mathbf{B}$ is a weighted sum of its columns, every output collapses to a scalar multiple of $\mathbf{b}_2$:

$$a_1\mathbf{b}_1 + a_2\mathbf{b}_2 = (2a_1 + a_2)\,\mathbf{b}_2.$$

Thus $\mathbf{B}$ sends the entire two-dimensional plane onto the single line $y = 2x$ through $(1, 2)$. Inevitably, distinct inputs collide on that line: both $(1, 1)$ and $(0, 3)$ land on $(3, 6)$. By linearity, their difference $(1, -2)$ must map to the difference of their outputs, $\mathbf{B}(1, -2) = \mathbf{0}$. The row view gives a second way to see why: both rows of $\mathbf{B}$, $(2, 1)$ and $(4, 2)$, are perpendicular to $(1, -2)$, so both hyperplane tests return zero. Figure 33.11 shows both lines side by side.

This example illustrates two fundamental subspaces attached to every $m \times n$ matrix $\mathbf{A}$. The *column space*, written $\operatorname{col}\mathbf{A}$, is the span of the columns of $\mathbf{A}$. Since $\mathbf{Ax}$ is a weighted combination of the columns with weights $x_1, \dots, x_n$, the column space is precisely the set of all reachable outputs. The *null space*, also called the *kernel* and written $\ker\mathbf{A}$, is the set of all inputs that $\mathbf{A}$ sends to zero:

$$\ker\mathbf{A} = \{\mathbf{x} \in \mathbb{R}^n : \mathbf{Ax} = \mathbf{0}\}.$$

It is a subspace of $\mathbb{R}^n$: if $\mathbf{Ax} = \mathbf{Ay} = \mathbf{0}$, then $\mathbf{A}(\alpha\mathbf{x} + \beta\mathbf{y}) = \alpha\mathbf{Ax} + \beta\mathbf{Ay} = \mathbf{0}$. For $\mathbf{B}$, the column space is the line $y = 2x$ in the output plane, and the null space is the line through $(1, -2)$ in the input plane. (That both live in $\mathbb{R}^2$ is just an accident of $\mathbf{B}$ being square; in general, $\ker\mathbf{A} \subseteq \mathbb{R}^n$ and $\operatorname{col}\mathbf{A} \subseteq \mathbb{R}^m$.)

Three general facts carry straight over from this example:

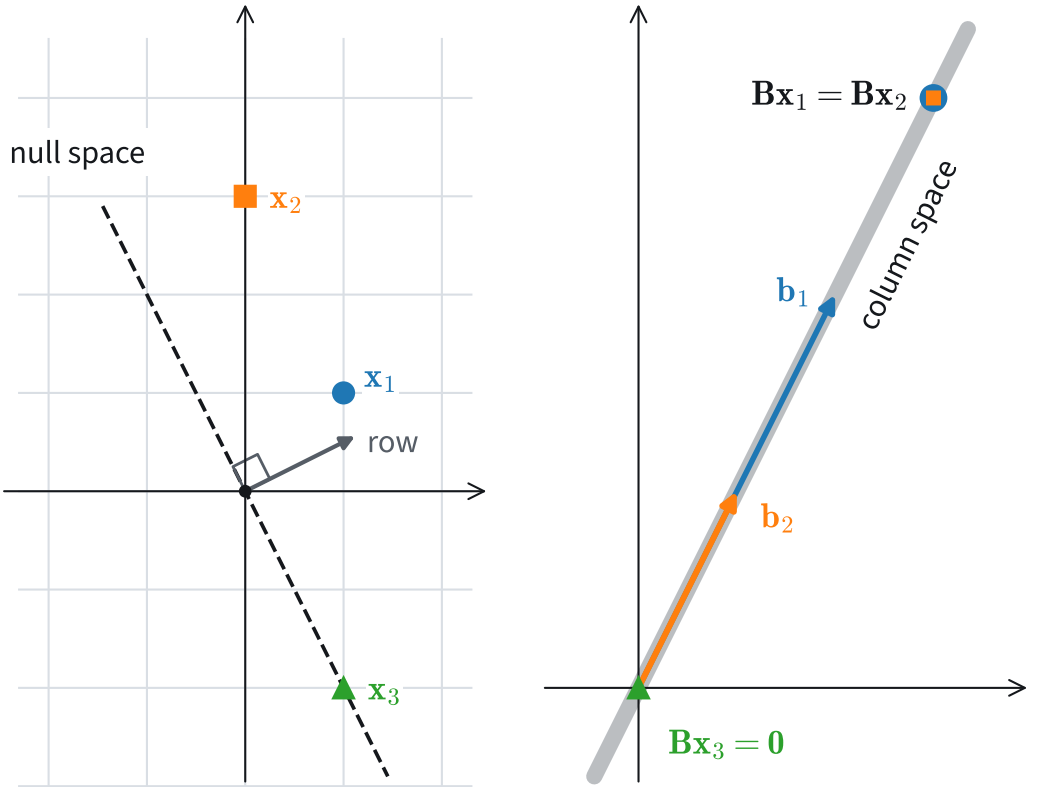


Figure 33.11: The matrix $\mathbf{B} = \left(\begin{smallmatrix} 2 & 1 \\ 4 & 2 \end{smallmatrix}\right)$ collapsing the plane. (a) In the input plane, the dashed null-space line through $(1, -2)$ is perpendicular to the row direction $(2, 1)$; three inputs are marked: $\mathbf{x}_1 = (1, 1)$, $\mathbf{x}_2 = (0, 3)$, and $\mathbf{x}_3 = (1, -2)$ on the null-space line. (b) In the output plane (same scale), every output lands on the column space, the line $y = 2x$ spanned by $\mathbf{b}_1$ and $\mathbf{b}_2$; both $\mathbf{x}_1$ and $\mathbf{x}_2$ map to $(3, 6)$, while $\mathbf{x}_3$ collapses to the origin. Because distinct inputs share the same image, no inverse can exist.

1. *The null space consists of all inputs perpendicular to every row.* In the row view, $(\mathbf{A}\mathbf{x})_i$ is the dot product of row i with $\mathbf{x}$, so all entries of $\mathbf{A}\mathbf{x}$ vanish if and only if $\mathbf{x}$ is orthogonal to every row.
2. *The columns are linearly dependent if and only if the null space contains a nonzero vector.* A nontrivial dependence $\sum_j x_j \mathbf{a}_j = \mathbf{0}$ among the columns $\mathbf{a}_j$ simply says $\mathbf{A}\mathbf{x} = \mathbf{0}$ for a nonzero vector $\mathbf{x}$, and reading the equation backwards gives the converse. For $\mathbf{B}$, the relation $\mathbf{b}_1 - 2\mathbf{b}_2 = \mathbf{0}$ yields the null vector $(1, -2)$.
3. *Two inputs produce the same output if and only if they differ by a null vector.* Indeed, $\mathbf{A}\mathbf{x} = \mathbf{A}\mathbf{y}$ is equivalent to $\mathbf{A}(\mathbf{x} - \mathbf{y}) = \mathbf{0}$. The set of all inputs that $\mathbf{B}$ maps to $(3, 6)$ is the shifted line $(1, 1) + t(1, -2)$, which passes through $(0, 3)$ at $t = -1$. Whenever $\ker \mathbf{A}$ contains a nonzero vector, distinct inputs collide and no inverse can exist.

33.2.2.2 Rank

How many input dimensions survive a linear map? The *rank* of a matrix is the dimension of its column space, or equivalently the maximum number of linearly independent columns:

$$\operatorname{rank} \mathbf{A} = \dim \operatorname{col} \mathbf{A}.$$

Our matrix $\mathbf{B}$ has rank 1: its two columns are dependent, and neither is zero. For a richer example, consider the 4×5 matrix

$$\mathbf{C} = \begin{bmatrix} 1 & 3 & 0 & -1 & 0 \\ -1 & 0 & 1 & 1 & -1 \\ 0 & 3 & 1 & 0 & -1 \\ 2 & 3 & -1 & -2 & 1 \end{bmatrix},$$

which has rank 2: its first two columns are linearly independent, yet every one of the $\binom{5}{3} = 10$ triples of columns is linearly dependent.

Remarkably, the number of dimensions that survive the map and the number of dimensions crushed to zero always add up to the dimension of the input space.

Proposition (rank–nullity). *For an $m \times n$ matrix $\mathbf{A}$,*

$$\operatorname{rank} \mathbf{A} + \dim \ker \mathbf{A} = n. \tag{33.14}$$

Proof. Pick a basis $\mathbf{w}_1, \dots, \mathbf{w}_k$ of $\ker \mathbf{A}$ and extend it to a basis $\mathbf{w}_1, \dots, \mathbf{w}_k, \mathbf{u}_1, \dots, \mathbf{u}_{n-k}$ of $\mathbb{R}^n$. (That any linearly independent list can be extended to a basis, and that all bases of a subspace have the same size, are standard facts we use without proof.) The images $\mathbf{A}\mathbf{u}_1, \dots, \mathbf{A}\mathbf{u}_{n-k}$ span $\operatorname{col} \mathbf{A}$, because any input $\mathbf{x} \in \mathbb{R}^n$ is a combination of the full basis and the $\mathbf{w}_j$ vanish under $\mathbf{A}$. They are also linearly independent: if $\sum_i c_i \mathbf{A}\mathbf{u}_i = \mathbf{0}$, then $\sum_i c_i \mathbf{u}_i$ lies in $\ker \mathbf{A}$ and can therefore be written as a combination of $\mathbf{w}_1, \dots, \mathbf{w}_k$. Within a basis, that is impossible unless every $c_i = 0$. Hence $\operatorname{col} \mathbf{A}$ has dimension $n - k$. $\blacksquare$

Figure 33.12 shows the basis that the proof builds. Each of its n vectors either vanishes under $\mathbf{A}$ or contributes one dimension to the column space, and no vector does both.

Let us check this conservation of dimensions on our examples. For $\mathbf{B}$ we get $1 + 1 = 2$. For the 256×784 Fashion-MNIST layer from the opening, the rank cannot exceed the 256 output coordinates, so the null space must have dimension at least $784 - 256 = 528$. For

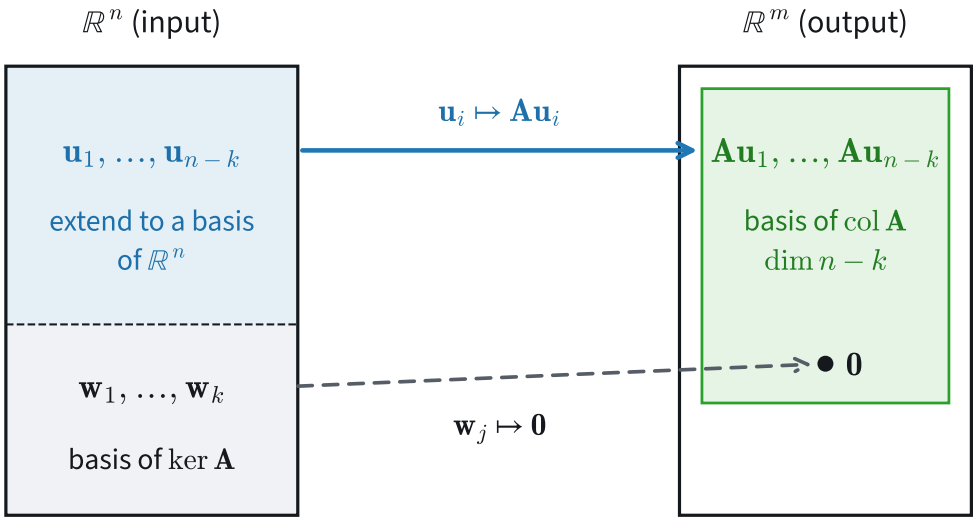


Figure 33.12: The basis in the proof of rank–nullity. The k vectors $\mathbf{w}_1, \dots, \mathbf{w}_k$ span $\ker \mathbf{A}$ and are sent to $\mathbf{0}$. The $n-k$ vectors $\mathbf{u}_1, \dots, \mathbf{u}_{n-k}$ complete them to a basis of $\mathbb{R}^n$, and their images form a basis of $\operatorname{col} \mathbf{A}$.

$\mathbf{C}$, the null space must have dimension $5 - 2 = 3$, meaning any three linearly independent null vectors form a basis of $\ker \mathbf{C}$. Below, we verify that $(3, -1, 3, 0, 0)$, $(1, 0, 0, 1, 0)$, and $(0, 0, 1, 0, 1)$ all map to zero under $\mathbf{C}$ and check the rank of the matrix `Z` holding them as columns.

```
B = np.array([[2.0, 1.0], [4.0, 2.0]])
C = np.array([[1, 3, 0, -1, 0], [-1, 0, 1, 1, -1],
              [0, 3, 1, 0, -1], [2, 3, -1, -2, 1]], dtype=float)
Z = np.array([[3, -1, 3, 0, 0], [1, 0, 0, 1, 0],
              [0, 0, 1, 0, 1]], dtype=float).T   # candidate null vectors
rank_C, rank_Z = np.linalg.matrix_rank(C), np.linalg.matrix_rank(Z)
np.linalg.matrix_rank(B), rank_C, np.abs(C @ Z).max(), rank_Z, rank_C + rank_Z
```

```
(np.int64(1), np.int64(2), np.float64(0.0), np.int64(3), np.int64(5))
```

The ranks of $\mathbf{B}$ and $\mathbf{C}$ come out to 1 and 2, the maximum entry of $|\mathbf{CZ}|$ is zero, and the three columns of `Z` have rank 3, confirming $2 + 3 = 5$.

What if we read a matrix by rows instead of columns? Its *row rank*, the maximum number of linearly independent rows, counts how many independent hyperplane tests the matrix performs. In $\mathbf{C}$, row 3 is row 1 plus row 2 and row 4 is row 1 minus row 2, so the row rank is 2, matching the column rank. That agreement is universal.

Proposition (row rank equals column rank). *For every matrix $\mathbf{A}$, the largest number of linearly independent rows equals the largest number of linearly independent columns.*

Proof. Let $r = \operatorname{rank} \mathbf{A}$, and place a basis of $\operatorname{col} \mathbf{A}$ into the columns of an $m \times r$ matrix $\mathbf{G}$. Since every column of $\mathbf{A}$ is a linear combination of the columns of $\mathbf{G}$, storing the coefficients

for column j of $\mathbf{A}$ in column j of an $r \times n$ matrix $\mathbf{H}$ gives the factorization

$$\mathbf{A} = \mathbf{G}\mathbf{H}. \tag{33.15}$$

Now read the same product by rows. Row i of $\mathbf{A}$ is $\sum_k g_{ik}\,(\text{row } k \text{ of } \mathbf{H})$, so every row of $\mathbf{A}$ lies in the span of the r rows of $\mathbf{H}$. Thus the row rank of $\mathbf{A}$ is at most r. Repeating the argument for $\mathbf{A}^\top$, whose columns are the rows of $\mathbf{A}$, shows that the column rank is at most the row rank, so the two must be equal. ■

The factorization $\mathbf{A} = \mathbf{G}\mathbf{H}$ is more than a proof device. It reveals that any rank-r matrix passes its input through a bottleneck of width r: $\mathbf{H}$ compresses $\mathbf{x} \in \mathbb{R}^n$ to r numbers, and $\mathbf{G}$ expands those r numbers back to $\mathbb{R}^m$. When r is much smaller than m and n, storing $\mathbf{G}$ and $\mathbf{H}$ requires only $r(m+n)$ parameters instead of mn, which is the core idea behind the low-rank updates of Section 33.5.4.1.

In practice, finding rank by testing subsets of columns would take exponentially long. By hand one uses Gaussian elimination (which this chapter does not cover). Numerical libraries like `np.linalg.matrix_rank` instead count how many *singular values* (Section 33.5) exceed a small tolerance. A tolerance is unavoidable because rank is not a continuous function of the matrix entries: a tiny perturbation can jump the rank from deficient to full (Section 33.2.3.6).

33.2.2.3 Inverses

Given an output $\mathbf{A}\mathbf{x}$, when can we reliably recover the input $\mathbf{x}$? An *inverse* of a square matrix $\mathbf{A}$ is a matrix $\mathbf{A}^{-1}$ that undoes $\mathbf{A}$ from both sides:

$$\mathbf{A}^{-1}\mathbf{A} = \mathbf{A}\mathbf{A}^{-1} = \mathbf{I},$$

so that $\mathbf{x} = \mathbf{A}^{-1}(\mathbf{A}\mathbf{x})$ for every input $\mathbf{x}$. For a 2×2 matrix with entries p, q, r, s satisfying $ps - qr \neq 0$, there is an explicit formula:

$$\mathbf{A}^{-1} = \frac{1}{ps - qr}\begin{bmatrix} s & -q \\ -r & p \end{bmatrix}. \tag{33.16}$$

A quick multiplication confirms it: multiplying $\left(\begin{smallmatrix} s & -q \\ -r & p \end{smallmatrix}\right)$ by $\mathbf{A}$ in either order yields $(ps - qr)\,\mathbf{I}$, and dividing by $ps - qr$ leaves $\mathbf{I}$. The condition $ps - qr \neq 0$ cannot be dropped. If $ps - qr = 0$, that product is the zero matrix; multiplying both sides by a hypothetical inverse $\mathbf{A}^{-1}$ would force $\left(\begin{smallmatrix} s & -q \\ -r & p \end{smallmatrix}\right) = \mathbf{0}$, and hence $\mathbf{A} = \mathbf{0}$, which plainly has no inverse. We will meet the quantity $ps - qr$ again as a signed area in Section 33.2.3.1. For our running matrix, $ps - qr = 3 + 2 = 5$, so

$$\mathbf{A}^{-1} = \frac{1}{5}\begin{bmatrix} 3 & -2 \\ 1 & 1 \end{bmatrix}, \qquad \mathbf{A}^{-1}(0, -5) = \tfrac{1}{5}\,(10, -5) = (2, -1).$$

Sure enough, $\mathbf{A}^{-1}$ reverses the worked example of Section 33.2.1.1.

Whenever an inverse exists, it is unique. Even better, any left inverse and any right inverse of the same matrix must coincide: if $\mathbf{L}\mathbf{A} = \mathbf{I}$ and $\mathbf{A}\mathbf{X} = \mathbf{I}$, then by associativity,

$$\mathbf{L} = \mathbf{L}(\mathbf{AX}) = (\mathbf{LA})\mathbf{X} = \mathbf{X}.$$

When does an inverse exist in general? For a square matrix, the answer requires only the null space and rank–nullity, and the rest of this chapter relies on it more than on any other result in this section.

Proposition (invertible matrices). *For an $n \times n$ matrix $\mathbf{A}$, the following are equivalent:* (i) $\mathbf{A}$ *is invertible;* (ii) *the columns of* $\mathbf{A}$ *are linearly independent;* (iii) $\ker \mathbf{A} = \{\mathbf{0}\}$*;* (iv) $\operatorname{rank} \mathbf{A} = n$.

Proof. Statements (ii) and (iii) are equivalent by fact 2 of Section 33.2.2.1, and (iii) and (iv) are equivalent by rank–nullity. If $\mathbf{A}$ is invertible and $\mathbf{Ax} = \mathbf{0}$, multiplying on the left by $\mathbf{A}^{-1}$ gives $\mathbf{x} = \mathbf{A}^{-1}\mathbf{Ax} = \mathbf{0}$, so (i) implies (iii).

The converse takes more work because we must actually construct the inverse. By (iv), the column space is an n-dimensional subspace of $\mathbb{R}^n$, which must be all of $\mathbb{R}^n$: appending any vector outside it to a basis of col $\mathbf{A}$ would produce $n+1$ linearly independent vectors in $\mathbb{R}^n$, which is impossible. Thus every target vector in $\mathbb{R}^n$ is reached. By (iii) and fact 3, no two inputs map to the same output. We can therefore define column j of a matrix $\mathbf{X}$ to be the unique solution of $\mathbf{Ax} = \mathbf{e}_j$. By construction, $\mathbf{AX} = \mathbf{I}$. To see that $\mathbf{X}$ also works on the left, take any $\mathbf{x} \in \mathbb{R}^n$: since $\mathbf{A}(\mathbf{XAx}) = (\mathbf{AX})\mathbf{Ax} = \mathbf{Ax}$ and no two inputs collide, we must have $\mathbf{XAx} = \mathbf{x}$. Hence $\mathbf{XA} = \mathbf{I}$ as well, and $\mathbf{X} = \mathbf{A}^{-1}$. ■

This immediately settles the claim left open in Section 33.2.1.3. For a square orthogonal matrix $\mathbf{Q}$ (satisfying $\mathbf{Q}^\top\mathbf{Q} = \mathbf{I}$), if $\mathbf{Qc} = \mathbf{0}$, then $\mathbf{c} = \mathbf{Q}^\top\mathbf{Qc} = \mathbf{0}$, so $\ker \mathbf{Q} = \{\mathbf{0}\}$. By the proposition, $\mathbf{Q}$ is invertible, its left inverse $\mathbf{Q}^\top$ must equal $\mathbf{Q}^{-1}$, and therefore $\mathbf{QQ}^\top = \mathbf{I}$.

A square matrix that has no inverse is called *singular*; $\mathbf{B}$ is our standard example. Rectangular matrices never have a two-sided inverse at all. The rank of an $m \times n$ matrix is at most $\min(m, n)$. When $m > n$, the column space has dimension at most $n < m$, so the map is not *onto*: some targets in $\mathbb{R}^m$ can never be hit. When $n > m$, rank–nullity forces a null space of dimension at least $n - m > 0$, so the map is not *one-to-one*: distinct inputs collide. Running a map backwards requires both properties, and the beauty of the proposition above is that for a square matrix, either one guarantees the other.

33.2.2.4 Solving Linear Systems in Practice

The matrix inverse is an indispensable tool for algebra and proofs, yet in numerical computation we almost never form it explicitly. When you need to solve $\mathbf{Ax} = \mathbf{b}$, use `np.linalg.solve(A, b)` rather than `np.linalg.inv(A) @ b`.

There are two main reasons: speed and accuracy. First, `solve` factors $\mathbf{A}$ once into triangular matrices (an *LU factorization*, costing about $n^3/3$ multiply–adds) and solves two fast triangular systems. Forming $\mathbf{A}^{-1}$ explicitly costs roughly n^3 multiply–adds, three times as much work, before you even multiply by $\mathbf{b}$. Second, `solve` is *backward stable*: its computed solution is the exact solution to a slightly perturbed system whose entries differ from $\mathbf{A}$ and $\mathbf{b}$ only at the level of floating-point rounding. Consequently, its residual $\|\mathbf{Ax} - \mathbf{b}\|$ stays on the order of machine precision times $\|\mathbf{A}\|\,\|\mathbf{x}\|$ (using the spectral norm of Section 1.3.4). Computing `inv(A) @ b` carries no such guarantee; its residual can be larger by a factor up to the *condition number* $\kappa(\mathbf{A})$, which measures how close $\mathbf{A}$ is to being singular (Section 33.5.3.2).

How close a matrix sits to singularity is a property of the problem itself, not of the algorithm. In double precision (roughly sixteen decimal digits), both methods can lose up to about $\log_{10} \kappa(\mathbf{A})$ digits of accuracy in the solution $\mathbf{x}$. That is a worst-case bound rather than an

exact prediction for any single right-hand side, so let us see how close a notorious example comes to it. The *Hilbert matrix*, with entries $h_{ij} = 1/(i + j - 1)$, is a classic test case of a nearly singular matrix. We take $\mathbf{x}_{\text{true}} = \mathbf{1}$, form $\mathbf{b} = \mathbf{H1}$ for the 10×10 Hilbert matrix $\mathbf{H}$, and attempt to recover $\mathbf{x}$ both ways.

```
H = 1 / (np.arange(10)[:, None] + np.arange(10) + 1)   # Hilbert matrix
x_true = np.ones(10)
b = H @ x_true
for x in (np.linalg.solve(H, b), np.linalg.inv(H) @ b):
    print(f'residual {np.linalg.norm(H @ x - b):.1e}, '
          f'error {np.linalg.norm(x - x_true):.1e}')
np.linalg.cond(H)
```

```
residual 5.7e-16, error 2.7e-04
residual 4.8e-05, error 9.0e-03
```

```
np.float64(16024416987428.36)
```

The condition number is $\kappa(\mathbf{H}) \approx 1.6 \cdot 10^{13}$, warning us that up to thirteen digits can vanish. As promised, `solve` keeps its residual near machine precision, while `inv` suffers a residual roughly eleven orders of magnitude larger, right in line with the factor κ. Notice, however, that a tiny residual does not guarantee an accurate solution. Both routes lose twelve or more of the sixteen digits, approaching the worst-case bound, with `inv` still worse by more than an order of magnitude.

Sparse matrices add a third, decisive reason never to invert. A million-by-million matrix with only five million nonzero entries fits comfortably in memory, whereas its inverse is typically completely dense, requiring roughly 10^{12} entries.

33.2.3 The Determinant

33.2.3.1 Area of a Parallelogram

By what factor does a matrix scale areas? Look back at Figure 33.8: our running matrix $\mathbf{A}$ sends the unit square (edges $\mathbf{e}_1$ and $\mathbf{e}_2$, area 1) to the parallelogram with edges $\mathbf{a}_1 = (1, -1)$ and $\mathbf{a}_2 = (2, 3)$ and corners $(0, 0)$, $(1, -1)$, $(3, 2)$, and $(2, 3)$. That parallelogram sits snugly inside the bounding box $[0, 3] \times [-1, 3]$ of area 12. The unused portion of the box consists of four right triangles with legs 1 and 1, 2 and 3, 1 and 1, and 2 and 3, whose areas sum to $\frac{1}{2} + 3 + \frac{1}{2} + 3 = 7$. Subtracting leaves an area of $12 - 7 = 5$ (Figure 33.13, panel (a)). The matrix $\mathbf{A}$ multiplies the area of the unit square by 5.

Carving out triangles works for one picture, but every new arrangement of corners would need a new diagram. Base times height handles every case in one stroke. Let the parallelogram have edges $\mathbf{u} = (p, r)$ and $\mathbf{w} = (q, s)$, the two columns of a general 2×2 matrix. Its base has length $\|\mathbf{u}\|$. Rotating $\mathbf{u}$ counter-clockwise by 90° gives $\mathbf{R}_{90}\mathbf{u} = (-r, p)$, so

$$\mathbf{n} = \frac{(-r, p)}{\|\mathbf{u}\|}$$

is a unit vector perpendicular to the base. From Section 33.1, the signed component of $\mathbf{w}$ along a unit vector $\mathbf{n}$ is the dot product $\mathbf{w} \cdot \mathbf{n}$. Geometrically, this is the height of the

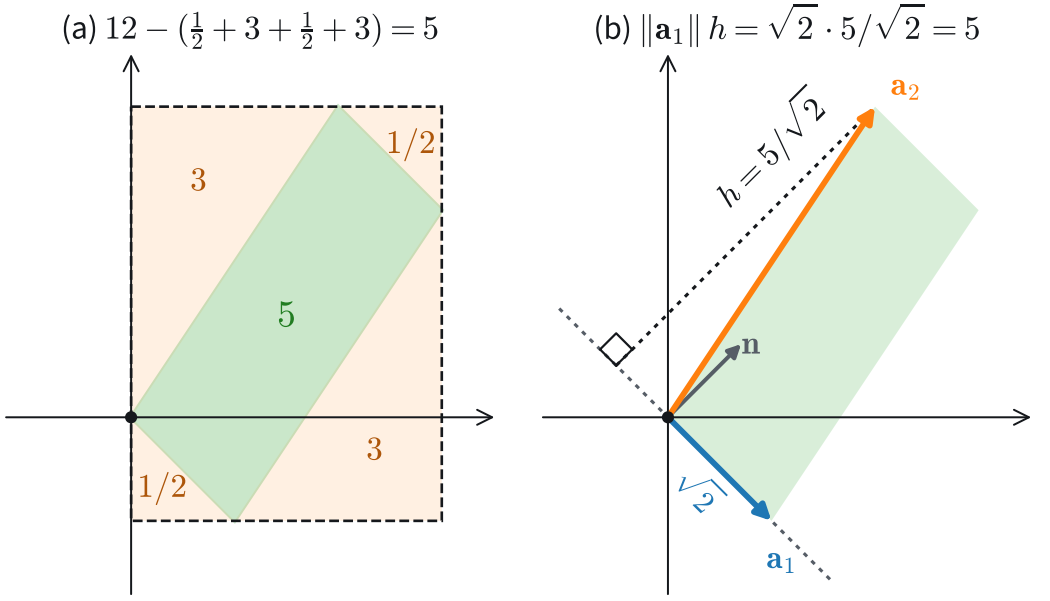


Figure 33.13: The image of the unit square under $\mathbf{A}$ has area 5. (a) Bounding-box method: the box $[0,3] \times [-1,3]$ of area 12 minus four right triangles of total area 7. (b) Base times height: the base $\mathbf{a}_1 = (1,-1)$ has length $\sqrt{2}$, the unit normal $\mathbf{n} = (1,1)/\sqrt{2}$ is perpendicular to it, and the height of $\mathbf{a}_2 = (2,3)$ above the base line is $\mathbf{a}_2 \cdot \mathbf{n} = 5/\sqrt{2}$.

parallelogram above its base, equipped with a sign: positive when $\mathbf{w}$ lies on the counter-clockwise side of $\mathbf{u}$, and negative when it lies on the clockwise side. Multiplying base by signed height gives

$$\|\mathbf{u}\|\,(\mathbf{w}\cdot\mathbf{n}) = \|\mathbf{u}\|\,\frac{-qr+ps}{\|\mathbf{u}\|} = ps - qr.$$

Conveniently, the lengths $\|\mathbf{u}\|$ cancel out, leaving a clean polynomial in the four matrix entries with no sign cases to split. For our running matrix, $\|\mathbf{u}\| = \sqrt{2}$, $\mathbf{n} = (1,1)/\sqrt{2}$, and $\mathbf{w}\cdot\mathbf{n} = 5/\sqrt{2}$, giving a signed area of 5 (panel (b)), positive because $\mathbf{a}_2$ lies counter-clockwise from $\mathbf{a}_1$.

We define the *determinant* of a 2×2 matrix to be this signed area of the image of the unit square:

$$\det \begin{bmatrix} p & q \\ r & s \end{bmatrix} = ps - qr. \tag{33.17}$$

Its magnitude $|\det \mathbf{A}|$ is the factor by which the map scales area, and its sign records whether the images of $\mathbf{e}_1$ and $\mathbf{e}_2$ preserve their counter-clockwise orientation. For $\mathbf{A}$, we get $1 \cdot 3 - 2 \cdot (-1) = 5$. Swapping the two columns of $\mathbf{A}$ produces $\left(\begin{smallmatrix} 2 & 1 \\ 3 & -1 \end{smallmatrix}\right)$, which spans the exact same parallelogram with its edges listed in the reverse order, and its determinant flips sign to

$-2 - 3 = -5$. For the singular matrix $\mathbf{B}$, we get $2 \cdot 2 - 1 \cdot 4 = 0$: the unit square flattens onto the line segment from $\mathbf{0}$ to $(3, 6)$, which has zero area.

Figure 33.14 contrasts all three regimes: a positive determinant preserves orientation, a negative determinant reverses it, and a zero determinant collapses the square.

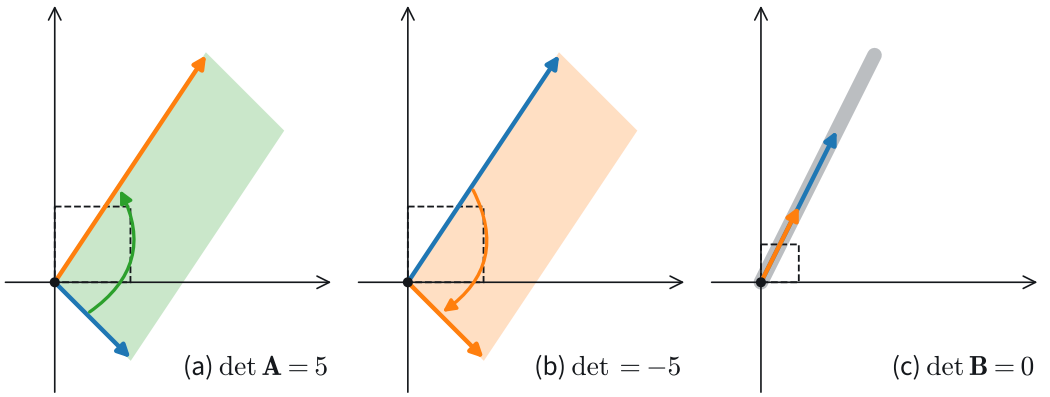


Figure 33.14: The determinant as signed area for our three 2×2 examples. The unit square (dashed) maps to the region spanned by the columns. (a) $\mathbf{A}$, with determinant 5: the columns keep their counter-clockwise order (curved arrow). (b) $\mathbf{A}$ with its columns swapped, determinant -5: the same parallelogram with reversed orientation. (c) $\mathbf{B}$, determinant 0: the square collapses onto a segment of the line $y = 2x$. Panel (c) is drawn at half the scale of (a) and (b).

Base times height gives us the plane. To move to n dimensions, we need the signed volume of an n-dimensional box.

33.2.3.2 The Determinant in General

How do we define the signed volume of the box spanned by n columns in $\mathbb{R}^n$? Rather than trying to compute high-dimensional volumes by chopping up shapes, we take a cleaner route: we write down the three geometric properties any signed volume must satisfy, and let those properties uniquely determine the formula. Viewed as a function of the columns $\mathbf{a}_1, \ldots, \mathbf{a}_n$ of an $n \times n$ matrix, the determinant is:

1. **Multilinear**: linear in each column separately while the other columns are held fixed. (Scaling one edge of a box scales its volume by the same factor; writing one edge as a sum splits the volume accordingly.)
2. **Alternating**: equal to zero whenever two columns are identical. (When two edges coincide, the box collapses into a lower-dimensional slab of zero volume.)
3. **Normalized**: $\det \mathbf{I} = 1$. (The unit cube has volume 1.)

Check that our 2×2 formula $ps - qr$ satisfies all three: it is linear in (p, r) and in (q, s), it vanishes whenever $(p, r) = (q, s)$, and it equals 1 on $\mathbf{I}$. Two powerful consequences follow immediately from these three axioms alone: a rule for swapping columns, and an explicit formula proving that at most one function can satisfy all three.

First, **swapping any two columns flips the sign of the determinant**. To see why, place $\mathbf{u}+\mathbf{w}$ in two column slots and expand using multilinearity:

$$\begin{aligned} 0 &= \det(\dots, \mathbf{u}+\mathbf{w}, \dots, \mathbf{u}+\mathbf{w}, \dots) \\ &= \det(\dots, \mathbf{u}, \dots, \mathbf{u}, \dots) + \det(\dots, \mathbf{u}, \dots, \mathbf{w}, \dots) \\ &\quad + \det(\dots, \mathbf{w}, \dots, \mathbf{u}, \dots) + \det(\dots, \mathbf{w}, \dots, \mathbf{w}, \dots). \end{aligned}$$

The left-hand side, the first term, and the last term all have two identical columns, so they vanish by the alternating property. What remains is

$$\det(\dots, \mathbf{u}, \dots, \mathbf{w}, \dots) = -\det(\dots, \mathbf{w}, \dots, \mathbf{u}, \dots).$$

Exchanging two edges of the box reverses its orientation, just as we saw in panel (b) of Figure 33.14.

Second, the three properties completely determine a formula. Expand each column in the standard basis as $\mathbf{a}_j = \sum_i a_{ij}\,\mathbf{e}_i$ and apply multilinearity column by column. This turns $\det \mathbf{A}$ into a large sum of terms $a_{i_1 1} a_{i_2 2} \cdots a_{i_n n}\, \det(\mathbf{e}_{i_1}, \dots, \mathbf{e}_{i_n})$, one for every choice of a row index i_j in each column j. Whenever two chosen row indices are equal, the term vanishes by the alternating property. When all n row indices are distinct, the list $\mathbf{e}_{i_1}, \dots, \mathbf{e}_{i_n}$ is just a reordering of the columns of $\mathbf{I}$, so repeated column swaps reduce $\det(\mathbf{e}_{i_1}, \dots, \mathbf{e}_{i_n})$ to ± 1.

In the 2×2 case, with columns $\mathbf{a}_1 = p\,\mathbf{e}_1 + r\,\mathbf{e}_2$ and $\mathbf{a}_2 = q\,\mathbf{e}_1 + s\,\mathbf{e}_2$, the expansion has four terms. The two with a repeated basis vector drop out, leaving

$$\det(\mathbf{a}_1, \mathbf{a}_2) = ps\,\det(\mathbf{e}_1, \mathbf{e}_2) + rq\,\det(\mathbf{e}_2, \mathbf{e}_1) = ps - qr,$$

recovering the signed area we derived by hand. In general, $\det \mathbf{A}$ is a signed sum over all ways of picking one entry from every row and every column. Writing $\sigma(j)$ for the row chosen in column j, we obtain the *permutation formula*:

$$\det \mathbf{A} = \sum_{\sigma} \operatorname{sgn}(\sigma)\, a_{\sigma(1)1}\, a_{\sigma(2)2} \cdots a_{\sigma(n)n}, \tag{33.18}$$

where σ ranges over all $n!$ permutations of $\{1, \dots, n\}$, and $\operatorname{sgn}(\sigma) = \det(\mathbf{e}_{\sigma(1)}, \dots, \mathbf{e}_{\sigma(n)})$ is $+1$ if σ requires an even number of swaps and -1 if it requires an odd number. Because any function satisfying the three axioms must equal this formula, there can be at most one determinant.

We do not prove existence in full here. The subtle point is that $\operatorname{sgn}(\sigma)$ is well defined (no permutation can be built from both an even and an odd number of swaps), a standard fact we take from (Horn and Johnson 2012). Once that parity fact is in hand, the formula satisfies all three axioms: each term contains one entry from each column (giving multilinearity), only the identity permutation survives for $\mathbf{I}$ (giving normalization), and the alternating property follows from the same parity argument (which we omit).

Notice that normalization entered only at the very end of the derivation. That observation gives us a sharper tool that we will use in the next subsection: if f is *any* multilinear and alternating function of the columns, the same swap rule and expansion apply, turning every factor $f(\mathbf{e}_{\sigma(1)}, \dots, \mathbf{e}_{\sigma(n)})$ into $\operatorname{sgn}(\sigma)\, f(\mathbf{I})$. Consequently,

$$f(\mathbf{A}) = f(\mathbf{I})\ \det \mathbf{A}.$$

In short, every multilinear alternating function of the columns is just a scalar multiple of the determinant.

For small hand calculations, grouping the terms of Equation 33.18 by the row picked in column 1 produces the *cofactor expansion*: each entry a_{i1} multiplies the determinant of the $(n-1)\times(n-1)$ submatrix formed by deleting row i and column 1, weighted by the alternating sign $(-1)^{i+1}$. For the 3×3 matrix

$$\mathbf{M} = \begin{bmatrix} 2 & 1 & 0 \\ 1 & 3 & 1 \\ 0 & 1 & 2 \end{bmatrix},$$

expanding along the first column gives

$$\begin{aligned} \det \mathbf{M} &= 2 \det \begin{bmatrix} 3 & 1 \\ 1 & 2 \end{bmatrix} - 1 \det \begin{bmatrix} 1 & 0 \\ 1 & 2 \end{bmatrix} + 0 \det \begin{bmatrix} 1 & 0 \\ 3 & 1 \end{bmatrix} \\ &= 2 \cdot 5 - 1 \cdot 2 + 0 = 8. \end{aligned} \tag{33.19}$$

Thus $\mathbf{M}$ maps the unit cube in $\mathbb{R}^3$ to a box of volume 8.

For a triangular matrix, the determinant simplifies dramatically: it is just the product of the diagonal entries. Of all the rules for computing determinants, this is the one to remember. Suppose $\mathbf{U}$ is upper triangular, so $u_{ij} = 0$ whenever $i > j$. In the permutation formula, a nonzero term must pick one entry from every column, all in distinct rows. From column 1, the only nonzero entry sits in row 1. From column 2, row 1 is already taken and rows $3, \dots, n$ are zero, forcing us to pick row 2. Continuing across the columns, only the diagonal permutation survives, and it requires zero swaps, so

$$\det \mathbf{U} = u_{11} u_{22} \cdots u_{nn}.$$

Running the same argument backwards from column n covers lower triangular matrices. Better still, it extends to blocks. Consider a *block upper triangular* matrix $\left(\begin{smallmatrix} \mathbf{D}_1 & * \\ \mathbf{0} & \mathbf{D}_2 \end{smallmatrix}\right)$ whose square diagonal blocks $\mathbf{D}_1$ and $\mathbf{D}_2$ need not be triangular themselves. Any nonzero permutation term must pick its entries in the columns of $\mathbf{D}_1$ from the rows of $\mathbf{D}_1$, because everything underneath is zero. That consumes all rows of $\mathbf{D}_1$, forcing the remaining columns to pick from the rows of $\mathbf{D}_2$; the off-diagonal block $*$ never contributes at all. Every surviving permutation splits into a pair of independent permutations on the two blocks, their signs multiply, and the sum factors cleanly:

$$\det \begin{bmatrix} \mathbf{D}_1 & * \\ \mathbf{0} & \mathbf{D}_2 \end{bmatrix} = \det \mathbf{D}_1\ \det \mathbf{D}_2.$$

Figure 33.15 marks the entries that survive in each case.

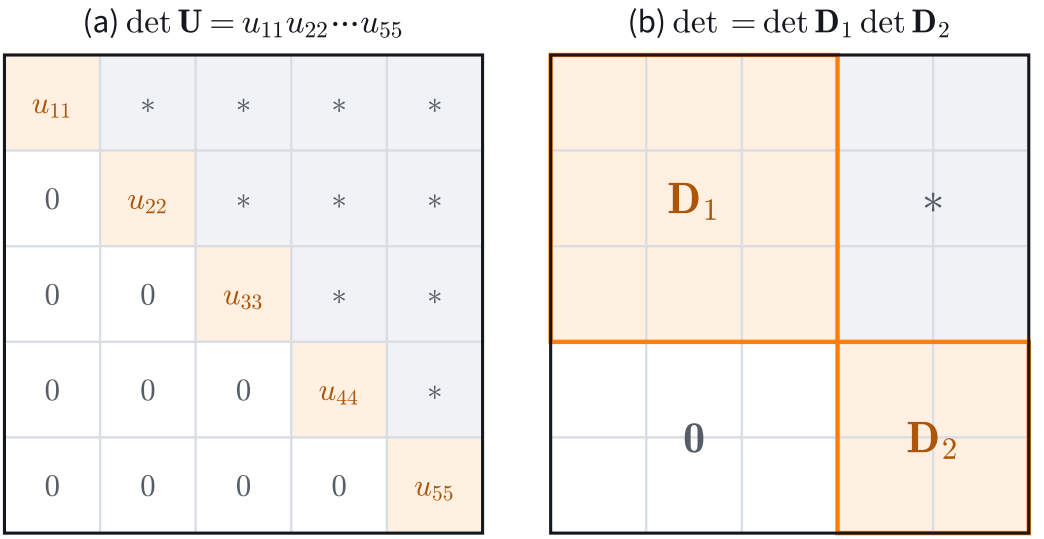


Figure 33.15: Entries that enter the determinant. (a) In an upper triangular matrix, the zeros below the diagonal rule out every permutation except the diagonal one, so the determinant is the product of the diagonal entries. (b) In a block upper triangular matrix, the zero block confines every surviving permutation to the diagonal blocks $\mathbf{D}_1$ and $\mathbf{D}_2$. The entries marked $*$ never enter.

These two rules are essential in modern machine learning, not just in hand calculations. A normalizing flow transforms a simple random variable through an invertible neural network and must evaluate the log-determinant of the network's Jacobian at every step, because a change of variables divides the probability density by the local volume scaling factor (Section 36.2.3.1). Flows are therefore designed so that each layer has a triangular or block-triangular Jacobian: the log-determinant then reduces to a sum of n scalar logarithms, avoiding the roughly $n^3/3$ multiply–adds required for a general matrix.

Another useful symmetry is that transposing leaves the determinant unchanged: $\det \mathbf{A}^\top = \det \mathbf{A}$. In the permutation formula for $\mathbf{A}^\top$, each term is $\operatorname{sgn}(\sigma) \prod_j a_{j\sigma(j)}$. Setting $j = \sigma^{-1}(i)$ rewrites the product as $\prod_i a_{\sigma^{-1}(i)\,i}$, which is the term for $\mathbf{A}$ corresponding to the inverse permutation σ^{-1}. Since undoing a sequence of swaps takes the exact same number of swaps, $\operatorname{sgn}(\sigma^{-1}) = \operatorname{sgn}(\sigma)$. The two sums therefore contain the exact same terms, which means every statement about columns of a determinant applies equally well to rows.

With $n!$ terms, the permutation formula is strictly a theoretical tool and is never used numerically beyond tiny matrices. Libraries compute determinants by factoring the matrix into triangular factors (as in `solve`) and multiplying their diagonal entries, adjusting the sign for any row exchanges, at a cost of roughly $n^3/3$ multiply–adds.

33.2.3.3 Multiplicativity

If $\mathbf{B}$ scales areas by $\det \mathbf{B}$ and $\mathbf{A}$ scales them by $\det \mathbf{A}$, then applying $\mathbf{B}$ followed by $\mathbf{A}$ should scale areas by the product of the two factors. Let us check two examples. For our running matrix, $\mathbf{A}^2 = \left(\begin{smallmatrix} -1 & 8 \\ -4 & 7 \end{smallmatrix}\right)$, which has $\det \mathbf{A}^2 = -7 + 32 = 25 = 5^2$. For the composite of

Section 33.2.1.2, $\det(\mathbf{R}_{90}\mathbf{T}) = 0 \cdot 1 - (-1) \cdot 1 = 1$, matching $\det \mathbf{R}_{90} \cdot \det \mathbf{T} = 1 \cdot 1$.

A simple tiling picture explains why this should hold in general. Cover any planar region with a fine grid of tiny squares. Because a linear map acts uniformly across space, it carries every square to a parallelogram whose area is $|\det \mathbf{A}|$ times that of the original square. Consequently, a linear map scales the area of *every* shape by $|\det \mathbf{A}|$. Applying $\mathbf{B}$ to a region of area V produces area $|\det \mathbf{B}|\, V$, and applying $\mathbf{A}$ next scales it to $|\det \mathbf{A}|\,|\det \mathbf{B}|\, V$. Since the composite $\mathbf{AB}$ is itself a single linear map, its scaling factor must be this product. Using the characterization from Section 33.2.3.2, we can turn that intuition into a three-line proof that handles the signs as well.

Proposition (multiplicativity of the determinant). *For square matrices* $\mathbf{A}$ *and* $\mathbf{B}$ *of the same size,*

$$\det(\mathbf{AB}) = \det(\mathbf{A})\ \det(\mathbf{B}). \tag{33.20}$$

Proof. Fix $\mathbf{A}$ and treat $D(\mathbf{B}) = \det(\mathbf{AB})$ as a function of the columns of $\mathbf{B}$. By Section 33.2.1.2, column j of $\mathbf{AB}$ is $\mathbf{Ab}_j$. Because $\mathbf{A}$ is linear and det is multilinear, D is multilinear in the columns of $\mathbf{B}$. If two columns of $\mathbf{B}$ coincide, the matching columns of $\mathbf{AB}$ coincide too, so $D(\mathbf{B}) = 0$ and D is alternating. Every multilinear alternating function of the columns is a scalar multiple of the determinant (Section 33.2.3.2), which gives

$$\det(\mathbf{AB}) = D(\mathbf{B}) = D(\mathbf{I})\ \det(\mathbf{B}) = \det(\mathbf{A})\ \det(\mathbf{B}). \qquad \blacksquare$$

Four consequences of Equation 33.20 appear repeatedly throughout the rest of this chapter:

1. *Determinants commute even when matrices do not.* Because determinants are ordinary scalars, $\det(\mathbf{AB}) = \det \mathbf{A}\ \det \mathbf{B} = \det \mathbf{B}\ \det \mathbf{A} = \det(\mathbf{BA})$. For our rotation and shear, $\mathbf{R}_{90}\mathbf{T} \neq \mathbf{TR}_{90}$, yet both products have determinant 1.

2. *Similarity invariance.* For any invertible matrix $\mathbf{V}$, applying consequence 1 to the factors $\mathbf{VA}$ and $\mathbf{V}^{-1}$ gives $\det(\mathbf{VAV}^{-1}) = \det(\mathbf{V}^{-1}\mathbf{VA}) = \det \mathbf{A}$. The matrix $\mathbf{VAV}^{-1}$ is simply the map $\mathbf{A}$ written in the coordinate system whose axes are the columns of $\mathbf{V}^{-1}$, and changing coordinates cannot alter the volume scaling factor.

3. *Inverses.* If $\mathbf{A}$ is invertible, setting $\mathbf{B} = \mathbf{A}^{-1}$ yields $\det \mathbf{A}\ \det \mathbf{A}^{-1} = \det \mathbf{I} = 1$. Therefore every invertible matrix satisfies $\det \mathbf{A} \neq 0$ and $\det \mathbf{A}^{-1} = 1/\det \mathbf{A}$ (for our running matrix, $1/5 = 0.2$).

4. *Orthogonal matrices.* Combining multiplicativity with $\det \mathbf{Q}^\top = \det \mathbf{Q}$ on $\mathbf{Q}^\top\mathbf{Q} = \mathbf{I}$ gives

$$\det(\mathbf{Q})^2 = \det(\mathbf{Q}^\top)\det(\mathbf{Q}) = \det(\mathbf{Q}^\top\mathbf{Q}) = \det(\mathbf{I}) = 1,$$

   so $\det \mathbf{Q} = \pm 1$. Rigid motions never change volume, and the sign distinguishes orientation-preserving rotations ($\det = +1$) from orientation-reversing reflections ($\det = -1$) in the plane (Section 33.2.1.3).

33.2.3.4 Invertibility and the Determinant

The determinant adds a fifth equivalent condition to the invertibility theorem of Section 33.2.2.3, and it is the only one expressed as a single number. In two dimensions the equivalence is immediate: the signed area $ps - qr$ vanishes if and only if the two columns lie on the same line through the origin (or one of them is $\mathbf{0}$), which is to say, if and only if the columns are linearly dependent. The same holds in every dimension.

Proposition (invertibility and the determinant). *An $n \times n$ matrix $\mathbf{A}$ is invertible if and only if* $\det \mathbf{A} \neq 0$. *Equivalently,* $\det \mathbf{A} = 0$ *if and only if the columns of $\mathbf{A}$ are linearly dependent.*

Proof. If $\mathbf{A}$ is invertible, then $\det \mathbf{A} \det \mathbf{A}^{-1} = 1$ by consequence 3 of multiplicativity, so $\det \mathbf{A} \neq 0$. Conversely, suppose $\mathbf{A}$ is not invertible. By the proposition in Section 33.2.2.3, its columns are linearly dependent, so some column is a combination of the others: $\mathbf{a}_j = \sum_{i \neq j} c_i \mathbf{a}_i$. Expanding column j by multilinearity gives $\det \mathbf{A} = \sum_{i \neq j} c_i \det(\dots, \mathbf{a}_i, \dots)$, where $\mathbf{a}_i$ now appears both in slot i and in slot j. Every term has two identical columns and therefore vanishes by the alternating property. ∎

For our running matrix, $\det \mathbf{A} = 5 \neq 0$ certifies invertibility; for $\mathbf{B}$, $\det \mathbf{B} = 0$ confirms singularity. Keep in mind, however, what this theorem does *not* promise: a nonzero determinant guarantees that $\mathbf{A}^{-1}$ exists mathematically, not that it is safe to compute numerically (Section 33.2.3.6).

33.2.3.5 Block Matrices and Low-Rank Updates

Often we already know the inverse of a part of a matrix: of one block, or of the matrix as it was before a small change. Can we reuse it? The answer comes from a single step of elimination, carried out on blocks instead of numbers. Split a square matrix $\mathbf{M}$ into four blocks $\mathbf{M}_{11}, \mathbf{M}_{12}, \mathbf{M}_{21}$, and $\mathbf{M}_{22}$, with $\mathbf{M}_{11}$ square and invertible. Subtracting $\mathbf{M}_{21}\mathbf{M}_{11}^{-1}$ times the first block row from the second clears the lower-left block, and the same step on the columns clears the upper-right one. Written as a product, this reads

$$\begin{bmatrix} \mathbf{M}_{11} & \mathbf{M}_{12} \\ \mathbf{M}_{21} & \mathbf{M}_{22} \end{bmatrix} = \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{M}_{21}\mathbf{M}_{11}^{-1} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{M}_{11} & \mathbf{0} \\ \mathbf{0} & \mathbf{S} \end{bmatrix} \begin{bmatrix} \mathbf{I} & \mathbf{M}_{11}^{-1}\mathbf{M}_{12} \\ \mathbf{0} & \mathbf{I} \end{bmatrix}, \qquad \mathbf{S} = \mathbf{M}_{22} - \mathbf{M}_{21}\mathbf{M}_{11}^{-1}\mathbf{M}_{12}, \tag{33.21}$$

as multiplying out the three factors confirms. The matrix $\mathbf{S}$ is the *Schur complement* of $\mathbf{M}_{11}$: it is what remains of $\mathbf{M}_{22}$ once the first block has been eliminated. For our running matrix with 1×1 blocks, $\mathbf{S} = 3 - (-1) \cdot 1^{-1} \cdot 2 = 5$.

Both outer factors are triangular with ones on the diagonal, so each has determinant 1. Multiplicativity and the block rule then give $\det \mathbf{M} = \det \mathbf{M}_{11} \det \mathbf{S}$ (in the example, $1 \cdot 5$), and $\mathbf{M}$ is invertible if and only if $\mathbf{S}$ is. An outer factor is inverted by flipping the sign of its off-diagonal block. Inverting the three factors in reverse order therefore yields

$$\mathbf{M}^{-1} = \begin{bmatrix} \mathbf{I} & -\mathbf{M}_{11}^{-1}\mathbf{M}_{12} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{M}_{11}^{-1} & \mathbf{0} \\ \mathbf{0} & \mathbf{S}^{-1} \end{bmatrix} \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ -\mathbf{M}_{21}\mathbf{M}_{11}^{-1} & \mathbf{I} \end{bmatrix}. \tag{33.22}$$

The lower-right block of $\mathbf{M}^{-1}$ is simply $\mathbf{S}^{-1}$, and the upper-left block is $\mathbf{M}_{11}^{-1} + \mathbf{M}_{11}^{-1}\mathbf{M}_{12}\mathbf{S}^{-1}\mathbf{M}_{21}\mathbf{M}_{11}^{-1}$. Nothing forces us to start in the upper-left corner, however. Eliminating $\mathbf{M}_{22}$ first exchanges the roles of the blocks, and the upper-left block of $\mathbf{M}^{-1}$ then

appears as the inverse of the other Schur complement, $\mathbf{M}_{11} - \mathbf{M}_{12}\mathbf{M}_{22}^{-1}\mathbf{M}_{21}$. That matrix returns as the conditional covariance of a Gaussian in Section 36.3.3.3.

The two expressions for the upper-left block must agree, and this is surprisingly useful. Suppose that we know $\mathbf{A}^{-1}$ for an $n \times n$ matrix $\mathbf{A}$, and that $\mathbf{A}$ then changes by a matrix of rank k, which we write as $\mathbf{U}\mathbf{W}^\top$ with $\mathbf{U}, \mathbf{W} \in \mathbb{R}^{n \times k}$. Choose the blocks $\mathbf{M}_{11} = \mathbf{A}$, $\mathbf{M}_{12} = -\mathbf{U}$, $\mathbf{M}_{21} = \mathbf{W}^\top$, and $\mathbf{M}_{22} = \mathbf{I}_k$. One Schur complement is then $\mathbf{A} + \mathbf{U}\mathbf{W}^\top$, and the other is the $k \times k$ matrix $\mathbf{I}_k + \mathbf{W}^\top\mathbf{A}^{-1}\mathbf{U}$. Equating the two upper-left blocks gives the *Woodbury identity*

$$(\mathbf{A} + \mathbf{U}\mathbf{W}^\top)^{-1} = \mathbf{A}^{-1} - \mathbf{A}^{-1}\mathbf{U}\,(\mathbf{I}_k + \mathbf{W}^\top\mathbf{A}^{-1}\mathbf{U})^{-1}\,\mathbf{W}^\top\mathbf{A}^{-1}. \tag{33.23}$$

A change of rank k in a matrix changes its inverse by a matrix of rank at most k, and finding that change requires only a $k \times k$ inverse. With $\mathbf{A}^{-1}$ in hand, the update costs on the order of n^2k multiply–adds instead of the n^3 of a fresh inversion. When $\mathbf{A}$ is diagonal, solving a system with $\mathbf{A} + \mathbf{U}\mathbf{W}^\top$ costs only on the order of nk^2. The two eliminations also give two expressions for $\det \mathbf{M}$, and hence $\det(\mathbf{A} + \mathbf{U}\mathbf{W}^\top) = \det \mathbf{A}\, \det(\mathbf{I}_k + \mathbf{W}^\top\mathbf{A}^{-1}\mathbf{U})$. For $k = 1$ the inner inverse is a division, and the identity is known as the *Sherman–Morrison formula*:

$$(\mathbf{A} + \mathbf{u}\mathbf{w}^\top)^{-1} = \mathbf{A}^{-1} - \frac{\mathbf{A}^{-1}\mathbf{u}\,\mathbf{w}^\top\mathbf{A}^{-1}}{1 + \mathbf{w}^\top\mathbf{A}^{-1}\mathbf{u}}. \tag{33.24}$$

This formula is why a least-squares fit can absorb or drop one example at a time: each example changes $\mathbf{X}^\top\mathbf{X}$ by a rank-one term. Exercise 6 applies both the determinant and the inverse to a normalizing flow.

33.2.3.6 Limits of the Determinant

Knowing that $\det \mathbf{A} \neq 0$ tells us a matrix can be inverted, but it says nothing about how *close* the matrix is to collapsing space, or along which directions. Two stark examples make the point from opposite extremes. First, the scaled identity $0.1\,\mathbf{I}_{400}$ is as well-behaved as a matrix can be, with inverse $10\,\mathbf{I}_{400}$, yet its determinant 10^{-400} is so tiny that it underflows beneath the smallest positive double-precision floating-point number. Second, the extreme shear $\left(\begin{smallmatrix} 1 & 10^8 \\ 0 & 1 \end{smallmatrix}\right)$ has determinant 1, yet changing its lower-left entry by just 10^{-8} makes it singular (5).

Below, we compute both determinants (using `slogdet` for the first to obtain its sign and log-magnitude safely) along with the singular values of the shear. Before running the cell, guess what rank NumPy will report for the *unchanged* shear.

```
sign, logdet = np.linalg.slogdet(0.1 * np.eye(400))
print(np.linalg.det(0.1 * np.eye(400)), sign, logdet)
shear = np.array([[1.0, 1e8], [0.0, 1.0]])
print(np.linalg.det(shear), np.linalg.matrix_rank(shear),
      np.linalg.svd(shear, compute_uv=False))
shear[1, 0] = 1e-8
print(np.linalg.det(shear), np.linalg.matrix_rank(shear))
```

```
0.0 1.0 -921.0340371976151
1.0 1 [1.e+08 1.e-08]
0.0 1
```

As expected, `det` on $0.1\,\mathbf{I}_{400}$ underflows to `0.0`, while `slogdet` computes $400 \ln 0.1 \approx -921.03$ effortlessly. The unchanged shear has determinant 1, and the perturbed shear has determinant 0 up to rounding, yet `matrix_rank` reports rank 1 for *both*. In exact arithmetic the unchanged shear has rank 2, but its two singular values are roughly 10^{8} and 10^{-8}. NumPy treats any singular value as zero once it falls below the largest singular value times the matrix size times machine precision, here $10^{8} \cdot 2 \cdot 2.2 \cdot 10^{-16} \approx 4 \cdot 10^{-8}$. Even though the shear's entries are stored exactly, floating-point arithmetic with numbers of size 10^{8} introduces rounding errors around 10^{-8}, so numerically the shear is indistinguishable from a singular matrix.

A single number simply cannot capture how a matrix stretches and compresses space along different directions. To see those directions, we need eigenvalues and singular values. For our running matrix $\mathbf{A}$, the eigenvalues are $2 \pm i$, whose product is $5 = \det \mathbf{A}$ (Section 33.3), and the singular values are roughly 3.618 and 1.382, whose product is $5 = |\det \mathbf{A}|$ (Section 33.5). In fact, for every square matrix, the determinant is the product of its eigenvalues (counted with multiplicity, including complex pairs), and $|\det \mathbf{A}|$ is the product of its singular values. It is the *ratio* of the largest to the smallest singular value, not the determinant, that measures how close a matrix is to collapsing space, and finding those directional factors is our next task.

33.2.4 Summary

A matrix is a linear map completely determined by its columns, which record where the standard coordinate axes land, and matrix multiplication is simply the composition of linear maps read from right to left. Einstein notation and `einsum` unify dot products, matrix products, traces, and multi-axis tensor contractions such as transformer attention under a single index rule.

Every $m \times n$ matrix defines a column space of reachable outputs and a null space of inputs sent to zero, linked by the rank–nullity theorem $\operatorname{rank} \mathbf{A} + \dim \ker \mathbf{A} = n$: a 256×784 layer necessarily erases at least 528 input dimensions. For a square matrix, having independent columns, trivial null space, full rank, a nonzero determinant, and an inverse are all equivalent. The determinant measures the signed factor by which a map scales volume; it multiplies under composition and equals the product of the diagonal entries for any triangular matrix. One step of block elimination produces the Schur complement, which gives the determinant and the inverse of a block matrix and, through the Woodbury identity, the inverse of a matrix after a low-rank change. Still, while a nonzero determinant guarantees that an inverse exists, it cannot tell us how close a matrix is to singularity, a question that leads directly to eigenvalues and singular values.

33.2.5 Exercises

1. **Maps that preserve angles.** Rotating every embedding vector by the same orthogonal matrix leaves all cosine similarities unchanged. This problem finds every linear map with that property.
    1. Find the condition on the columns of a square matrix $\mathbf{A}$ under which $\mathbf{A}\mathbf{e}_1$ and $\mathbf{A}\mathbf{e}_2$ are orthogonal. Show that $\mathbf{A} = \operatorname{diag}(1, 10)$ satisfies it but changes the angle between $(1, 1)$ and $(1, -1)$.
    2. Suppose that an invertible $\mathbf{A}$ preserves the angle between every pair of nonzero vectors. Apply this to the pairs $\mathbf{e}_i, \mathbf{e}_j$ and $\mathbf{e}_i + \mathbf{e}_j, \mathbf{e}_i - \mathbf{e}_j$ for $i \neq j$ to show that

$\mathbf{A}^\top\mathbf{A} = c^2\mathbf{I}$ for some $c > 0$, so that $\mathbf{A} = c\mathbf{Q}$ with $\mathbf{Q}$ orthogonal. Conversely, show that every such matrix preserves all cosine similarities.

3. Suppose instead that $\|\mathbf{Ax}\| = \|\mathbf{x}\|$ for every $\mathbf{x}$. Expand $\|\mathbf{x}+\mathbf{y}\|^2$ and $\|\mathbf{x}-\mathbf{y}\|^2$ to show that $\mathbf{A}$ preserves every dot product, and conclude that $\mathbf{A}$ is orthogonal.

2. **Rank of a product.** A linear autoencoder sends a d-dimensional input through a hidden layer of width $k < d$, $\mathbf{x} \mapsto \mathbf{W}_2\mathbf{W}_1\mathbf{x}$ with $\mathbf{W}_1 \in \mathbb{R}^{k\times d}$ and $\mathbf{W}_2 \in \mathbb{R}^{d\times k}$.
   1. Let $\mathbf{A}$ be an $m \times n$ matrix and $\mathbf{B}$ an $n \times p$ matrix. Show that the column space of $\mathbf{AB}$ lies in the column space of $\mathbf{A}$, and that the null space of $\mathbf{B}$ lies in the null space of $\mathbf{AB}$. Use the first fact, and rank–nullity with the second, to prove $\operatorname{rank}(\mathbf{AB}) \le \min(\operatorname{rank}\mathbf{A}, \operatorname{rank}\mathbf{B})$.
   2. Conclude that $\mathbf{W}_2\mathbf{W}_1$ has rank at most k. What is the smallest possible dimension of its null space? What is the largest dimension of a subspace on which the autoencoder reconstructs every input exactly?
   3. Construct a 2×3 matrix and a 3×2 matrix, both of rank two, whose 2×2 product has rank one.

3. **Matrices that commute with a rotation.** Matrix products do not commute in general. This problem finds the exceptions for $\mathbf{R}_{90}$.
   1. Find all 2×2 matrices $\mathbf{X}$ with $\mathbf{XR}_{90} = \mathbf{R}_{90}\mathbf{X}$, and show that they are $\left(\begin{smallmatrix} p & -q \\ q & p \end{smallmatrix}\right) = p\mathbf{I} + q\mathbf{R}_{90}$.
   2. Using $\mathbf{R}_{90}^2 = -\mathbf{I}$, show that $(p\mathbf{I}+q\mathbf{R}_{90})(r\mathbf{I}+s\mathbf{R}_{90}) = (pr-qs)\,\mathbf{I}+(ps+qr)\,\mathbf{R}_{90}$, so these matrices multiply like the complex numbers $p + qi$.
   3. For $(p, q) \neq (0, 0)$, show that $p\mathbf{I} + q\mathbf{R}_{90} = \sqrt{p^2+q^2}\,\mathbf{R}_\phi$ for the angle ϕ with $(\cos\phi, \sin\phi) = (p, q)/\sqrt{p^2+q^2}$, a scaled rotation. Compute its determinant, and conclude that it is invertible unless $p = q = 0$.
   4. Does the running matrix $\mathbf{A}$ commute with $\mathbf{R}_{90}$? Compute both products.

4. **Householder reflections.** Let $\mathbf{u}$ be a unit vector in $\mathbb{R}^n$ and $\mathbf{H} = \mathbf{I} - 2\mathbf{uu}^\top$.
   1. Show that $\mathbf{H}$ is symmetric and orthogonal, that $\mathbf{Hu} = -\mathbf{u}$, and that $\mathbf{Hx} = \mathbf{x}$ for every $\mathbf{x}$ perpendicular to $\mathbf{u}$, so that $\mathbf{H}$ reflects in the hyperplane with normal $\mathbf{u}$.
   2. Write $\mathbf{H} = \mathbf{I} - 2\mathbf{P}$ with $\mathbf{P} = \mathbf{uu}^\top$, and explain the formula with the projection onto $\mathbf{u}$ from Section 33.1: what does $\mathbf{x} - 2\mathbf{Px}$ do to the component of $\mathbf{x}$ along $\mathbf{u}$?
   3. Prove $\det\mathbf{H} = -1$. Extend $\mathbf{u}$ to an orthonormal basis, the columns of an orthogonal matrix $\mathbf{Q} = [\mathbf{u}\,\mathbf{q}_2 \cdots \mathbf{q}_n]$, compute $\mathbf{Q}^\top\mathbf{HQ}$, and use multiplicativity.
   4. Find the unit vector $\mathbf{u}$ whose reflection sends $(3, 4)$ to $(5, 0)$ (hint: $\mathbf{u}$ is parallel to $(3, 4) - (5, 0)$), write down $\mathbf{H}$, and check $\det\mathbf{H} = -1$. Reflections of this kind, applied column by column, are how libraries factor a matrix into an orthogonal and a triangular factor.

5. code **Determinant and near-singularity.** A square matrix is invertible if and only if its determinant is nonzero. How well does the size of the determinant measure how close a matrix is to being singular?
   1. Use multilinearity to prove $\det(c\mathbf{A}) = c^n \det\mathbf{A}$ for an $n \times n$ matrix $\mathbf{A}$ and a scalar c, and check the identity in code for the matrix $\mathbf{M}$ of Equation 33.19 with $c = 2$.
   2. Draw an $n \times n$ lower triangular matrix with independent standard Gaussian entries on and below the diagonal, for $n = 20$, 100, and 1000. Compare the logarithm that `np.linalg.slogdet` returns with the sum of the logarithms of the absolute diagonal entries, which the triangular rule makes exact up to rounding. From which n on do the two disagree, and by how much at $n = 1000$? Compute `np.linalg.cond` for $n = 20$ and $n = 100$, and explain the failure. (The routine does not know that the matrix is triangular. It factors it with row exchanges, like any other matrix.)

3. The text states that a change of 10^{-8} in the lower left entry makes the shear $\left(\begin{smallmatrix} 1 & 10^{8} \\ 0 & 1 \end{smallmatrix}\right)$ singular. Verify it, and show that no smaller change of a single entry does.

6. **Determinant of a rank-one update.** ● A *planar flow* (Rezende and Mohamed 2015) transforms $\mathbf{z} \in \mathbb{R}^d$ by $f(\mathbf{z}) = \mathbf{z} + \mathbf{u}\, h(\mathbf{w} \cdot \mathbf{z} + b)$ for vectors $\mathbf{u}, \mathbf{w}$, a scalar b, and a scalar nonlinearity h. Computing the density of $f(\mathbf{z})$ requires the determinant of the Jacobian $\mathbf{I} + h'(\mathbf{w} \cdot \mathbf{z} + b)\, \mathbf{u}\mathbf{w}^\top$.
   1. Use multilinearity and the alternating property to show $\det(\mathbf{u}\mathbf{w}^\top) = 0$ for $n \times n$ matrices with $n \geq 2$.
   2. Verify the two factorizations

$$\begin{aligned} \begin{bmatrix} \mathbf{I} & -\mathbf{u} \\ \mathbf{w}^\top & 1 \end{bmatrix} &= \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{w}^\top & 1 \end{bmatrix} \begin{bmatrix} \mathbf{I} & -\mathbf{u} \\ \mathbf{0}^\top & 1 + \mathbf{w} \cdot \mathbf{u} \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{I} & -\mathbf{u} \\ \mathbf{0}^\top & 1 \end{bmatrix} \begin{bmatrix} \mathbf{I} + \mathbf{u}\mathbf{w}^\top & \mathbf{0} \\ \mathbf{w}^\top & 1 \end{bmatrix}. \end{aligned}$$

      Use the permutation formula Equation 33.18 to show that a matrix whose last column is $\mathbf{e}_{n+1}$ has the determinant of its upper-left $n \times n$ block. Deduce $\det(\mathbf{I} + \mathbf{u}\mathbf{w}^\top) = 1 + \mathbf{w} \cdot \mathbf{u}$ from Equation 33.20 and the triangular rule.
   3. Write the logarithm of the absolute Jacobian determinant of a planar flow, and compare the cost of evaluating it with the cost of a general $d \times d$ determinant. For $h = \tanh$, which values of $\mathbf{u} \cdot \mathbf{w}$ keep the Jacobian invertible at every $\mathbf{z}$?
   4. Use the Sherman–Morrison formula Equation 33.24 to write the inverse of the Jacobian $\mathbf{I} + c\, \mathbf{u}\mathbf{w}^\top$, where $c = h'(\mathbf{w} \cdot \mathbf{z} + b)$. Check that the formula breaks down at the same values of c at which the determinant vanishes, and count the multiply–adds needed to apply the inverse to a vector. The inverse Jacobian does not invert f itself: show that recovering $\mathbf{z}$ from $f(\mathbf{z})$ reduces to solving one scalar equation for $\mathbf{w} \cdot \mathbf{z}$.

7. code **Einstein summation for attention.** Let the query, key and value tensors have the shapes (n_b, n_h, n_q, d_h), (n_b, n_h, n_k, d_h) and (n_b, n_h, n_k, d_h).
   1. Write the `einsum` strings for the scores $s_{bhit} = \sum_j q_{bhij} k_{bhtj}$ and for the weighted sum of values $o_{bhij} = \sum_t p_{bhit}\, v_{bhtj}$, where p_{bhit} are the attention weights, and check both against `@` with `swapaxes`.
   2. Count the multiply–adds of each contraction as a product of the sizes. Compare them with the multiply–adds of applying one $d_h \times d_h$ linear map to every query, and find the number of keys n_k beyond which the scores cost more.
   3. Write $\operatorname{tr}(\mathbf{A}^\top \mathbf{A})$, $\operatorname{tr}(\mathbf{A}\mathbf{A}^\top)$ and $\sum_{ij} a_{ij}^2$ as one `einsum` each for a random 3×5 matrix, and check that they agree. Which of the two equalities is an instance of Equation 33.13, and why does the other one hold?

33.3 Eigendecompositions

In Section 33.2 we asked what a matrix does to space when it is applied once. Many computations in machine learning, however, apply the same matrix over and over. A deep linear network with identical layers computes $\mathbf{A}^{50}\mathbf{x}$ after 50 layers, and gradient descent on a quadratic objective multiplies its error by the same fixed matrix at every step. In each case we want to know whether the signal grows, dies out, or settles down, and how quickly that happens. Multiplying out the matrix powers gives us the numbers, but it does not explain them.

A random walk between two web pages is the smallest version of this question. Suppose a surfer on page 1 stays on page 1 or clicks to page 2 with probability $1/2$ each; from page 2 she stays with probability $3/4$ and returns to page 1 with probability $1/4$. Let $\mathbf{p}_k$ hold the probabilities of being on each page after k clicks. Each click multiplies this vector by a fixed matrix,

$$\mathbf{p}_{k+1} = \mathbf{P}\mathbf{p}_k, \qquad \mathbf{P} = \begin{pmatrix} 1/2 & 1/4 \\ 1/2 & 3/4 \end{pmatrix},$$

whose columns record the click probabilities from each page and therefore sum to one. Starting on page 1 with $\mathbf{p}_0 = (1, 0)$, the first few distributions are $(0.5, 0.5)$, $(0.375, 0.625)$, $(0.344, 0.656)$, and $(0.336, 0.664)$. The changes shrink with every click, as if the walk were settling onto a fixed distribution. Where is the surfer after 50 clicks, and how fast does the distribution converge? PageRank asks the very same question of the entire web.

The key is to rewrite the starting vector in the coordinates of a few special directions, the eigenvectors, along which the matrix acts simply as multiplication by a number. In those coordinates, each component evolves on its own as a scalar power λ^k.

In Section 1.3 we already met eigenvectors as the directions that a matrix only stretches: the symmetric matrix $\mathbf{S} = \left(\begin{smallmatrix} 2 & 1 \\ 1 & 2 \end{smallmatrix}\right)$ stretches $(1, 1)$ by 3 and $(1, -1)$ by 1, and for any symmetric matrix we had $\mathbf{A}^k\mathbf{x} = \sum_i c_i \lambda_i^k \mathbf{v}_i$. Here we drop the symmetry requirement. We learn how to find the eigenvalues of an arbitrary square matrix, see when a full set of eigenvectors exists and what replaces it when one does not, and then prove why symmetry guarantees so much more.

Three results carry the most weight: the eigenvector expansion Equation 33.28 of $\mathbf{A}^k\mathbf{x}$, the spectral theorem, and the quadratic form written as a weighted sum of squares, Equation 33.36. A reader in a hurry can jump from Section 33.3.1.3 straight to Section 33.3.2.1. We leave two practical questions to Section 33.4: how to locate and compute eigenvalues when a matrix is large, and the situations in which eigenvalues can mislead us. Throughout this section we compute with NumPy, whose `eig` function returns complex eigenvalues whenever a real matrix has them.

```
import numpy as np
```

33.3.1 Eigenvalues and Diagonalization

33.3.1.1 Eigenvectors and Eigenvalues

Certain input vectors are only rescaled by $\mathbf{P}$, as though the matrix were a single number. Applying $\mathbf{P}$ to $(1, 2)$ gives $\mathbf{P}(1, 2) = (\frac{1}{2}+\frac{1}{2},\ \frac{1}{2}+\frac{3}{2}) = (1, 2)$, leaving it completely unchanged, while applying $\mathbf{P}$ to $(1, -1)$ gives $(\frac{1}{2} - \frac{1}{4},\ \frac{1}{2} - \frac{3}{4}) = \frac{1}{4}(1, -1)$. Along the first direction $\mathbf{P}$ multiplies by 1; along the second it multiplies by $1/4$.

In general, a nonzero vector $\mathbf{v}$ is called an *eigenvector* of a square matrix $\mathbf{A}$, with corresponding *eigenvalue* λ, if

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v}. \tag{33.25}$$

In geometric terms, **A** only stretches an eigenvector by the factor λ, reversing its direction when $\lambda < 0$. For the surfer's matrix **P**, the eigenpairs $(\lambda, \mathbf{v})$ are $(1, (1, 2))$ and $(\frac{1}{4}, (1, -1))$. We exclude the zero vector from being an eigenvector because it trivially satisfies Equation 33.25 for every λ. However, all solutions of $\mathbf{Av} = \lambda\mathbf{v}$ for a fixed λ, together with $\mathbf{0}$, form a linear subspace called the *eigenspace* of λ.

The eigenvalue $\lambda = 0$, on the other hand, is perfectly valid. It means that $\mathbf{Av} = \mathbf{0}$ for some nonzero **v**, so a square matrix is singular if and only if 0 is one of its eigenvalues. Notice also that a real matrix need not have any real eigenvectors: a rotation by 90° turns every nonzero direction in the plane. Its eigenvalues turn out to be complex, a case we take up in Section 33.3.1.6.

It is tempting to confuse eigenvectors with the axes of the ellipse produced when an invertible 2×2 matrix acts on the unit circle. Figure 33.16 shows why the two concepts are distinct. For the symmetric matrix **S**, the eigenvectors do lie along the ellipse axes. For the upper-triangular matrix $\left(\begin{smallmatrix} 1 & 1 \\ 0 & 2 \end{smallmatrix}\right)$, however, they do not: its eigenvectors are $(1, 0)$ with $\lambda = 1$ and $(1, 1)$ with $\lambda = 2$, whereas the ellipse axes point at 58° and -32° (Section 33.5 shows how to compute them). The ellipse axes mark the directions of largest and smallest stretch, while the eigenvectors mark the directions that are not rotated at all. For symmetric matrices these two sets of directions always coincide, as the spectral theorem of Section 33.3.2.1 proves; for general matrices they need not.

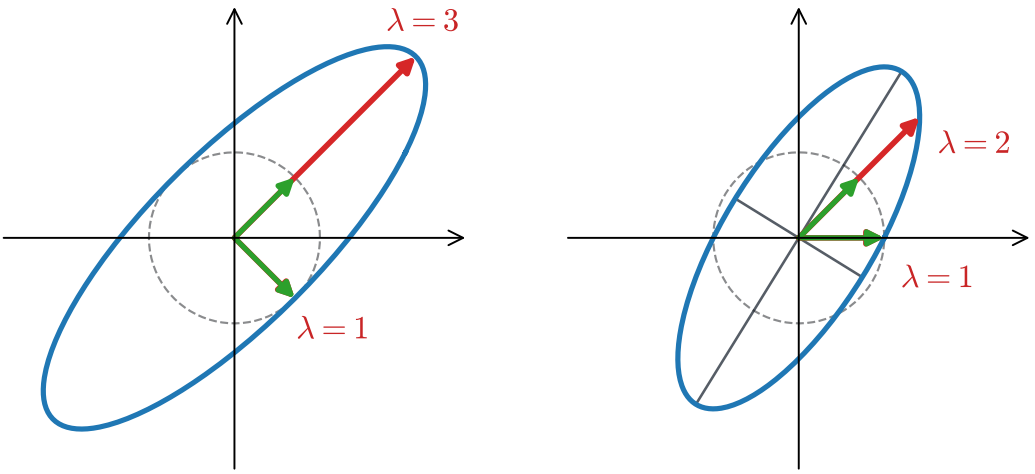


Figure 33.16: The unit circle (dashed) and its image (blue). Left: for the symmetric matrix $\left(\begin{smallmatrix} 2 & 1 \\ 1 & 2 \end{smallmatrix}\right)$, the unit eigenvectors (green) lie along the ellipse axes, and their images (red) are scaled by 3 and 1 (for $\lambda = 1$ the two arrows coincide). Right: for the upper-triangular matrix $\left(\begin{smallmatrix} 1 & 1 \\ 0 & 2 \end{smallmatrix}\right)$, the eigenvectors $(1, 0)$ and $(1, 1)$ do not align with the ellipse axes (grey).

33.3.1.2 The Characteristic Polynomial

We spotted the eigenvectors of **P** by inspection, which does not take us far on larger matrices. To find the eigenvalues of any square matrix systematically, move $\lambda\mathbf{v}$ to the left-hand side of Equation 33.25 and factor out **v**:

$$(\mathbf{A} - \lambda\mathbf{I})\mathbf{v} = \mathbf{0}. \tag{33.26}$$

A nonzero solution $\mathbf{v}$ exists if and only if $\mathbf{A}-\lambda\mathbf{I}$ has a nontrivial null space—that is, if and only if it is singular, so that $\det(\mathbf{A}-\lambda\mathbf{I})=0$. Expanding this determinant yields a polynomial of degree n in λ, called the *characteristic polynomial* $p(\lambda)=\det(\mathbf{A}-\lambda\mathbf{I})$. Its roots are precisely the eigenvalues of $\mathbf{A}$. Once a root λ is known, solving the homogeneous linear system Equation 33.26 gives its eigenspace.

Let us run this recipe on the surfer's matrix:

$$\det(\mathbf{P}-\lambda\mathbf{I})=\left(\tfrac{1}{2}-\lambda\right)\left(\tfrac{3}{4}-\lambda\right)-\tfrac{1}{4}\cdot\tfrac{1}{2}=\lambda^2-\tfrac{5}{4}\lambda+\tfrac{1}{4}=(\lambda-1)\left(\lambda-\tfrac{1}{4}\right).$$

The roots are 1 and $1/4$, matching the two scaling factors we noticed earlier. Because $\mathbf{P}-\lambda\mathbf{I}$ is singular at each root, a single row suffices to find the eigenvector. For $\lambda=1$, the first row of $\mathbf{P}-\mathbf{I}$ gives $-\frac{1}{2}v_1+\frac{1}{4}v_2=0$, so $v_2=2v_1$ and we may choose $\mathbf{v}=(1,2)$. For $\lambda=\frac{1}{4}$, the first row of $\mathbf{P}-\frac{1}{4}\mathbf{I}$ gives $\frac{1}{4}v_1+\frac{1}{4}v_2=0$, yielding $\mathbf{v}=(1,-1)$. Keep in mind that every eigenvector is determined only up to a nonzero scalar factor: if $\mathbf{v}$ is an eigenvector, then so is $c\mathbf{v}$ for any $c\neq 0$, since both sides of Equation 33.25 scale by c.

In practice we leave this algebra to a numerical library. NumPy's `eig` returns the eigenvalues together with a matrix whose columns are the corresponding eigenvectors; the last line below checks Equation 33.25 for all pairs at once.

```
P = np.array([[0.5, 0.25], [0.5, 0.75]])
lam, V = np.linalg.eig(P)
print('eigenvalues:', lam)
print('eigenvectors (columns):\n', V)
print('max |P V - V diag(lam)|:', np.abs(P @ V - V * lam).max())
```

```
eigenvalues: [0.25 1.  ]
eigenvectors (columns):
 [[-0.70710678 -0.4472136 ]
 [ 0.70710678 -0.89442719]]
max |P V - V diag(lam)|: 0.0
```

Order, scale, and sign are all at the library's discretion: here `eig` returns the eigenvalues as $1/4$ and 1, and the columns are unit-length vectors parallel to $(1,-1)$ and $(1,2)$.

Numerical libraries do not actually form and solve the characteristic polynomial. For $n\geq 5$ there is no general closed-form formula for polynomial roots in radicals, and even for moderate n the roots are notoriously sensitive to tiny rounding errors in the polynomial coefficients. Instead, `eig` applies a sequence of orthogonal changes of basis that gradually drive the matrix into triangular form without altering its eigenvalues. A triangular matrix displays its eigenvalues right on its diagonal: subtracting $\lambda\mathbf{I}$ keeps it triangular, and the determinant of a triangular matrix is simply the product of its diagonal entries (Section 33.2.3.2). For matrices far too large to factor at all, power iteration (Section 33.4.1.2) extracts dominant eigenvalues using only matrix–vector products.

33.3.1.3 Diagonalization

We can now answer our opening question about $\mathbf{P}^{50}\mathbf{p}_0$. Collect the two eigenvectors of $\mathbf{P}$ as the columns of a matrix $\mathbf{V}$, and place their eigenvalues on the diagonal of Λ:

$$\mathbf{V} = \begin{pmatrix} 1 & 1 \\ 2 & -1 \end{pmatrix}, \qquad \Lambda = \begin{pmatrix} 1 & 0 \\ 0 & 1/4 \end{pmatrix}.$$

Written side by side as columns, the two eigenvector equations $\mathbf{P}\mathbf{v}_i = \lambda_i \mathbf{v}_i$ compactly read $\mathbf{P}\mathbf{V} = \mathbf{V}\Lambda$. Because $\det \mathbf{V} = -1 - 2 = -3 \neq 0$, the matrix $\mathbf{V}$ is invertible, with $\mathbf{V}^{-1} = \frac{1}{3}\left(\begin{smallmatrix} 1 & 1 \\ 2 & -1 \end{smallmatrix}\right)$. Multiplying on the right by $\mathbf{V}^{-1}$ yields

$$\mathbf{P} = \mathbf{V}\Lambda\mathbf{V}^{-1}. \tag{33.27}$$

Read this factorization from right to left as three steps: change to eigenvector coordinates ($\mathbf{V}^{-1}$), scale each coordinate independently by its eigenvalue (Λ), and change back to the original coordinates ($\mathbf{V}$). A square matrix $\mathbf{A}$ that can be written as $\mathbf{A} = \mathbf{V}\Lambda\mathbf{V}^{-1}$, with $\mathbf{V}$ invertible and Λ diagonal, is called *diagonalizable*, and the factorization itself is its *eigendecomposition*. Such a factorization exists if and only if $\mathbf{A}$ has n linearly independent eigenvectors to fill the columns of $\mathbf{V}$.

Taking powers of a diagonalizable matrix now becomes effortless. In a product of k copies of $\mathbf{V}\Lambda\mathbf{V}^{-1}$, every adjacent pair $\mathbf{V}^{-1}\mathbf{V}$ cancels out:

$$\mathbf{A}^k = \overbrace{(\mathbf{V}\Lambda\mathbf{V}^{-1})\cdots(\mathbf{V}\Lambda\mathbf{V}^{-1})}^{k \text{ times}} = \mathbf{V}\overbrace{\Lambda\cdots\Lambda}^{k \text{ times}}\mathbf{V}^{-1} = \mathbf{V}\Lambda^k\mathbf{V}^{-1}.$$

Powers of a diagonalizable matrix thus reduce to powers of scalars. Applying $\mathbf{A}^k$ to a vector $\mathbf{x}$, and writing $\mathbf{c} = \mathbf{V}^{-1}\mathbf{x}$ for the coordinates of $\mathbf{x}$ in the eigenbasis (so that $\mathbf{x} = \sum_i c_i \mathbf{v}_i$), we obtain

$$\mathbf{A}^k\mathbf{x} = \mathbf{V}\Lambda^k\mathbf{c} = \sum_{i=1}^{n} c_i \lambda_i^k \mathbf{v}_i. \tag{33.28}$$

Each eigen-coordinate evolves independently as the geometric sequence $c_i\lambda_i^k$. For the surfer starting on page 1, $\mathbf{c} = \mathbf{V}^{-1}\mathbf{p}_0 = \frac{1}{3}(1,2) = (\frac{1}{3}, \frac{2}{3})$, which gives the closed-form distribution after any number of clicks k:

$$\mathbf{p}_k = \mathbf{P}^k\mathbf{p}_0 = \frac{1}{3}\begin{pmatrix}1\\2\end{pmatrix} + \frac{2}{3}\left(\frac{1}{4}\right)^k\begin{pmatrix}1\\-1\end{pmatrix}. \tag{33.29}$$

This completely settles the opening question. As k grows, the second term vanishes and the distribution approaches $\pi = (1/3, 2/3)$: in the long run, the surfer spends twice as much time on page 2 as on page 1. The remaining error is $\frac{2}{3}\,4^{-k}$ in each entry and always points along the eigenvector $(1,-1)$. Every click divides that error by 4, so after 10 clicks it has dropped to 6×10^{-7}, and after 50 clicks it is below 10^{-30}. As a sanity check, setting $k = 3$ gives $(\frac{1}{3} + \frac{1}{96},\ \frac{2}{3} - \frac{1}{96}) = (0.34375, 0.65625)$, matching the third iterate we computed at the start. Figure 33.17 illustrates the geometry: because the entries of every $\mathbf{p}_k$ sum to one, the iterates slide along the line $x + y = 1$ toward the point where the eigenvector of $\lambda = 1$ intersects it.

Let us verify that fifty matrix multiplications and the two-term eigen-expansion produce the same $\mathbf{p}_{50}$, and check the ratio of consecutive errors.

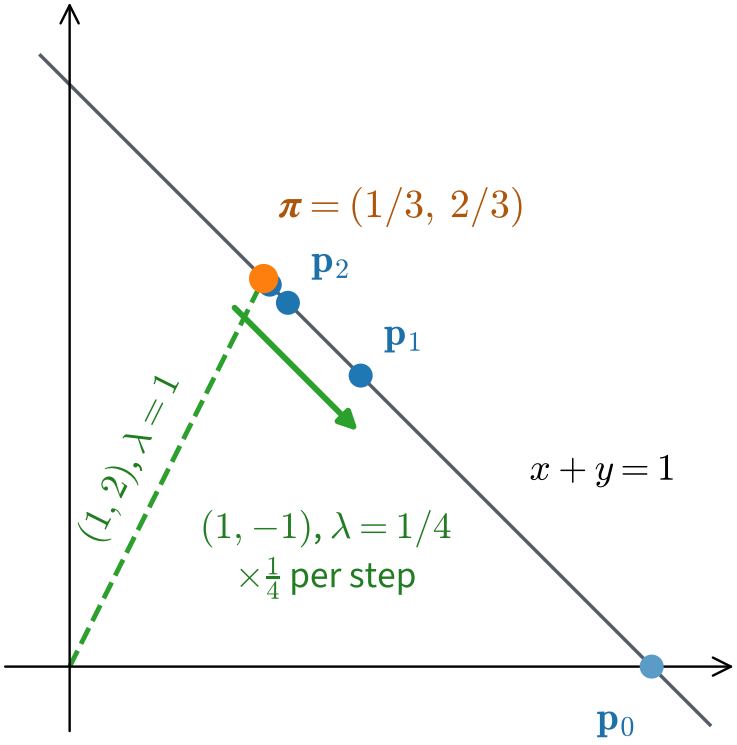


Figure 33.17: The surfer's distributions $\mathbf{p}_k$ slide along $x + y = 1$ toward $\pi = (1/3, 2/3)$, where the eigenvector $(1, 2)$ of $\lambda = 1$ meets the line. The error points along the eigenvector $(1, -1)$ of $\lambda = 1/4$ and shrinks by a factor of 4 per click.

```
p0 = np.array([1.0, 0.0])
c = np.linalg.solve(V, p0)              # coordinates of p0 in the eigenbasis
print('P^50 p0             :', np.linalg.matrix_power(P, 50) @ p0)
print('sum c_i lam_i^50 v_i:', V @ (c * lam ** 50))
pi = np.array([1 / 3, 2 / 3])
errs = np.array([np.abs(np.linalg.matrix_power(P, k) @ p0 - pi).max()
                 for k in range(8)])
print('error ratios:', np.round(errs[1:] / errs[:-1], 4))
```

```
P^50 p0             : [0.33333333 0.66666667]
sum c_i lam_i^50 v_i: [0.33333333 0.66666667]
error ratios: [0.25 0.25 0.25 0.25 0.25 0.25 0.25]
```

The two computations agree, and every consecutive error ratio is 0.25. The eigenvalue 1 governs the part of $\mathbf{p}_0$ that survives, while the eigenvalue $1/4$ governs the part that dies out.

The same recipe applies to other functions of a diagonalizable matrix. When no eigenvalue is zero, $\mathbf{A}^{-1} = \mathbf{V}\Lambda^{-1}\mathbf{V}^{-1}$: simply invert each eigenvalue on the diagonal. Similarly, the matrix exponential $e^{\mathbf{A}t} = \mathbf{V}e^{\Lambda t}\mathbf{V}^{-1}$, which solves the linear differential equation $\dot{\mathbf{x}} = \mathbf{A}\mathbf{x}$, is built by exponentiating each eigenvalue; Section 38.1 develops it in detail.

Similarity. The eigendecomposition tells us that $\mathbf{P}$ and Λ represent the very same linear map written in two different bases. In general, two matrices related by $\mathbf{A}' = \mathbf{B}\mathbf{A}\mathbf{B}^{-1}$ for some invertible $\mathbf{B}$ are called *similar*. The next proposition confirms that changing basis cannot alter a matrix's eigenvalues. It relies on the multiplicativity of the determinant from Section 33.2,

$$\det(\mathbf{X}\mathbf{Y}) = \det\mathbf{X}\,\det\mathbf{Y},$$

which in particular gives $\det(\mathbf{B}^{-1}) = 1/\det\mathbf{B}$ from $\mathbf{B}\mathbf{B}^{-1} = \mathbf{I}$.

Proposition (similar matrices share the characteristic polynomial). *For every square* $\mathbf{A}$ *and every invertible* $\mathbf{B}$*,*

$$\det(\mathbf{B}\mathbf{A}\mathbf{B}^{-1} - \lambda\mathbf{I}) = \det(\mathbf{A} - \lambda\mathbf{I}). \tag{33.30}$$

Proof. Rewrite $\lambda\mathbf{I} = \mathbf{B}(\lambda\mathbf{I})\mathbf{B}^{-1}$ and factor out $\mathbf{B}$ and $\mathbf{B}^{-1}$: $\det(\mathbf{B}\mathbf{A}\mathbf{B}^{-1} - \lambda\mathbf{I}) = \det\big(\mathbf{B}(\mathbf{A} - \lambda\mathbf{I})\mathbf{B}^{-1}\big) = \det\mathbf{B}\,\det(\mathbf{A} - \lambda\mathbf{I})\,\det\mathbf{B}^{-1} = \det(\mathbf{A} - \lambda\mathbf{I})$. ∎

Similar matrices therefore have the same eigenvalues with the same multiplicities as roots, and any quantity determined by the characteristic polynomial is a *similarity invariant*. The two most useful invariants are the determinant and the trace. For any diagonalizable matrix,

$$\det(\mathbf{A}) = \det(\mathbf{V}\Lambda\mathbf{V}^{-1}) = \det(\mathbf{V})\det(\Lambda)\det(\mathbf{V}^{-1}) = \det(\Lambda) = \lambda_1\lambda_2\cdots\lambda_n. \tag{33.31}$$

Thus the determinant is the product of the eigenvalues: whatever volume distortion $\mathbf{V}$ introduces, $\mathbf{V}^{-1}$ undoes, leaving only the axis scaling of Λ. For the trace, we use its cyclic property (Section 33.2.1.4),

$$\operatorname{tr}(\mathbf{X}\mathbf{Y}) = \operatorname{tr}(\mathbf{Y}\mathbf{X}),$$

which holds for any $\mathbf{X}$ and $\mathbf{Y}$ whose products are square. Setting $\mathbf{X} = \mathbf{V}$ and $\mathbf{Y} = \Lambda\mathbf{V}^{-1}$ gives

$$\operatorname{tr}(\mathbf{A}) = \operatorname{tr}(\mathbf{V}\Lambda\mathbf{V}^{-1}) = \operatorname{tr}(\Lambda\mathbf{V}^{-1}\mathbf{V}) = \operatorname{tr}(\Lambda) = \lambda_1 + \cdots + \lambda_n. \tag{33.32}$$

Thus the trace is the sum of the eigenvalues. On the surfer's matrix we can check both identities at a glance: $\operatorname{tr}\mathbf{P} = \frac{1}{2} + \frac{3}{4} = \frac{5}{4} = 1 + \frac{1}{4}$ and $\det\mathbf{P} = \frac{3}{8} - \frac{1}{8} = \frac{1}{4} = 1 \cdot \frac{1}{4}$. For any 2×2 matrix, these two numbers fix both eigenvalues directly as the roots of $\lambda^2 - (\operatorname{tr})\lambda + \det$.

In fact, both identities hold for *every* square matrix, diagonalizable or not, provided we count the eigenvalues over $\mathbb{C}$ with their multiplicities as roots. By the *fundamental theorem of algebra* (which we take as given), every degree-n polynomial factors completely over $\mathbb{C}$:

$$\det(\mathbf{A} - \lambda\mathbf{I}) = \prod_{i=1}^{n} (\lambda_i - \lambda).$$

Setting $\lambda = 0$ immediately yields $\det\mathbf{A} = \prod_i \lambda_i$. For the trace, compare the coefficients of λ^{n-1} on both sides. On the right-hand side the coefficient is $(-1)^{n-1}\sum_i \lambda_i$. On the left-hand side, only the product of the diagonal entries of $\mathbf{A} - \lambda\mathbf{I}$ reaches degree $n-1$, giving $(-1)^{n-1}\sum_i a_{ii}$. Hence $\operatorname{tr}\mathbf{A} = \sum_i \lambda_i$. Because any non-real eigenvalues of a real matrix come in complex conjugate pairs, both the sum and the product remain real.

Finally, the rank of a diagonalizable matrix equals its number of nonzero eigenvalues, counted with multiplicity: multiplying by the invertible matrices $\mathbf{V}$ and $\mathbf{V}^{-1}$ preserves dimensions, so $\operatorname{rank}\mathbf{A} = \operatorname{rank}\Lambda$. Exercise 2 examines what happens when a matrix is not diagonalizable.

33.3.1.4 Algebraic and Geometric Multiplicity

The surfer's matrix had two distinct eigenvalues and yielded two linearly independent eigenvectors. That was no coincidence: distinct eigenvalues always produce independent eigenvectors.

Proposition (distinct eigenvalues give independent eigenvectors). *Eigenvectors belonging to distinct eigenvalues are linearly independent. In particular, any $n \times n$ matrix with n distinct eigenvalues is diagonalizable.*

Proof. Suppose, for the sake of contradiction, that some eigenvectors $\mathbf{v}_1, \dots, \mathbf{v}_m$ with distinct eigenvalues $\lambda_1, \dots, \lambda_m$ are linearly dependent. Among all nontrivial dependence relations, choose one with the fewest nonzero coefficients, $\sum_{i=1}^{m} c_i\mathbf{v}_i = \mathbf{0}$, and relabel if needed so that $c_m \neq 0$. Such a relation must have at least two nonzero coefficients, since $c_m\mathbf{v}_m = \mathbf{0}$ alone would force $\mathbf{v}_m = \mathbf{0}$. Now apply $\mathbf{A}$ to the relation and subtract λ_m times the original relation:

$$\mathbf{0} = \mathbf{A}\sum_i c_i\mathbf{v}_i - \lambda_m\sum_i c_i\mathbf{v}_i = \sum_{i=1}^{m-1} c_i(\lambda_i - \lambda_m)\,\mathbf{v}_i.$$

The $\mathbf{v}_m$ term drops out, leaving a shorter relation. Yet it is still nontrivial: any other nonzero coefficient c_i ($i < m$) survives as $c_i(\lambda_i - \lambda_m) \neq 0$ because the eigenvalues are distinct. This contradicts the minimality of our chosen relation. When $\mathbf{A}$ has n distinct eigenvalues, its n

independent eigenvectors form a basis of $\mathbb{C}^n$ (or of $\mathbb{R}^n$ when all eigenvalues are real), so $\mathbf{V}$ is invertible. ■

When eigenvalues repeat, we have to keep track of two different counts. The number of times λ appears as a root of the characteristic polynomial is its *algebraic multiplicity* $a(\lambda)$. The dimension of its eigenspace—the number of linearly independent eigenvectors it contributes—is its *geometric multiplicity* $g(\lambda)$.

Proposition. *For every eigenvalue λ of a square matrix, $1 \le g(\lambda) \le a(\lambda)$.*

Proof. Because $\mathbf{A} - \lambda\mathbf{I}$ is singular, its null space is nontrivial, which gives $g \ge 1$. For the upper bound, pick a basis $\mathbf{v}_1, \dots, \mathbf{v}_g$ of the eigenspace, extend it to a basis of $\mathbb{C}^n$, and place these basis vectors as the columns of $\mathbf{B}$. The first g columns of $\mathbf{AB}$ are simply $\lambda\mathbf{v}_1, \dots, \lambda\mathbf{v}_g$, so in this new basis the matrix takes a block upper-triangular form:

$$\mathbf{B}^{-1}\mathbf{A}\mathbf{B} = \begin{pmatrix} \lambda\mathbf{I}_g & \mathbf{X} \\ \mathbf{0} & \mathbf{Y} \end{pmatrix}.$$

By Equation 33.30, this matrix has the same characteristic polynomial p as $\mathbf{A}$. Let us write the polynomial variable as t, since λ now stands for our fixed eigenvalue. Subtracting $t\mathbf{I}$ preserves the block upper-triangular structure, with diagonal blocks $(\lambda - t)\mathbf{I}_g$ and $\mathbf{Y} - t\mathbf{I}$. Because the determinant of a block upper-triangular matrix is the product of the determinants of its diagonal blocks (Section 33.2.3.2), we get $p(t) = (\lambda - t)^g \det(\mathbf{Y} - t\mathbf{I})$. Thus $(\lambda - t)^g$ divides the characteristic polynomial, which means $a \ge g$. ■

An eigenvalue can never supply more independent eigenvectors than its multiplicity as a root, though it can supply fewer. Now pick a basis for each eigenspace and pool all the chosen vectors together. They are jointly independent: if a linear combination of them equals zero, group the terms by eigenvalue; each group is either zero or an eigenvector, and the proposition on distinct eigenvalues forces every group to vanish. A matrix therefore possesses at most $\sum_\lambda g(\lambda)$ linearly independent eigenvectors. Over $\mathbb{C}$, the algebraic multiplicities sum to n and $g(\lambda) \le a(\lambda)$ for every λ, which gives a crisp criterion: *the eigenvectors of $\mathbf{A}$ span $\mathbb{C}^n$ if and only if $g(\lambda) = a(\lambda)$ for every complex eigenvalue λ.*

For a real matrix to have a basis of *real* eigenvectors, all of its eigenvalues must also be real. The 90° rotation $\mathbf{R}_{90} = \left(\begin{smallmatrix} 0 & -1 \\ 1 & 0 \end{smallmatrix}\right)$ shows why that extra requirement is essential: its eigenvalues $\pm i$ each satisfy $g = a = 1$, so it has a full complex eigenbasis, yet no real eigenvector at all.

When eigenvalues repeat, diagonalizability can fail altogether. The simplest counterexample is the *shear* from Section 33.2.1.2:

$$\mathbf{T} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}. \tag{33.33}$$

Its characteristic polynomial is $(1 - \lambda)^2$, so $\lambda = 1$ has algebraic multiplicity 2. Yet $(\mathbf{T} - \mathbf{I})\mathbf{v} = \mathbf{0}$ becomes $\left(\begin{smallmatrix} 0 & 1 \\ 0 & 0 \end{smallmatrix}\right)\mathbf{v} = \mathbf{0}$, which forces $v_2 = 0$. The eigenspace is only the line spanned by $(1, 0)$, so the geometric multiplicity is 1. There is no eigenbasis, and the shear is not diagonalizable. Any matrix with $g(\lambda) < a(\lambda)$ for some eigenvalue is called *defective*. Figure 33.18 makes the shortage visible: the shear slides every horizontal line sideways in proportion to its height, leaving the horizontal axis as the sole line through the origin that

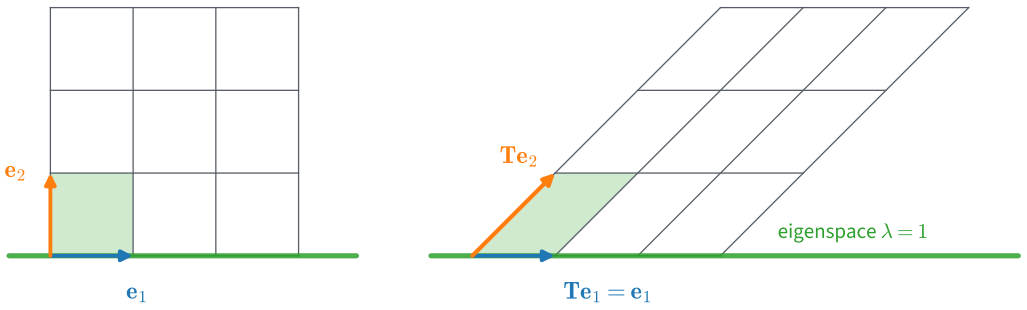


Figure 33.18: The defective shear acting on the plane. Every horizontal line slides to the right by an amount proportional to its height, tilting the grid, while the x-axis (green) maps to itself: $\mathbf{T}\mathbf{e}_1 = \mathbf{e}_1$ is the only eigendirection, with $\lambda = 1$. The other basis vector picks up a horizontal component, $\mathbf{T}\mathbf{e}_2 = (1, 1)$, so no second direction is preserved: geometric multiplicity 1 against algebraic multiplicity 2.

keeps its direction. (By contrast, a diagonalizable 2×2 matrix with a repeated eigenvalue λ would have to be $\lambda\mathbf{I}$, which preserves every direction.)

Iterating the shear shows how defectiveness changes the behavior of matrix powers: $\mathbf{T}^k = \left(\begin{smallmatrix} 1 & k \\ 0 & 1 \end{smallmatrix}\right)$, so $\mathbf{T}^k\mathbf{e}_2 = (k, 1)$ grows without bound even though the only eigenvalue is 1. No pure eigen-expansion can produce that linear factor k. The next subsection explains where it comes from.

33.3.1.5 The Jordan Normal Form

When geometric multiplicity falls short, no eigenbasis exists. What structure takes its place? For what follows in this book, one consequence matters most: a defective eigenvalue multiplies λ^k by a polynomial in k. Over the complex numbers, every square matrix is similar to a block-diagonal matrix built from *Jordan blocks*

$$\mathbf{J}_m(\lambda) = \begin{pmatrix} \lambda & 1 & & \\ & \lambda & \ddots & \\ & & \ddots & 1 \\ & & & \lambda \end{pmatrix},$$

which are $m \times m$ blocks carrying a single eigenvalue λ down the diagonal and 1s on the superdiagonal immediately above it. This factorization is the *Jordan normal form* (Horn and Johnson 2012), which we use without proof. By Equation 33.30, it has the same eigenvalues as the original matrix.

Each Jordan block supplies exactly one eigenvector, namely its first basis vector. The geometric multiplicity of λ is therefore the number of blocks carrying λ, while its algebraic multiplicity is the sum of their sizes. In particular, a matrix is diagonalizable if and only if every Jordan block has size 1×1. The shear $\mathbf{T}$ of Equation 33.33 is the single block $\mathbf{J}_2(1)$.

Under repeated application, a Jordan block does more than simply scale. Write $\mathbf{J}_m(\lambda) = \lambda\mathbf{I} + \mathbf{N}$, where $\mathbf{N}$ has 1s just above the diagonal and zeros everywhere else. Geometrically, $\mathbf{N}$ acts as a shift, sending each basis vector to the previous one, so $\mathbf{N}^m = \mathbf{0}$ (any matrix with

a vanishing power is called *nilpotent*). Because $\mathbf{I}$ and $\mathbf{N}$ commute, we can expand powers with the binomial theorem. For $m = 2$, every term beyond the first two contains $\mathbf{N}^2 = \mathbf{0}$, leaving

$$\mathbf{J}_2(\lambda)^k = (\lambda\mathbf{I} + \mathbf{N})^k = \lambda^k\mathbf{I} + k\lambda^{k-1}\mathbf{N}.$$

In the long run, the eigenvalue still decides between growth and decay, but it is now multiplied by a polynomial in k (of degree $m-1$ for an $m \times m$ block). When $|\lambda| = 1$, as in the shear, the iterates drift linearly. When $|\lambda|$ is slightly below 1, something more surprising happens: the upper-right entry $k\lambda^{k-1}$ initially grows, peaks near $k \approx -1/\log|\lambda|$, and only then decays toward zero. For $\lambda = 0.99$ this puts the peak near $k \approx 100$, where the entry reaches $100 \cdot 0.99^{99} \approx 37$. Let us check this prediction numerically.

```
J = np.array([[0.99, 1.], [0., 0.99]])   # the Jordan block J_2(0.99)
for k in [1, 50, 100, 200, 800]:
    Jk = np.linalg.matrix_power(J, k)
    pred = k * 0.99 ** (k - 1)   # predicted corner entry, k * lambda^(k-1)
    print(f'k = {k:3d}   ||J^k|| = {np.linalg.norm(Jk, 2):8.3f}   '
          f'k * 0.99^(k-1) = {pred:8.3f}')
```

```
k =   1   ||J^k|| =    1.609   k * 0.99^(k-1) =    1.000
k =  50   ||J^k|| =   30.568   k * 0.99^(k-1) =   30.556
k = 100   ||J^k|| =   36.977   k * 0.99^(k-1) =   36.973
k = 200   ||J^k|| =   27.067   k * 0.99^(k-1) =   27.067
k = 800   ||J^k|| =    0.260   k * 0.99^(k-1) =    0.260
```

Every eigenvalue of $\mathbf{J}$ is 0.99, yet the norm of its powers climbs to about 37 at $k = 100$ and is still 27 at $k = 200$ before the exponential factor 0.99^k finally takes over. Eigenvalues alone do not warn us about this transient growth, a phenomenon we revisit in Section 33.4.2.3.

Jordan blocks represent one way a real matrix can lack a real eigenbasis: it simply has too few eigenvectors. The second way is quite different in nature. A real matrix can lack any real eigenvector in a plane because it *rotates* that plane.

33.3.1.6 Complex Eigenvalues

Consider the planar rotation by an angle θ (Section 33.2.1.3),

$$\mathbf{R}_\theta = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}.$$

Whenever θ is not a multiple of π, $\mathbf{R}_\theta$ turns every nonzero real vector, leaving no real eigenvector. Its characteristic polynomial $\lambda^2 - 2\cos\theta\,\lambda + 1$ has the complex conjugate roots $\lambda = \cos\theta \pm i\sin\theta = e^{\pm i\theta}$. Here the modulus $|\lambda| = 1$ reflects that $\mathbf{R}_\theta$ preserves lengths, while the argument $\pm\theta$ records the rotation angle.

Now look at the chapter's running 2×2 matrix, which shows no obvious rotation in its entries:

$$\mathbf{A} = \begin{pmatrix} 1 & 2 \\ -1 & 3 \end{pmatrix}, \qquad \operatorname{tr}\mathbf{A} = 4, \quad \det\mathbf{A} = 5.$$

Its characteristic polynomial is $\lambda^2 - 4\lambda + 5$, whose roots are $\lambda = 2 \pm i = \sqrt{5}\, e^{\pm i\theta}$ with $\theta = \arctan \frac{1}{2} \approx 26.6^\circ$. To find an eigenvector for $2+i$, read off the second row of $\mathbf{A} - (2+i)\mathbf{I}$, which gives $-w_1 + (1-i)w_2 = 0$. Setting $w_2 = 1$ yields

$$\mathbf{w} = \begin{pmatrix} 1-i \\ 1 \end{pmatrix} = \underbrace{\begin{pmatrix} 1 \\ 1 \end{pmatrix}}_{\mathbf{u}} + i \underbrace{\begin{pmatrix} -1 \\ 0 \end{pmatrix}}_{\mathbf{v}}.$$

Where is the rotation hiding? Take any real matrix $\mathbf{A}$ with a complex eigenpair $\mathbf{A}\mathbf{w} = \lambda\mathbf{w}$, where $\lambda = a + ib$ and $b \neq 0$, and separate the eigenvector into its real and imaginary parts, $\mathbf{w} = \mathbf{u} + i\mathbf{v}$. Expanding $\mathbf{A}(\mathbf{u} + i\mathbf{v}) = (a+ib)(\mathbf{u} + i\mathbf{v})$ and equating real and imaginary parts gives

$$\mathbf{A}\mathbf{u} = a\mathbf{u} - b\mathbf{v}, \qquad \mathbf{A}\mathbf{v} = b\mathbf{u} + a\mathbf{v}.$$

Both images stay inside $\operatorname{span}\{\mathbf{u}, \mathbf{v}\}$, so this subspace is invariant under $\mathbf{A}$. Moreover, it is a genuine two-dimensional plane. If $\mathbf{u}$ and $\mathbf{v}$ were linearly dependent, their span would be a single line (not zero, because $\mathbf{w} \neq \mathbf{0}$) mapped into itself, on which $\mathbf{A}$ would act by multiplying by some real scalar μ. That would force $\mathbf{A}\mathbf{w} = \mu\mathbf{w}$, making $\lambda = \mu$ real and contradicting $b \neq 0$. Placing $\mathbf{v}$ and $\mathbf{u}$ as the columns of $\mathbf{B} = \begin{bmatrix} \mathbf{v} & \mathbf{u} \end{bmatrix}$ turns the two identities into $\mathbf{A}\mathbf{B} = \mathbf{B} \left(\begin{smallmatrix} a & -b \\ b & a \end{smallmatrix}\right)$, or equivalently,

$$\mathbf{B}^{-1}\mathbf{A}\mathbf{B} = \begin{pmatrix} a & -b \\ b & a \end{pmatrix} = \underbrace{\sqrt{a^2+b^2}}_{r} \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}, \qquad \theta = \arg(a+ib). \tag{33.34}$$

In the basis $(\mathbf{v}, \mathbf{u})$ of its invariant plane, $\mathbf{A}$ acts as a rotation–scaling block $r\mathbf{R}_\theta$: it scales by the modulus r and rotates by the argument θ. Because the derivation used only $\mathbf{A}\mathbf{w} = \lambda\mathbf{w}$, this block is independent of how $\mathbf{w}$ is scaled or rotated in phase. The connection also works in reverse: if we identify $(x, y) \in \mathbb{R}^2$ with the complex number $x + iy$, then the block maps (x, y) to $(ax - by,\ bx + ay)$, which is simply complex multiplication by $a + ib$.

For our running matrix $\mathbf{A}$, we have $\mathbf{B} = \left(\begin{smallmatrix} -1 & 1 \\ 0 & 1 \end{smallmatrix}\right)$ and $\mathbf{B}^{-1}\mathbf{A}\mathbf{B} = \left(\begin{smallmatrix} 2 & -1 \\ 1 & 2 \end{smallmatrix}\right)$, which scales by $\sqrt{5}$ and rotates by 26.6°. One might expect every step to lengthen a vector by the factor $\sqrt{5} \approx 2.24$. Let us compute the block from the eigenvector returned by `eig` and track the actual lengths of the iterates $\mathbf{A}^k(1, 0)$.

```
A = np.array([[1., 2.], [-1., 3.]])
lam, V = np.linalg.eig(A)
j = np.argmax(lam.imag)                  # the eigenvalue a + ib with b > 0
r, theta = np.abs(lam[j]), np.angle(lam[j])
u, v = V[:, j].real, V[:, j].imag
B = np.column_stack([v, u])              # basis (v, u) of the invariant plane
print(f'lambda = {lam[j]:.4f}   r = {r:.4f}   theta = {theta:.4f}')
print('B^-1 A B =\n', np.round(np.linalg.inv(B) @ A @ B, 4))
xs = [np.array([1., 0.])]
for _ in range(8):
    xs.append(A @ xs[-1])
norms = np.linalg.norm(xs, axis=1)
```

```
    print('norm ratios:', np.round(norms[1:] / norms[:-1], 2))
    x20 = np.linalg.matrix_power(A, 20) @ np.array([1., 0.])
    print(f'average rate ||A^20 x||^(1/20) = {np.linalg.norm(x20) ** (1 / 20):.3f}')
```

```
lambda = 2.0000+1.0000j   r = 2.2361   theta = 0.4636
B^-1 A B =
 [[ 2. -1.]
 [ 1.  2.]]
norm ratios: [1.41 2.92 3.45 2.76 2.27 1.88 1.5  1.54]
average rate ||A^20 x||^(1/20) = 2.252
```

Regardless of which eigenvector `eig` chooses, the resulting block is $\left(\begin{smallmatrix} 2 & -1 \\ 1 & 2 \end{smallmatrix}\right)$. Yet the vector lengths do *not* grow by a steady factor of 2.24 at each step. Along the trajectory $(1,0) \to (1,-1) \to (-1,-4) \to (-9,-11) \to (-31,-24)$, the per-step norm ratios swing between roughly 1.4 and 3.5. The reason is that the scaling is uniform only in the coordinates $(\mathbf{v}, \mathbf{u})$, which are not orthonormal. Back in standard coordinates, the iterates spiral outward along stretched ellipses, completing a full turn roughly every $2\pi/\theta \approx 13.6$ steps. Only the long-run geometric average $\|\mathbf{A}^k\mathbf{x}\|^{1/k}$ settles at $\sqrt{5}$ (reaching 2.25 by $k = 20$). The motion is a rigid rotation and uniform scaling in standard coordinates only when the matrix is *normal*, a property we define in Section 33.4.2.2.

Up to a real change of basis, every real square matrix is built from the three elementary behaviors we have now uncovered, alone or in combination: stretching along a line, shearing within a Jordan block, and rotating with scaling in a plane. This decomposition is known as the *real Jordan form* (Horn and Johnson 2012).

33.3.2 Symmetric Matrices and Quadratic Forms

33.3.2.1 The Spectral Theorem

Which matrices guarantee a real, orthonormal eigenbasis? Symmetry eliminates both obstacles we just encountered: it rules out complex eigenvalues as well as shortages of eigenvectors. The symmetric matrix $\mathbf{S}$ from Section 1.3 shows what happens. Its unit eigenvectors $(1,1)/\sqrt{2}$ and $(1,-1)/\sqrt{2}$ are perpendicular, giving

$$\mathbf{S} = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix} = \underbrace{\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}}_{\mathbf{V}} \begin{pmatrix} 3 & 0 \\ 0 & 1 \end{pmatrix} \underbrace{\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}}_{\mathbf{V}^\top}.$$

Because the eigenvector matrix $\mathbf{V}$ is orthogonal ($\mathbf{V}^\top\mathbf{V} = \mathbf{I}$), its inverse is simply its transpose. The spectral theorem guarantees that this holds for *every* real symmetric matrix. It is the central theorem of this chapter: positive definiteness, the Rayleigh quotient, and all of Section 33.5 rest on it.

Theorem (spectral theorem). *Every real symmetric matrix $\mathbf{A}$ has real eigenvalues and an orthonormal basis of real eigenvectors. Equivalently,*

$$\mathbf{A} = \mathbf{V}\Lambda\mathbf{V}^\top, \tag{33.35}$$

where $\Lambda = \operatorname{diag}(\lambda_1, \dots, \lambda_n)$ is real and $\mathbf{V}$ is orthogonal.

We prove this in three steps: first, every eigenvalue is real; second, eigenvectors for distinct eigenvalues are orthogonal; third, a full orthonormal eigenbasis always exists. For complex vectors we use the Hermitian inner product $\langle \mathbf{x}, \mathbf{y} \rangle = \mathbf{x}^* \mathbf{y}$, where $\mathbf{x}^*$ is the conjugate transpose (which reduces to the ordinary dot product on real vectors). The only external fact we borrow is that the characteristic polynomial has at least one complex root (the fundamental theorem of algebra), ensuring that at least one eigenpair exists over $\mathbb{C}$. Parts (i) and (ii) take only two lines each; the real work is in part (iii).

Proof, part (i): the eigenvalues are real. Suppose $\mathbf{A}\mathbf{v} = \lambda \mathbf{v}$ with $\mathbf{v} \neq \mathbf{0}$, allowing complex entries for the moment. Because $\mathbf{A}$ is real and symmetric, $\mathbf{A}^* = \mathbf{A}^\top = \mathbf{A}$, and therefore

$$\lambda \, \mathbf{v}^* \mathbf{v} = \mathbf{v}^* \mathbf{A} \mathbf{v} = (\mathbf{A}\mathbf{v})^* \mathbf{v} = \bar{\lambda} \, \mathbf{v}^* \mathbf{v}.$$

Since $\mathbf{v}^* \mathbf{v} = \|\mathbf{v}\|^2 > 0$, we can divide both sides by it to obtain $\lambda = \bar{\lambda}$: the eigenvalue must be real. Its eigenvector can then be chosen real as well, because it solves the real homogeneous system $(\mathbf{A} - \lambda \mathbf{I})\mathbf{v} = \mathbf{0}$. ■

Proof, part (ii): eigenvectors of distinct eigenvalues are orthogonal. Let $\mathbf{A}\mathbf{u} = \lambda \mathbf{u}$ and $\mathbf{A}\mathbf{v} = \mu \mathbf{v}$ with $\lambda \neq \mu$, where both eigenvalues and both eigenvectors are real by part (i). Using $\mathbf{A} = \mathbf{A}^\top$ to move $\mathbf{A}$ across the inner product,

$$\lambda \langle \mathbf{u}, \mathbf{v} \rangle = \langle \mathbf{A}\mathbf{u}, \mathbf{v} \rangle = \langle \mathbf{u}, \mathbf{A}\mathbf{v} \rangle = \mu \langle \mathbf{u}, \mathbf{v} \rangle.$$

Thus $(\lambda - \mu)\langle \mathbf{u}, \mathbf{v} \rangle = 0$. Because $\lambda \neq \mu$, we must have $\langle \mathbf{u}, \mathbf{v} \rangle = 0$. ■

Proof, part (iii): an orthonormal eigenbasis. We argue by induction on n; for $n = 1$ the claim is immediate. For $n > 1$, part (i) supplies a real eigenpair $(\lambda_1, \mathbf{v}_1)$ with $\|\mathbf{v}_1\| = 1$. Consider the $(n-1)$-dimensional orthogonal complement $U = \mathbf{v}_1^\perp$. Symmetry ensures that $\mathbf{A}$ maps U back into itself: for any $\mathbf{x} \in U$,

$$\langle \mathbf{A}\mathbf{x}, \mathbf{v}_1 \rangle = \langle \mathbf{x}, \mathbf{A}\mathbf{v}_1 \rangle = \lambda_1 \langle \mathbf{x}, \mathbf{v}_1 \rangle = 0.$$

Choose an orthonormal basis of U and assemble its vectors as the columns of $\mathbf{Q} \in \mathbb{R}^{n \times (n-1)}$. Because every column of $\mathbf{AQ}$ lies in U, we can write $\mathbf{AQ} = \mathbf{QC}$ where $\mathbf{C} = \mathbf{Q}^\top \mathbf{A} \mathbf{Q}$ represents $\mathbf{A}$ restricted to U. Notice that $\mathbf{C}$ is symmetric, $\mathbf{C}^\top = \mathbf{Q}^\top \mathbf{A}^\top \mathbf{Q} = \mathbf{C}$, and has size $(n-1) \times (n-1)$. By the inductive hypothesis, $\mathbf{C} = \mathbf{V}' \Lambda' \mathbf{V}'^\top$ for an orthogonal matrix $\mathbf{V}'$. Now look at the columns of $\mathbf{QV}'$: they are orthonormal because $(\mathbf{QV}')^\top \mathbf{QV}' = \mathbf{V}'^\top \mathbf{V}' = \mathbf{I}$; they lie in U, so they are orthogonal to $\mathbf{v}_1$; and they are eigenvectors of $\mathbf{A}$, because $\mathbf{AQV}' = \mathbf{QCV}' = \mathbf{QV}' \Lambda'$. Appending them to $\mathbf{v}_1$ gives the orthogonal matrix $\mathbf{V} = [\mathbf{v}_1 \ \mathbf{QV}']$ satisfying $\mathbf{AV} = \mathbf{V}\Lambda$. ■

Geometrically, Equation 33.35 breaks any symmetric linear map into three clean actions. First, $\mathbf{V}^\top$ aligns the orthonormal eigenvectors with the coordinate axes—a pure rotation, possibly combined with a reflection (for $\mathbf{S}$ above, $\det \mathbf{V} = -1$). Next, Λ stretches each coordinate axis by its eigenvalue. Finally, $\mathbf{V}$ turns the axes back. This proves what the left panel of Figure 33.16 illustrated for $\mathbf{S}$, now for every symmetric matrix.

Every matrix $\mathbf{A}$, rectangular or square, produces a symmetric matrix $\mathbf{A}^\top \mathbf{A}$. Its quadratic form $\mathbf{x}^\top \mathbf{A}^\top \mathbf{A} \mathbf{x} = \|\mathbf{A}\mathbf{x}\|^2$ can never be negative—a property that we name, test, and exploit next.

33.3.2.2 Positive Definiteness

Can the quadratic form $\mathbf{x}^\top \mathbf{A}\mathbf{x}$ ever turn negative? This single question decides whether a critical point of a loss surface is a minimum, whether a matrix is a valid covariance, and whether a least-squares problem has a unique solution. A symmetric matrix $\mathbf{A}$ is *positive semidefinite* (PSD), written $\mathbf{A} \succeq 0$, if

$$\mathbf{x}^\top \mathbf{A}\mathbf{x} \geq 0 \quad \text{for all } \mathbf{x},$$

and *positive definite* (PD), written $\mathbf{A} \succ 0$, if the inequality is strict for every nonzero $\mathbf{x}$. For instance, $\mathbf{x}^\top \mathbf{S}\mathbf{x} = 2x_1^2 + 2x_1x_2 + 2x_2^2 = x_1^2 + x_2^2 + (x_1 + x_2)^2$ is strictly positive whenever $\mathbf{x} \neq \mathbf{0}$, so $\mathbf{S} \succ 0$, whereas $\left(\begin{smallmatrix} 1 & 0 \\ 0 & -1 \end{smallmatrix}\right)$ is neither PD nor PSD because $\mathbf{e}_2$ gives -1. Completing the square by hand quickly becomes impractical in higher dimensions, but the eigenvalues settle the question cleanly for any symmetric matrix.

Proposition (definiteness from eigenvalues). *For symmetric* $\mathbf{A}$*,*

$$\mathbf{A} \succeq 0 \iff \lambda_i \geq 0 \text{ for all } i, \qquad \mathbf{A} \succ 0 \iff \lambda_i > 0 \text{ for all } i.$$

Proof. Apply the spectral theorem to write $\mathbf{A} = \mathbf{V}\Lambda\mathbf{V}^\top$, and switch to eigenvector coordinates $\mathbf{y} = \mathbf{V}^\top \mathbf{x}$—an orthogonal change of basis that preserves lengths. Then

$$\mathbf{x}^\top \mathbf{A}\mathbf{x} = \mathbf{x}^\top \mathbf{V}\Lambda\mathbf{V}^\top \mathbf{x} = \mathbf{y}^\top \Lambda \mathbf{y} = \sum_{i=1}^{n} \lambda_i \, y_i^2 = \sum_{i=1}^{n} \lambda_i \, (\mathbf{v}_i^\top \mathbf{x})^2. \tag{33.36}$$

In eigenvector coordinates, the quadratic form is simply a weighted sum of squares whose weights are the eigenvalues. If every $\lambda_i \geq 0$, this sum is clearly non-negative. Conversely, plugging in $\mathbf{x} = \mathbf{v}_j$ gives $\mathbf{y} = \mathbf{e}_j$ and $\mathbf{v}_j^\top \mathbf{A}\mathbf{v}_j = \lambda_j$, forcing $\lambda_j \geq 0$. The positive definite case follows by the same argument with strict inequalities, since $\mathbf{y} \neq \mathbf{0}$ whenever $\mathbf{x} \neq \mathbf{0}$. ∎

Let us check Equation 33.36 on $\mathbf{S}$ with $\mathbf{x} = \mathbf{e}_1$: the eigen-coordinates are $\mathbf{y} = \mathbf{V}^\top \mathbf{e}_1 = (1/\sqrt{2}, 1/\sqrt{2})$, and the weighted sum gives $3 \cdot \frac{1}{2} + 1 \cdot \frac{1}{2} = 2$, matching $\mathbf{e}_1^\top \mathbf{S}\mathbf{e}_1 = s_{11}$. Three consequences account for most appearances of positive definiteness in deep learning:

- **PD implies invertible.** If $\mathbf{A} \succ 0$ and $\mathbf{A}\mathbf{x} = \mathbf{0}$, then $\mathbf{x}^\top \mathbf{A}\mathbf{x} = 0$, which forces $\mathbf{x} = \mathbf{0}$.
- **Gram and covariance matrices are PSD.** For any matrix $\mathbf{X}$, $\mathbf{x}^\top \mathbf{X}^\top \mathbf{X}\mathbf{x} = \|\mathbf{X}\mathbf{x}\|^2 \geq 0$, so $\mathbf{X}^\top \mathbf{X} \succeq 0$. It is strictly PD if and only if $\mathbf{X}$ has full column rank (so that $\mathbf{X}\mathbf{x} = \mathbf{0}$ implies $\mathbf{x} = \mathbf{0}$). Because the sample covariance of centered data is a Gram matrix divided by the number of samples, it is always PSD.
- **A PD Hessian guarantees a strict local minimum.** Suppose f is twice continuously differentiable with $\nabla f(\mathbf{x}^*) = \mathbf{0}$, and its Hessian $\nabla^2 f(\mathbf{x}^*)$ (the symmetric matrix of second partial derivatives) is PD. For any small step δ, the second-order Taylor expansion (Section 34.2.3.2) reads $f(\mathbf{x}^* + \delta) = f(\mathbf{x}^*) + \frac{1}{2}\delta^\top \nabla^2 f(\mathbf{x}^*)\delta + o(\|\delta\|^2)$. By Equation 33.36, the quadratic term is at least $\frac{1}{2}\lambda_{\min}\|\delta\|^2 > 0$, so $\mathbf{x}^*$ is a strict local minimum. Note that this condition is sufficient rather than necessary: $f(x) = x^4$ has a strict minimum at 0 even though its second derivative there is 0.

Positive definiteness also admits a fast algorithmic test, the *Cholesky factorization*: $\mathbf{A} \succ 0$ if and only if $\mathbf{A} = \mathbf{L}\mathbf{L}^\top$ for a lower-triangular matrix $\mathbf{L}$ with positive diagonal entries. One direction is immediate: such an $\mathbf{L}$ is invertible, so $\mathbf{x}^\top \mathbf{A}\mathbf{x} = \|\mathbf{L}^\top \mathbf{x}\|^2 > 0$ for every nonzero

x. Attempting a Cholesky factorization is the standard numerical way to test positive definiteness, and it is considerably cheaper than computing all eigenvalues. As a bonus, the same factor **L** transforms independent standard Gaussian samples **z** into samples **Lz** with covariance $\mathbf{L}\mathbf{L}^\top = \mathbf{A}$.

Geometrically, the signs of the eigenvalues govern the shape of the surface $z = \mathbf{x}^\top \mathbf{A}\mathbf{x}$ (Figure 33.19): a bowl when all eigenvalues are positive, a trough that is flat along any zero-eigenvalue direction, and a saddle when signs are mixed.

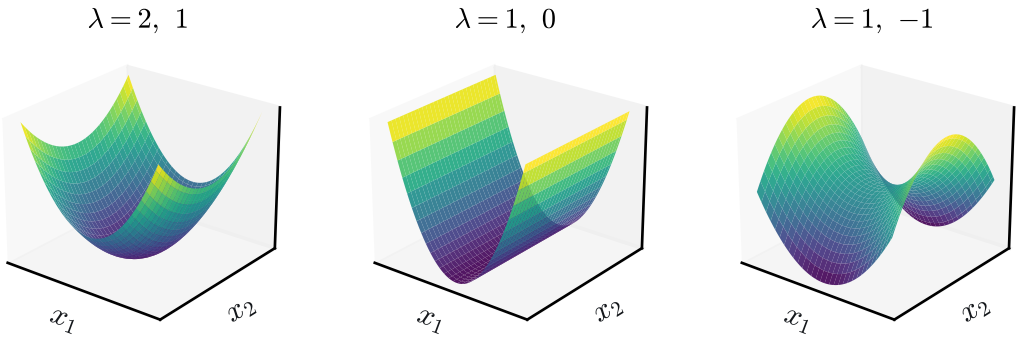


Figure 33.19: The quadratic-form surface $z = \mathbf{x}^\top \mathbf{A}\mathbf{x}$ for three diagonal matrices takes its shape from the eigenvalue signs: a bowl when all $\lambda_i > 0$ (positive definite), a trough when one $\lambda_i = 0$ (positive semidefinite), and a saddle for mixed signs (indefinite).

We can see the rank condition for Gram matrices directly in their eigenvalues. Below we compare a tall matrix **X** with independent columns, whose $\mathbf{X}^\top \mathbf{X}$ should be PD, against one whose second column is twice the first, which should yield a PSD matrix with a zero eigenvalue.

```
X_full = np.array([[1., 0.], [1., 1.], [0., 1.], [2., 1.]])  # rank 2
X_dep = np.array([[1., 2.], [1., 2.], [0., 0.], [2., 4.]])   # col 2 = 2*col 1
for X, name in [(X_full, 'full column rank'), (X_dep, 'dependent columns')]:
    eigvals = np.linalg.eigvalsh(X.T @ X).round(4)
    eigvals[np.abs(eigvals) < 1e-9] = 0.0  # clean up tiny round-off / -0.0
    print(f'{name}: eigenvalues of X^T X = {eigvals}')
```

```
full column rank: eigenvalues of X^T X = [1.1459 7.8541]
dependent columns: eigenvalues of X^T X = [ 0. 30.]
```

Both eigenvalues are strictly positive in the full-rank case. In the dependent case, one drops to zero: the quadratic form vanishes along $(2, -1)$, the very direction that **X** maps to zero.

33.3.2.3 The Rayleigh Quotient

Identity Equation 33.36 does more than test signs: it turns the largest and smallest eigenvalues into the solutions of optimization problems. For a symmetric matrix **A**, define the *Rayleigh quotient*

$$R(\mathbf{x}) = \frac{\mathbf{x}^\top \mathbf{A}\mathbf{x}}{\mathbf{x}^\top \mathbf{x}}, \qquad \mathbf{x} \neq \mathbf{0}.$$

This measures the quadratic form per unit squared length, and whenever $\mathbf{x}$ is an eigenvector, $R(\mathbf{x})$ equals its eigenvalue. Throughout this subsection we order the eigenvalues by their signed values, $\lambda_1 \geq \cdots \geq \lambda_n$, rather than by magnitude.

Proposition (Rayleigh). *For symmetric* $\mathbf{A}$ *with eigenvalues* $\lambda_1 \geq \cdots \geq \lambda_n$*,*

$$\lambda_1 = \max_{\mathbf{x} \neq \mathbf{0}} R(\mathbf{x}), \qquad \lambda_n = \min_{\mathbf{x} \neq \mathbf{0}} R(\mathbf{x}),$$

with the maximum and minimum attained at the corresponding eigenvectors.

Proof. Rescaling $\mathbf{x}$ leaves $R(\mathbf{x})$ unchanged, so we may restrict attention to unit vectors. Setting $\mathbf{y} = \mathbf{V}^\top \mathbf{x}$ preserves the norm ($\|\mathbf{y}\| = \|\mathbf{x}\| = 1$) and turns Equation 33.36 into $R(\mathbf{x}) = \sum_i \lambda_i y_i^2$. This is a weighted average of the eigenvalues with non-negative weights y_i^2 summing to 1. Every average lies between its smallest and largest values, so $\lambda_n \leq R(\mathbf{x}) \leq \lambda_1$. Putting all the weight onto a single coordinate, at $\mathbf{x} = \mathbf{v}_1$ or $\mathbf{x} = \mathbf{v}_n$, attains the two bounds. ■

For our running symmetric matrix $\mathbf{S}$, this tells us that $R(\mathbf{x})$ ranges over $[1, 3]$, peaking along the eigenvector at 45° and reaching its minimum along the eigenvector at 135°. Every intermediate direction blends the eigenvalues 1 and 3 in proportion to its squared eigencoordinates.

Maximizing over subspaces recovers all intermediate eigenvalues as well: the *Courant–Fischer theorem* states that $\lambda_j = \max_{\dim U = j} \min_{\mathbf{0} \neq \mathbf{x} \in U} R(\mathbf{x})$, the highest guaranteed Rayleigh quotient across all j-dimensional subspaces U (Exercise 4d guides you through the proof).

Two applications of the extreme eigenvalues recur throughout this book. First, for any matrix $\mathbf{A}$, the largest Rayleigh quotient of $\mathbf{A}^\top \mathbf{A}$ is $\max_{\mathbf{x} \neq \mathbf{0}} \|\mathbf{A}\mathbf{x}\|^2 / \|\mathbf{x}\|^2$, the square of the largest factor by which $\mathbf{A}$ can stretch a vector. Second, consider gradient descent on a convex quadratic $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{A}\mathbf{x}$ with $\mathbf{A} \succ 0$, whose update is $\mathbf{x} \leftarrow (\mathbf{I} - \eta \mathbf{A})\mathbf{x}$. By Equation 33.28, each eigen-coordinate is multiplied by $1 - \eta\lambda_i$ at every step, so the iteration converges from every starting point if and only if $|1 - \eta\lambda_i| < 1$ for all i—that is, $0 < \eta < 2/\lambda_{\max}$. The largest eigenvalue thus caps the step size (it is the smoothness constant L of Section 35.1), and for step sizes $\eta \leq 1/\lambda_{\max}$ the smallest eigenvalue dictates the slowest contraction rate, $1 - \eta\lambda_{\min}$. Their ratio $\lambda_{\max}/\lambda_{\min}$ is the condition number of $\mathbf{A}$, which Section 33.5.3.2 generalizes to arbitrary matrices.

33.3.2.4 Sample Covariance and the Marchenko–Pastur Law

When we look for structure in high-dimensional data, we inspect the largest eigenvalues of the sample covariance matrix. To judge which eigenvalues are truly large, however, we first need a baseline: what do the eigenvalues look like when the data contain no structure at all? Suppose we draw n samples of pure noise in d dimensions, with independent coordinates of mean 0 and variance 1 (so the true covariance is $\mathbf{I}$), and stack them as the rows of $\mathbf{X} \in \mathbb{R}^{n \times d}$. Do the eigenvalues of the sample covariance $\mathbf{C} = \frac{1}{n}\mathbf{X}^\top \mathbf{X}$ cluster near 1? For any unit vector $\mathbf{w}$, the Rayleigh quotient $\mathbf{w}^\top \mathbf{C}\mathbf{w}$ is the sample variance of the data projected onto $\mathbf{w}$. The extreme eigenvalues of $\mathbf{C}$ are therefore the sample variances along the most and least variable directions—the very quantities that principal component analysis interprets as structure (Section 33.5.2.2).

Surprisingly, they do *not* cluster near 1. Let $d, n \to \infty$ while holding the *aspect ratio* $c = d/n \leq 1$ fixed. Then the histogram of eigenvalues of $\mathbf{C}$ converges to a deterministic density supported on the interval

$$[(1-\sqrt{c})^2,\ (1+\sqrt{c})^2],$$

known as the *Marchenko–Pastur bulk* (Marchenko and Pastur 1967), and provided the noise has a finite fourth moment, the extreme eigenvalues converge right to its edges. When $c > 1$, the matrix $\mathbf{C}$ has rank at most $n < d$, so at least $d - n$ eigenvalues are identically zero in addition to the bulk. Even with four samples per dimension ($c = 1/4$), pure-noise eigenvalues spread all the way across $[0.25, 2.25]$: the sample variance along the loudest random direction is more than double the true variance, while along the quietest it is only a quarter. At $n = 4000$ and $d = 1000$, this asymptotic limit is already strikingly accurate. Let us test it.

```
n, d = 4000, 1000                       # aspect ratio c = d / n = 1/4
X = np.random.randn(n, d)               # pure noise: true covariance I
ev = np.linalg.eigvalsh(X.T @ X / n)
c = d / n
print(f'eigenvalues of C: min {ev.min():.3f}, max {ev.max():.3f}')
print(f'Marchenko-Pastur edges: {(1 - c ** 0.5) ** 2:.3f}, '
      f'{(1 + c ** 0.5) ** 2:.3f}')
```

```
eigenvalues of C: min 0.247, max 2.238
Marchenko-Pastur edges: 0.250, 2.250
```

On every run, the smallest and largest eigenvalues land within a few hundredths of the predicted edges 0.25 and 2.25. Any eigenvalue falling inside this bulk is indistinguishable from pure noise; only an eigenvalue clearly above the upper edge provides evidence of genuine structure. Exercise 5 explores how strong a signal must be before its eigenvalue escapes the bulk.

Throughout this section we took two things for granted: that we can actually compute the eigenvalues of a matrix, and that those eigenvalues tell us what its powers do after a handful of steps just as well as after many. For large matrices and for matrices far from symmetric, neither assumption is safe. Section 33.4 takes up both.

33.3.3 Summary

In eigenvector coordinates, a diagonalizable matrix acts as a list of scalars, $\mathbf{A}^k\mathbf{x} = \sum_i c_i \lambda_i^k \mathbf{v}_i$, explaining repeated application in Markov chains, deep linear networks, and gradient descent on quadratics. A full eigenbasis exists over $\mathbb{C}$ if and only if every eigenvalue's geometric multiplicity matches its algebraic multiplicity; otherwise Jordan blocks introduce polynomial factors in k that can cause transient growth, while complex conjugate pairs correspond to rotation and scaling in an invariant plane. Real symmetric matrices avoid both pitfalls: they always have real eigenvalues and an orthonormal eigenbasis. Their quadratic forms become weighted sums of squares, so eigenvalue signs determine positive definiteness and the extreme eigenvalues are the extremes of the Rayleigh quotient. Finally, the eigenvalues of a pure-noise sample covariance spread across the Marchenko–Pastur interval, meaning that only eigenvalues above the upper bulk edge signal real structure.

33.3.4 Exercises

1. **Trace, determinant, and definiteness.** The identities Equation 33.31 and Equation 33.32 fix both eigenvalues of a 2×2 matrix through their sum and product.
   1. Find the eigenvalues of $\left(\begin{smallmatrix} 4 & 1 \\ 1 & 4 \end{smallmatrix}\right)$ from its trace and determinant alone, without forming the characteristic polynomial. Then find its eigenvectors. Do the same for the surfer's matrix $\mathbf{P}$, whose trace is $5/4$ and determinant $1/4$.
   2. Show that a symmetric 2×2 matrix is positive definite if and only if its trace and its determinant are both positive. Which sign conditions on trace and determinant characterize positive semidefiniteness?
   3. Each of the matrices $\left(\begin{smallmatrix} 2 & 1 \\ 1 & 2 \end{smallmatrix}\right)$, $\left(\begin{smallmatrix} 1 & 2 \\ 2 & 1 \end{smallmatrix}\right)$, and $\left(\begin{smallmatrix} 1 & 1 \\ 1 & 1 \end{smallmatrix}\right)$ is the Hessian of the quadratic $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{A}\mathbf{x}$ at its critical point $\mathbf{0}$. Use (b) to decide for each f whether the origin is a strict minimum, a minimum that is not strict, or a saddle point. For the saddle point, give a direction along which f decreases.
   4. Show that the Cholesky factor $\mathbf{L}$ of $\left(\begin{smallmatrix} a & b \\ b & c \end{smallmatrix}\right)$ has the diagonal entries $\sqrt{a}$ and $\sqrt{(ac - b^2)/a}$. Why do the conditions under which a factor with positive diagonal exists coincide with the test of (b)?

2. **Rank and zero eigenvalues.** The section shows that the rank of a diagonalizable matrix equals the number of its nonzero eigenvalues, counted with multiplicity.
   1. Find the rank and the eigenvalues of $\mathbf{N} = \left(\begin{smallmatrix} 0 & 1 \\ 0 & 0 \end{smallmatrix}\right)$. Which step of the argument for diagonalizable matrices fails for $\mathbf{N}$?
   2. Let g_0 be the geometric multiplicity of the eigenvalue 0 of an $n \times n$ matrix $\mathbf{A}$, with $g_0 = 0$ when 0 is not an eigenvalue. Show that $\operatorname{rank} \mathbf{A} = n - g_0$ for every square matrix. Conclude that the rank is at least the number of nonzero eigenvalues counted with algebraic multiplicity, with equality if and only if the eigenvalue 0 is not defective.
   3. The *singular values* of a matrix are $\sigma_i = \sqrt{\lambda_i(\mathbf{A}^\top \mathbf{A})}$, the square roots of the eigenvalues of the PSD matrix $\mathbf{A}^\top \mathbf{A}$. Find the singular values of $\mathbf{N}$ and of the Jordan block $\mathbf{J}_m(0)$ of Section 33.3.1.5. Compare the number of nonzero singular values with the rank and with the number of nonzero eigenvalues.

3. **The centering matrix.** Let $\mathbf{1} \in \mathbb{R}^n$ be the all-ones vector and $\mathbf{H} = \mathbf{I} - \frac{1}{n}\mathbf{1}\mathbf{1}^\top$.
   1. Show that $n\mathbf{I} - \mathbf{1}\mathbf{1}^\top$ is positive semidefinite by writing $\mathbf{x}^\top(n\mathbf{I} - \mathbf{1}\mathbf{1}^\top)\mathbf{x}$ as a sum of squares of the differences $x_i - x_j$. Then find all eigenvalues of $n\mathbf{I} - \mathbf{1}\mathbf{1}^\top$ and their eigenvectors.
   2. Show that $\mathbf{H}$ is symmetric with $\mathbf{H}^2 = \mathbf{H}$, and deduce its eigenvalues and its rank from (a).
   3. Let the rows of $\mathbf{X} \in \mathbb{R}^{n \times d}$ be n samples. Subtracting the sample mean from every row gives $\mathbf{H}\mathbf{X}$, so that the sample covariance is $\mathbf{C} = \frac{1}{n}\mathbf{X}^\top \mathbf{H}\mathbf{X}$. Show that $\mathbf{C} \succeq 0$ and $\operatorname{rank} \mathbf{C} \le n - 1$. How many zero eigenvalues does the sample covariance of 512-dimensional embeddings computed from a minibatch of 256 examples have at least? This is the case $c = d/n > 1$ of Section 33.3.2.4. Which operations on $\mathbf{C}$ does it rule out?

   Adapted from Gilbert Strang, MIT 18.065 Problem Sets, Problems for Lecture 13, Problem 4.

4. code **Accuracy of computed eigenvalues.** ● The algorithm behind `np.linalg.eigvals` is backward stable: it returns the exact eigenvalues of a matrix $\mathbf{A} + \mathbf{E}$ with $\|\mathbf{E}\|_2$ of the order of $\epsilon_{\text{mach}}\|\mathbf{A}\|_2$, where $\epsilon_{\text{mach}} \approx 2.2 \times 10^{-16}$ in double precision (Golub and

Van Loan 1996). A small perturbation can move a defective eigenvalue by a fractional power of ε.

1. Let $\mathbf{e}_1, \dots, \mathbf{e}_m$ be the standard basis vectors. Show that $\mathbf{J}_m(\lambda) + \varepsilon\, \mathbf{e}_m \mathbf{e}_1^{\top}$, which places ε in the lower-left corner of the Jordan block, has the eigenvalues $\lambda + \varepsilon^{1/m}\omega$, where ω ranges over the m-th roots of unity.
2. For $m = 1, \dots, 4$, form $\mathbf{Q}\mathbf{J}_m(2)\mathbf{Q}^{\top}$ with an orthogonal $\mathbf{Q}$ taken from the QR factorization of a Gaussian matrix, and record the median over 20 draws of $\max_i |\widehat{\lambda}_i - 2|$ for the computed eigenvalues $\widehat{\lambda}_i$. Compare with $\epsilon_{\text{mach}}^{1/m}$. Repeat with the symmetric matrix $\mathbf{Q}(2\mathbf{I})\mathbf{Q}^{\top}$.
3. Let $\mathbf{A}$ and $\mathbf{E}$ be symmetric. Use the Rayleigh proposition of Section 33.3.2.3 to show that $\lambda_{\max}(\mathbf{A} + \mathbf{E}) \le \lambda_{\max}(\mathbf{A}) + \|\mathbf{E}\|_2$ and $\lambda_{\min}(\mathbf{A} + \mathbf{E}) \ge \lambda_{\min}(\mathbf{A}) - \|\mathbf{E}\|_2$. How does this explain the symmetric result of (b)?
4. Prove the Courant–Fischer theorem $\lambda_j = \max_{\dim U = j} \min_{\mathbf{0} \neq \mathbf{x} \in U} R(\mathbf{x})$ for symmetric $\mathbf{A}$ with $\lambda_1 \ge \cdots \ge \lambda_n$. For the upper bound, intersect U with $\operatorname{span}(\mathbf{v}_j, \dots, \mathbf{v}_n)$, which has dimension $n - j + 1$, so the intersection is at least a line; for the lower bound take $U = \operatorname{span}(\mathbf{v}_1, \dots, \mathbf{v}_j)$. Deduce Weyl's inequality $|\lambda_j(\mathbf{A} + \mathbf{E}) - \lambda_j(\mathbf{A})| \le \|\mathbf{E}\|_2$ for every j, which extends (c) to all eigenvalues.

5. code **Spikes above the bulk.** ● Section 33.3.2.4 treats the Marchenko–Pastur bulk as the range of pure-noise eigenvalues. Suppose that the data contain one real direction: the n samples have independent Gaussian coordinates of variance 1, except the first coordinate, whose variance is $1 + \beta$. For $n \to \infty$ with $c = d/n$ fixed, Baik and Silverstein (2006) show that the largest eigenvalue of $\mathbf{C} = \frac{1}{n}\mathbf{X}^{\top}\mathbf{X}$ converges to

$$\lambda_{\max} \to \begin{cases} (1 + \sqrt{c})^2, & \beta \le \sqrt{c}, \\ (1 + \beta)\,(1 + c/\beta)\,, & \beta > \sqrt{c}. \end{cases}$$

   1. For $n = 2000$ and $d = 500$, compute the largest eigenvalue of $\mathbf{C}$ for $\beta \in \{0, 0.3, 0.5, 0.7, 1, 2\}$, averaged over five draws, and compare with the prediction.
   2. Fix $d = 500$ and $\beta = 0.4$. Use the formula to find the sample size n above which the largest eigenvalue leaves the bulk. Then increase the number of samples through $n \in \{500, 1000, 2000, 4000, 8000\}$, and compare the largest eigenvalue with the bulk edge, with the prediction, and with the largest eigenvalue for $\beta = 0$ at the same n. Why is the separation hard to see at $n = 4000$?
   3. A practitioner keeps the principal components whose eigenvalues exceed the bulk edge. How large must the variance of a direction be at $c = 1/4$ for it to be kept? How many samples per dimension are needed to detect a direction whose variance exceeds the others by 10%?

33.4 Spectra in Practice

In Section 33.3, our matrices were small enough to factor by hand, and their eigenvalues delivered on every promise: the surfer's error shrank by $1/4$ at every click, just as the second eigenvalue predicted. Outside textbook examples, both comforts vanish. The link matrix of the web has billions of rows, so nobody writes down its characteristic polynomial or computes a full eigendecomposition. Even when eigenvalues are known, they promise less than they appear to. A Jordan block whose eigenvalues are both 0.99 still grew to norm 37 before

decaying (Section 33.3.1.5), and backpropagation through a recurrent network multiplies a gradient by fifty *different* matrices rather than applying one matrix fifty times.

This section takes up those two problems in turn. First, what can we learn about the eigenvalues of a matrix that is far too large to factor? Surprisingly much: the entries alone confine all eigenvalues to a few discs in the complex plane, and repeated matrix–vector multiplication, the one operation that remains cheap at scale, finds the largest eigenvalue and its eigenvector. That is how PageRank was originally computed. Second, what do eigenvalues leave unsaid about a finite number of steps, or about a product of changing matrices? Here the answer is sobering. Eigenvalues govern the long-run behavior of powers of a single matrix; for anything else we need matrix norms, and ultimately the singular values of Section 33.5.

Not every topic below carries the same weight in deep learning, and it helps to know where to focus. Three ideas recur throughout the book. Power iteration (Section 33.4.1.2) estimates the largest stretch of a layer, which is the workhorse behind spectral normalization (Section 33.5.4.3). Transient growth and products of Jacobians (Section 33.4.2.3, Section 33.4.2.5) supply the mathematics behind the exploding and vanishing gradients of Section 8.5. And the random matrices of Section 33.4.2.6 model the weight matrices of Section 4.4 at initialization. Gershgorin's theorem and PageRank are classical highlights that appear less often in deep learning, so a reader in a hurry may skim both; pseudospectra (Section 33.4.2.4) are a specialized diagnostic that can likewise be skipped on a first reading. All numerical examples below use NumPy.

```
import numpy as np
```

33.4.1 Locating and Computing Eigenvalues

33.4.1.1 Gershgorin Discs

Can we tell roughly where the eigenvalues of a matrix lie without ever computing them? For a diagonal matrix, the eigenvalues are simply the diagonal entries. Intuition suggests that if the off-diagonal entries are small, the eigenvalues should stay close to those diagonal entries. The following theorem makes that intuition exact, and its proof takes only three lines.

Theorem (Gershgorin (Gershgorin 1931)**).** *Let $\mathbf{A} = (a_{ij})$ be any $n \times n$ matrix, and for each row i let $r_i = \sum_{j \neq i} |a_{ij}|$ be the sum of the absolute values of its off-diagonal entries. Let $\mathcal{D}_i$ be the closed disc in the complex plane with centre a_{ii} and radius r_i. Then every eigenvalue of $\mathbf{A}$ lies in the union $\bigcup_i \mathcal{D}_i$.*

Proof. Suppose $\mathbf{A}\mathbf{v} = \lambda \mathbf{v}$ with $\mathbf{v} \neq \mathbf{0}$, and pick an index i where $\mathbf{v}$ has its largest entry in magnitude, so that $|v_i| \geq |v_j|$ for all j and $v_i \neq 0$. Reading off row i of $\mathbf{A}\mathbf{v} = \lambda \mathbf{v}$ gives

$$\sum_{j} a_{ij} v_j = \lambda v_i \quad \Longrightarrow \quad (\lambda - a_{ii})\, v_i = \sum_{j \neq i} a_{ij} v_j .$$

Dividing both sides by v_i and taking absolute values, we use the fact that $|v_j / v_i| \leq 1$ for every j:

$$|\lambda - a_{ii}| = \Big|\sum_{j \neq i} a_{ij}\,\tfrac{v_j}{v_i}\Big| \le \sum_{j \neq i} |a_{ij}| = r_i,$$

which says that $\lambda \in \mathcal{D}_i$. $\blacksquare$

Notice what the proof actually tells us: each eigenvalue falls into the disc of the row where its eigenvector is largest. Because we do not know that row in advance, the theorem guarantees only that every eigenvalue lies somewhere in the union of all n discs. To see this in action, consider the non-symmetric matrix

$$\mathbf{A} = \begin{pmatrix} 4 & 1 & 0 \\ -1 & 4 & 0.5 \\ 0.5 & 0 & 1 \end{pmatrix}. \tag{33.37}$$

Its three discs have centres 4, 4, and 1, with radii 1, 1.5, and 0.5. The two discs centred at 4 overlap (in fact, one sits inside the other), whereas the disc centred at 1 is completely separated from them, because $|4 - 1| = 3 > 1.5 + 0.5$. The actual eigenvalues are $3.99 \pm 0.96i$ and 1.03: the complex conjugate pair sits inside the two discs centred at 4, and the real eigenvalue sits inside the isolated disc centred at 1. Figure 33.20 displays the discs alongside the eigenvalues.

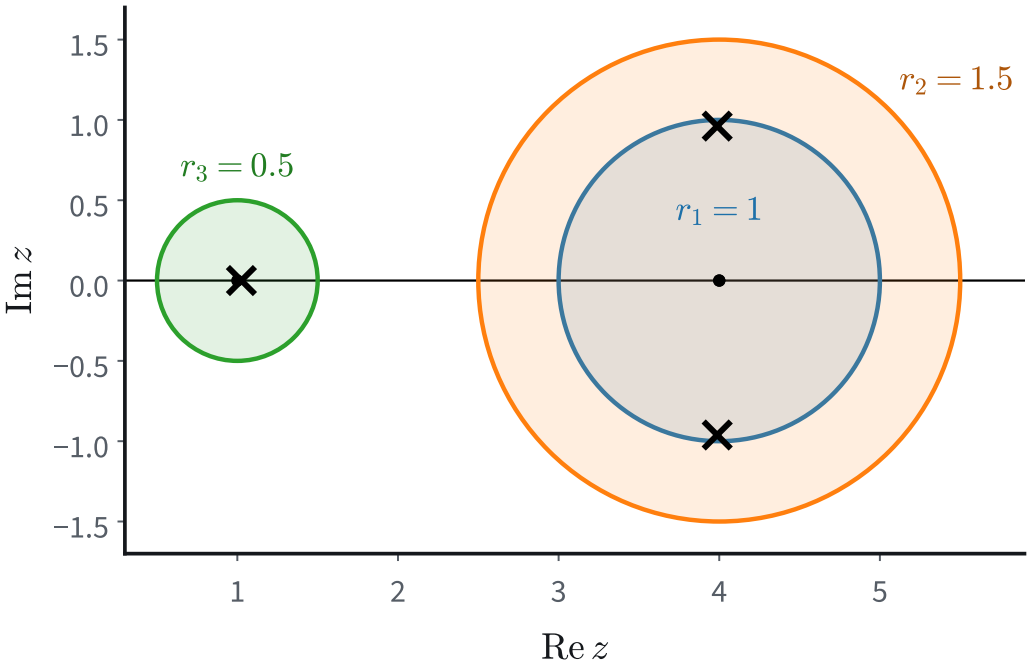


Figure 33.20: Gershgorin discs of the 3×3 matrix in the text (centres at the diagonal entries, radii given by the sums of the absolute off-diagonal entries in each row) and its eigenvalues (crosses). The two overlapping discs centred at 4 together hold two eigenvalues, the complex pair; the disjoint disc centred at 1 holds the third.

That split, with two eigenvalues in the overlapping pair and one in the isolated disc, is no accident. *If a union of m discs is connected and disjoint from all the remaining discs, it contains exactly m eigenvalues*, counted with multiplicity. The reason is continuity: as we scale the off-diagonal entries from zero up to their actual values, the eigenvalues start at the diagonal entries and move continuously, so they can never jump across the empty gap between separated groups of discs (Horn and Johnson 2012). Let us verify the centres, radii, and eigenvalues numerically.

```
A = np.array([[4.0, 1.0, 0.0],
              [-1.0, 4.0, 0.5],
              [0.5, 0.0, 1.0]])
centres = np.diag(A)
radii = np.abs(A).sum(axis=1) - np.abs(centres)
print('centres:', centres, '  radii:', radii)
for z in np.linalg.eigvals(A):
    print(f'eigenvalue {z:.3f}   distances to the centres: '
          f'{np.abs(z - centres).round(2)}')
```

```
centres: [4. 4. 1.]   radii: [1.  1.5 0.5]
eigenvalue 3.987+0.961j   distances to the centres: [0.96 0.96 3.14]
eigenvalue 3.987-0.961j   distances to the centres: [0.96 0.96 3.14]
eigenvalue 1.025+0.000j   distances to the centres: [2.97 2.97 0.03]
```

Here the complex pair sits at distance 0.96 from 4, right near the rim of the inner disc of radius 1, showing that Gershgorin's bound can be nearly tight.

One immediate corollary requires no computation at all. If $\mathbf{A}$ is *strictly diagonally dominant* with positive diagonal entries, meaning $a_{ii} > r_i$ for every i, then every disc lies strictly in the open right half-plane. Zero cannot be an eigenvalue, so $\mathbf{A}$ is automatically invertible. If $\mathbf{A}$ is also symmetric, its eigenvalues are real and strictly positive, which makes it positive definite.

33.4.1.2 Power Iteration

Gershgorin discs locate eigenvalues only roughly. What if we want an accurate eigenvalue of a matrix with a billion rows? Factoring such a matrix is out of the question, yet if it is sparse we can still multiply it by a vector cheaply. What happens if we start with a vector $\mathbf{v}_0$ and simply multiply by $\mathbf{A}$ over and over again?

The power expansion $\mathbf{A}^k\mathbf{x} = \sum_i c_i \lambda_i^k \mathbf{v}_i$ of Section 33.3.1.3 gives the answer. Suppose $\mathbf{A}$ is diagonalizable with eigenvectors $\mathbf{v}_i$, ordered by the magnitude of their eigenvalues, and suppose the largest eigenvalue is *strictly dominant*, meaning $|\lambda_1| > |\lambda_2| \geq \cdots \geq |\lambda_n|$. (When $\mathbf{A}$ is real, strict dominance forces λ_1 to be real as well, because a complex eigenvalue would share its modulus with its complex conjugate.) Expanding $\mathbf{v}_0 = \sum_i c_i \mathbf{v}_i$ and factoring out λ_1^k yields

$$\mathbf{A}^k \mathbf{v}_0 = \sum_{i=1}^{n} c_i \lambda_i^k \mathbf{v}_i = \lambda_1^k \Big(c_1 \mathbf{v}_1 + \sum_{i \geq 2} c_i \Big(\frac{\lambda_i}{\lambda_1} \Big)^k \mathbf{v}_i \Big). \tag{33.38}$$

Every ratio $|\lambda_i / \lambda_1|$ for $i \geq 2$ is strictly less than 1, so every trailing term vanishes as k grows. Provided $c_1 \neq 0$, the bracket converges to $c_1 \mathbf{v}_1$. Thus the direction of $\mathbf{A}^k \mathbf{v}_0$ approaches the

dominant eigenvector $\mathbf{v}_1$ with an error decaying like $|\lambda_2/\lambda_1|^k$, while its length scales by a factor approaching $|\lambda_1|$ at each step. To keep the entries from overflowing or underflowing, we normalize the vector back to unit length after every multiplication; this simple loop is *power iteration* (Golub and Van Loan 1996). When $\mathbf{A}$ is symmetric, we can do even better for the eigenvalue itself: the Rayleigh quotient of the current iterate converges to λ_1 at the squared rate $|\lambda_2/\lambda_1|^{2k}$. Because the eigenvectors of a symmetric matrix are orthogonal, a directional misalignment of size δ shifts the Rayleigh quotient only by $O(\delta^2)$ (Exercise 2).

We can see both rates in a concrete 2×2 example. In Section 1.3, we computed the iterates of $\mathbf{S} = \left(\begin{smallmatrix} 2 & 1 \\ 1 & 2 \end{smallmatrix}\right)$ starting from $\mathbf{e}_1$: $(2,1)$, $(5,4)$, $(14,13)$, and in general $((3^k+1)/2,\ (3^k-1)/2)$. In terms of the unit eigenvectors $\mathbf{v}_1 = (1,1)/\sqrt{2}$ and $\mathbf{v}_2 = (1,-1)/\sqrt{2}$, we have $\mathbf{e}_1 = (\mathbf{v}_1+\mathbf{v}_2)/\sqrt{2}$, so

$$\mathbf{S}^k\mathbf{e}_1 = \frac{3^k\mathbf{v}_1 + \mathbf{v}_2}{\sqrt{2}}, \qquad \tan\angle(\mathbf{S}^k\mathbf{e}_1, \mathbf{v}_1) = 3^{-k}, \qquad 3 - R(\mathbf{S}^k\mathbf{e}_1) = \frac{2}{9^k+1}.$$

With $|\lambda_2/\lambda_1| = 1/3$, the directional error drops by a factor of 3 at every step: the sine of the angle to $\mathbf{v}_1$ is 0.316, 0.110, and 0.037 for $k = 1, 2, 3$. Meanwhile the Rayleigh-quotient error drops by a factor of 9 per step: 0.2, 0.024, and 0.0027. Figure 33.21 plots both curves on a logarithmic scale.

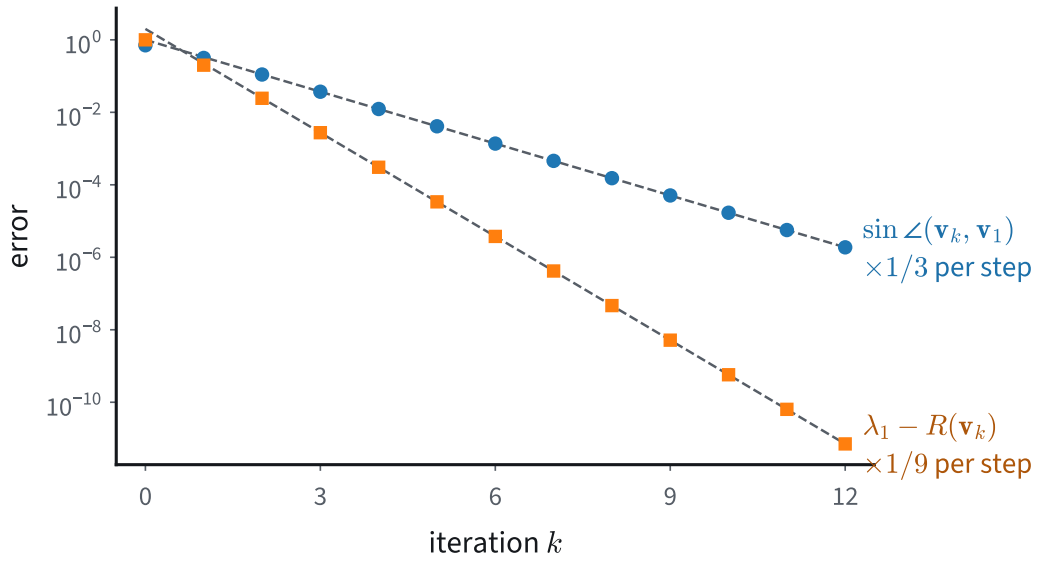


Figure 33.21: Power iteration on $\mathbf{S} = \left(\begin{smallmatrix} 2 & 1 \\ 1 & 2 \end{smallmatrix}\right)$ starting from $\mathbf{e}_1$. The direction error drops by a factor of 3 per step, whereas the eigenvalue estimate improves by a factor of 9 per step, the squared rate that comes from orthogonal eigenvectors. Dashed lines show 3^{-k} and $2\cdot 9^{-k}$.

When can power iteration fail? Two assumptions in Equation 33.38 matter. First, the starting vector needs a nonzero component along $\mathbf{v}_1$ ($c_1 \neq 0$); choosing $\mathbf{v}_0$ at random guarantees this with probability one. Second, the largest eigenvalue must be strictly dominant. If the largest modulus is shared by a complex conjugate pair, the dominant component of the iterate rotates indefinitely instead of settling on a single line. The ratio of consecutive norms

then oscillates forever, just as it did for the running matrix of Section 33.3.1.6. Even in that case, however, the geometric average $\|\mathbf{A}^k\mathbf{v}_0\|^{1/k}$ still converges to the largest modulus.

Matrices with strictly positive entries avoid this trouble altogether: as we will see in Section 33.4.1.3, the Perron–Frobenius theorem guarantees a real, simple, strictly dominant eigenvalue. Let us test this by running 200 steps of power iteration on a random 5×5 matrix with entries drawn uniformly from $[0, 1]$, starting from a random vector, and comparing the final norm ratio with the true largest eigenvalue modulus.

```
A = np.random.rand(5, 5)                # positive entries
v = np.random.randn(5)
for _ in range(200):
    Av = A @ v
    ratio = np.linalg.norm(Av) / np.linalg.norm(v)
    v = Av / np.linalg.norm(Av)
mods = np.sort(np.abs(np.linalg.eigvals(A)))[::-1]
print(f'norm ratio after 200 steps = {ratio:.10f}')
print(f'largest |eigenvalue|       = {mods[0]:.10f}')
print(f'|lambda_2| / |lambda_1|    = {mods[1] / mods[0]:.3f}')
```

```
norm ratio after 200 steps = 2.5254088042
largest |eigenvalue|       = 2.5254088042
|lambda_2| / |lambda_1|    = 0.299
```

On every draw, the norm ratio matches the largest eigenvalue modulus to all printed digits. To see how typical this fast convergence is, let us compute the gap ratio $|\lambda_2/\lambda_1|$ across 10,000 such positive matrices. For comparison, we also draw 10,000 matrices with standard Gaussian entries, which have no positivity guarantee, and measure how often their dominant eigenvalues form a complex conjugate pair.

```
mods = np.sort(np.abs(np.linalg.eigvals(np.random.rand(10000, 5, 5))), axis=1)
print('uniform entries, gap ratio |lambda_2| / |lambda_1|, 5% and 95% quantiles:',
      np.quantile(mods[:, -2] / mods[:, -1], [0.05, 0.95]).round(2))
lam = np.linalg.eigvals(np.random.randn(10000, 5, 5))
top = lam[np.arange(10000), np.abs(lam).argmax(axis=1)]   # a dominant eigenvalue
print('Gaussian entries, fraction with a complex dominant pair:',
      np.mean(top.imag != 0).round(2))
```

```
uniform entries, gap ratio |lambda_2| / |lambda_1|, 5% and 95% quantiles: [0.14 0.37]
Gaussian entries, fraction with a complex dominant pair: 0.43
```

For uniform positive entries, the gap ratio falls between roughly 0.14 and 0.37 in nine draws out of ten, so 200 iterations drive the direction error far below floating-point precision. For Gaussian entries, by contrast, about four draws in ten have a complex dominant pair, and on those matrices the norm ratio never settles.

33.4.1.3 PageRank

For matrices with non-negative entries, the dominant eigenvector is often the very object we care about, and power iteration is how we compute it. To state when that eigenvector is unique and well behaved, call a non-negative matrix *irreducible* if, in the directed graph that places an edge $j \to i$ whenever $a_{ij} > 0$, every node can reach every other node. The *Perron–*

Frobenius theorem (Horn and Johnson 2012) guarantees that any irreducible non-negative matrix has a real, positive eigenvalue equal to the largest modulus among all its eigenvalues, that this eigenvalue is simple, and that its eigenvector can be chosen with strictly positive entries. If every entry of the matrix is strictly positive, this eigenvalue is also strictly dominant, meaning every other eigenvalue has strictly smaller modulus.

Google's original PageRank algorithm (Brin and Page 1998) turns web ranking into exactly such an eigenvector problem. Consider a miniature web of three pages with links $1 \to 2$, $2 \to 1$, and $2 \to 3$. A random surfer follows an outgoing link chosen uniformly at random. Page 3 has no outgoing links, so we repair that dead end by letting the surfer jump from page 3 to any page with equal probability. The repaired transition matrix $\mathbf{P}$ is column-stochastic, with column j holding the click probabilities out of page j:

$$\mathbf{P} = \begin{pmatrix} 0 & \tfrac{1}{2} & \tfrac{1}{3} \\ 1 & 0 & \tfrac{1}{3} \\ 0 & \tfrac{1}{2} & \tfrac{1}{3} \end{pmatrix}.$$

In the long run, the fraction of time the surfer spends on each page is the stationary distribution π satisfying $\mathbf{P}\pi = \pi$. We can solve this by hand: rows 1 and 3 both give $\pi_1 = \tfrac{1}{2}\pi_2 + \tfrac{1}{3}\pi_3 = \pi_3$, and substituting $\pi_1 = \pi_3$ back into row 1 gives $\pi_2 = \tfrac{4}{3}\pi_3$. Normalizing the entries to sum to 1 yields $\pi = (0.3, 0.4, 0.3)$. Page 2 ranks first because page 1 sends it all of its clicks, whereas page 2 splits its clicks between pages 1 and 3. The eigenvalues of $\mathbf{P}$ are 1, $-\tfrac{2}{3}$, and 0, so power iteration from any initial distribution converges at the rate $2/3$ per step.

In fact, *every* column-stochastic matrix has 1 as an eigenvalue of largest modulus. First, because every column sums to 1, we have $\mathbf{1}^{\top}\mathbf{P} = \mathbf{1}^{\top}$, which means 1 is an eigenvalue of $\mathbf{P}^{\top}$, and therefore also of $\mathbf{P}$ (a matrix and its transpose share the same characteristic polynomial, because $\det \mathbf{X}^{\top} = \det \mathbf{X}$). Second, applying Gershgorin's theorem to $\mathbf{P}^{\top}$, whose rows are the columns of $\mathbf{P}$, places every eigenvalue inside a disc with centre $p_{ii} \geq 0$ and radius $1 - p_{ii}$; every such disc sits inside the unit disc.

The real web, however, is messier than our three-page example: it contains link cycles and isolated pockets of pages from which a surfer cannot escape. The repaired matrix $\mathbf{P}$ can then fail to be irreducible or can have several eigenvalues of modulus 1. For instance, a simple cycle of four pages has eigenvalues ± 1 and $\pm i$, so undamped power iteration on it never settles (Exercise 4). PageRank solves both problems at once by iterating the *damped* matrix

$$\mathbf{G} = \alpha \mathbf{P} + (1 - \alpha)\tfrac{1}{n}\mathbf{1}\mathbf{1}^{\top}, \qquad \alpha = 0.85,$$

which describes a surfer who follows a random link with probability 0.85 and teleports to a uniformly random page with probability 0.15. Because every entry of $\mathbf{G}$ is strictly positive, the Perron–Frobenius theorem applies by construction. Better still, Haveliwala and Kamvar (2003) show that $|\lambda_2(\mathbf{G})| \leq \alpha$, so the error of power iteration shrinks by at least a factor of 0.85 at every step, no matter how tangled the link graph is. That single bound made web-scale PageRank practical: fifty sweeps cut the error by more than a factor of 3000 ($0.85^{50} \approx 3 \times 10^{-4}$), regardless of how many billions of pages are in the graph.

At web scale, $\mathbf{P}$ is stored sparsely as one entry per link, and the dense matrix $\mathbf{G}$ is never formed explicitly. For any probability vector $\mathbf{p}$ (where $\mathbf{1}^{\top}\mathbf{p} = 1$), we can evaluate $\mathbf{G}\mathbf{p} =$

$\alpha \mathbf{P}\mathbf{p} + (1-\alpha)\frac{1}{n}\mathbf{1}$ using just one sparse matrix–vector product plus a constant shift. The dead-end repair for pages with no outgoing links is a rank-one term handled in the same way.

33.4.2 Iteration and Stability

33.4.2.1 The Spectral Radius

Power iteration singled out one number: the largest eigenvalue modulus. That number is called the *spectral radius*, $\rho(\mathbf{A}) = \max_i |\lambda_i|$, and it governs the long-run growth rate of the powers of a single fixed matrix. Gelfand's formula makes this exact: for every square matrix $\mathbf{A}$ and every matrix norm,

$$\lim_{k \to \infty} \|\mathbf{A}^k\|^{1/k} = \rho(\mathbf{A}).$$

The easier half of this statement, $\rho(\mathbf{A}) \le \|\mathbf{A}^k\|^{1/k}$ for every $k \ge 1$, is Exercise 1.

What Gelfand's formula does *not* tell us is what happens at any fixed, finite step k, and we have already met two matrices where that gap matters. The shear $\mathbf{T} = \left(\begin{smallmatrix} 1 & 1 \\ 0 & 1 \end{smallmatrix}\right)$ has $\rho(\mathbf{T}) = 1$, yet its powers $\mathbf{T}^k = \left(\begin{smallmatrix} 1 & k \\ 0 & 1 \end{smallmatrix}\right)$ grow without bound, so $\rho \le 1$ does not even keep the iterates bounded. The Jordan block $\mathbf{J}_2(0.99)$ has $\rho < 1$ and must eventually decay to zero, yet its powers climb to norm 37 before turning downward, so $\rho < 1$ does not rule out large transient growth. When, then, do the eigenvalues control *every* step rather than only the distant limit?

33.4.2.2 Normal Matrices

Both of our counterexamples, the shear and the Jordan block, are defective: they lack enough eigenvectors to form a basis. Let us go straight to the opposite extreme and consider matrices that admit an *orthonormal* basis of eigenvectors. Such a matrix acts independently along mutually perpendicular directions, so its eigenvalues ought to tell the whole story at every step.

A real matrix is called *normal* if it commutes with its transpose, $\mathbf{A}^\top \mathbf{A} = \mathbf{A}\mathbf{A}^\top$ (for a complex matrix, we replace the transpose by the conjugate transpose $\mathbf{A}^*$, requiring $\mathbf{A}^*\mathbf{A} = \mathbf{A}\mathbf{A}^*$). Symmetric matrices, orthogonal matrices, and the rotation–scaling blocks $r\mathbf{R}_\theta$ of Section 33.3.1.6 are all normal. By contrast, our running example $\mathbf{A} = \left(\begin{smallmatrix} 1 & 2 \\ -1 & 3 \end{smallmatrix}\right)$ is not normal: $\mathbf{A}\mathbf{A}^\top = \left(\begin{smallmatrix} 5 & 5 \\ 5 & 10 \end{smallmatrix}\right)$, whereas $\mathbf{A}^\top\mathbf{A} = \left(\begin{smallmatrix} 2 & -1 \\ -1 & 13 \end{smallmatrix}\right)$. That failure to commute is the reason its iterates spiraled along ellipses rather than circles.

The *complex spectral theorem* (Horn and Johnson 2012) characterizes this class completely: a matrix is normal if and only if it has an orthonormal eigenbasis over $\mathbb{C}$, meaning $\mathbf{A} = \mathbf{U}\Lambda\mathbf{U}^*$ with $\mathbf{U}$ unitary ($\mathbf{U}^*\mathbf{U} = \mathbf{I}$). The proof follows the same inductive argument as the real spectral theorem in Section 33.3.2.1.

To measure the worst-case amplification in a single step, write $\|\mathbf{X}\|_2 = \max_{\|\mathbf{x}\|=1} \|\mathbf{X}\mathbf{x}\|$ for the *spectral norm* of $\mathbf{X}$, the largest factor by which $\mathbf{X}$ can stretch a vector. Because a unitary change of basis preserves Euclidean lengths, every power of a normal matrix satisfies

$$\|\mathbf{A}^k\|_2 = \|\mathbf{U}\Lambda^k\mathbf{U}^*\|_2 = \max_i |\lambda_i|^k = \rho(\mathbf{A})^k. \tag{33.39}$$

For a normal matrix, the eigenvalues control the worst-case amplification at every single step k, not just asymptotically.

What happens when $\mathbf{A}$ is diagonalizable, $\mathbf{A} = \mathbf{V}\Lambda\mathbf{V}^{-1}$, but not normal? Now the eigenvector matrix $\mathbf{V}$ is no longer orthogonal, and submultiplicativity of the spectral norm gives only the upper bound

$$\|\mathbf{A}^k\|_2 \le \underbrace{\|\mathbf{V}\|_2\|\mathbf{V}^{-1}\|_2}_{\kappa_2(\mathbf{V})}\,\rho(\mathbf{A})^k. \tag{33.40}$$

The prefactor $\kappa_2(\mathbf{V})$ is the condition number (Section 33.5.3.2) of the eigenvector matrix. It equals 1 when the eigenvectors are orthogonal, and grows arbitrarily large when two eigenvectors point in nearly the same direction. The transient growth it permits has a clean geometric cause: cancellation. When eigenvectors are nearly parallel, an ordinary unit vector can only be expressed as a combination of huge eigen-components that nearly cancel each other out. As we apply $\mathbf{A}$ repeatedly, those components decay at different rates, the delicate cancellation comes undone, and the iterate balloons in length before the common exponential decay finally takes over.

33.4.2.3 Transient Growth

The simplest matrix that displays this transient growth is the upper-triangular 2×2 matrix

$$\mathbf{M} = \begin{pmatrix} 0.9 & 4 \\ 0 & 0.8 \end{pmatrix}. \tag{33.41}$$

Its eigenvalues are its diagonal entries 0.9 and 0.8. Both lie safely inside the unit circle, and because they are distinct, $\mathbf{M}$ is diagonalizable. Let us compute its eigenvectors by hand and expand the unit vector $\mathbf{e}_2 = (0, 1)$ in them. For $\lambda = 0.9$, the second row of $\mathbf{M} - 0.9\mathbf{I}$ reads $-0.1v_2 = 0$, giving $\mathbf{v}_1 = (1, 0)$. For $\lambda = 0.8$, the first row of $\mathbf{M} - 0.8\mathbf{I}$ reads $0.1v_1 + 4v_2 = 0$, giving $\mathbf{v}_2 = (-40, 1)$. Notice how close these two directions are: the angle between them is only $\arctan(1/40) \approx 1.4^\circ$. Expanding $\mathbf{e}_2$ in this nearly parallel basis gives $\mathbf{e}_2 = 40\mathbf{v}_1 + \mathbf{v}_2$, and applying the eigenvector power expansion (Section 33.3.1.3) yields

$$\mathbf{M}^k\mathbf{e}_2 = 40 \cdot 0.9^k\,\mathbf{v}_1 + 0.8^k\,\mathbf{v}_2 = \begin{pmatrix} 40\,(0.9^k - 0.8^k) \\ 0.8^k \end{pmatrix}. \tag{33.42}$$

At $k = 0$, a vector of length 1 is formed as the difference of two eigen-components of length roughly 40. Both components decay at every step, yet because 0.8^k falls faster than 0.9^k, their cancellation unravels and the sum initially grows. The first coordinate $40(0.9^k - 0.8^k)$ peaks where its derivative with respect to k vanishes, namely where $(9/8)^k = \ln 0.8 / \ln 0.9$, which occurs at $k \approx 6.4$ with a height of about 10.8. Figure 33.22 illustrates the two components at $k = 0$ and at $k = 6$. For comparison, the general bound Equation 33.40 with unit eigenvectors has $\kappa_2(\mathbf{V}) = 80$ and predicts $\|\mathbf{M}^6\|_2 \le 80 \cdot 0.9^6 \approx 42.5$. The bound holds, though it is conservative by a factor of four compared with the true peak of 10.8.

Let us check this calculation numerically and compare $\mathbf{M}$ against the diagonal matrix $\Lambda = \mathrm{diag}(0.9, 0.8)$, which has the exact same eigenvalues but is normal.

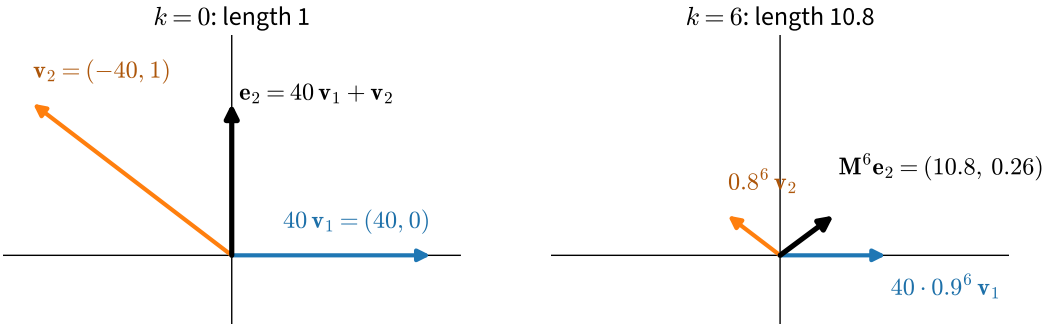


Figure 33.22: Growth of $\mathbf{M}^k\mathbf{e}_2$ before decay. Left: at $k = 0$, the unit vector $\mathbf{e}_2$ is the difference of two eigen-components of length about 40 that nearly cancel. Right: six steps later the two components have decayed by 0.9^6 and 0.8^6, breaking the cancellation and producing a vector more than ten times longer. The vertical axis is stretched about 30-fold; drawn to scale, $\mathbf{v}_1$ and $\mathbf{v}_2$ are only 1.4° apart.

```
M = np.array([[0.9, 4.0], [0.0, 0.8]])
Lam = np.diag([0.9, 0.8])                  # same eigenvalues, normal
lam, V = np.linalg.eig(M)                  # unit eigenvectors as columns
print('eigenvalues:', lam)
print('coefficients of e_2 on the unit eigenvectors:',
      np.linalg.solve(V, [0., 1.]))
ks = np.arange(61)
norm_M = [np.linalg.norm(np.linalg.matrix_power(M, k), 2) for k in ks]
norm_L = [np.linalg.norm(np.linalg.matrix_power(Lam, k), 2) for k in ks]
k_peak = int(np.argmax(norm_M))
print(f'max ||M^k|| = {norm_M[k_peak]:.3f} at k = {k_peak};  '
      f'max ||Lambda^k|| = {max(norm_L):.3f}')
print(f'kappa_2(V) = {np.linalg.cond(V):.1f}')
```

```
eigenvalues: [0.9 0.8]
coefficients of e_2 on the unit eigenvectors: [40.         40.01249805]
max ||M^k|| = 10.788 at k = 6;  max ||Lambda^k|| = 1.000
kappa_2(V) = 80.0
```

Expressed in the unit eigenvectors, $\mathbf{e}_2$ carries two coefficients of magnitude about 40, matching the lengths of $40\mathbf{v}_1$ and $\mathbf{v}_2$. The spectral norm $\|\mathbf{M}^k\|_2$ climbs to 10.8 at $k = 6$, whereas $\|\Lambda^k\|_2$ never exceeds its starting value of 1. Even though both matrices share the same eigenvalues, the normal matrix contracts from the very first step while the non-normal matrix grows more than ten-fold before decaying. The left panel of Figure 33.23 plots both trajectories on a logarithmic scale; past the peak, both curves eventually descend with the same slope $\log 0.9$, just as Gelfand's formula guarantees.

33.4.2.4 Pseudospectra

This subsection adds a diagnostic for non-normality rather than a new phenomenon, and may be skipped on a first pass. Beyond transient growth, a second symptom of non-normality is extreme sensitivity of the eigenvalues to small perturbations. To make that

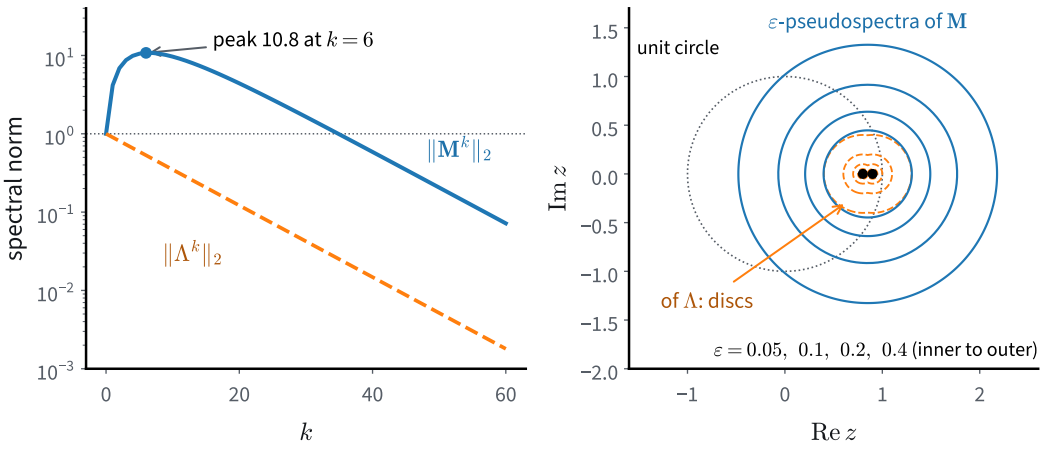


Figure 33.23: Left: $\|\mathbf{M}^k\|_2$ and $\|\Lambda^k\|_2$ for the same eigenvalues 0.9 and 0.8; $\mathbf{M}$ climbs to 10.8 at $k = 6$ before both curves decay with slope $\log 0.9$. Right: the ε-pseudospectra for $\varepsilon = 0.05, 0.1, 0.2, 0.4$; around the normal matrix Λ they are simply discs of radius ε, whereas around $\mathbf{M}$ they bulge far past the unit circle.

sensitivity visible, the *ε-pseudospectrum* collects every complex number that becomes an eigenvalue of $\mathbf{A} + \mathbf{E}$ for some perturbation $\mathbf{E}$ of spectral norm at most ε. Conveniently, the same set can be computed directly from the *resolvent* $(z\mathbf{I} - \mathbf{A})^{-1}$:

$$\begin{aligned} \Lambda_\varepsilon(\mathbf{A}) &= \{z \in \mathbb{C} : z \text{ is an eigenvalue of } \mathbf{A} + \mathbf{E} \text{ for some } \|\mathbf{E}\|_2 \le \varepsilon\} \\ &= \{z \in \mathbb{C} : \|(z\mathbf{I} - \mathbf{A})^{-1}\|_2 \ge 1/\varepsilon\}, \end{aligned} \tag{33.43}$$

where the resolvent norm is taken to be infinite at the eigenvalues of $\mathbf{A}$ itself. Proving the equivalence of these two descriptions is Exercise 3. For a normal matrix, the resolvent norm at z is simply the reciprocal of the distance from z to the nearest eigenvalue, so $\Lambda_\varepsilon(\mathbf{A})$ is just the union of discs of radius ε around the eigenvalues. For a non-normal matrix like $\mathbf{M}$, the resolvent norm can be vastly larger, as we can check at $z = 1$ on the unit circle.

```
I = np.eye(2)
print(f'||(I - M)^-1||      = {np.linalg.norm(np.linalg.inv(I - M), 2):.1f}')
print(f'||(I - Lambda)^-1|| = {np.linalg.norm(np.linalg.inv(I - Lam), 2):.1f}')
```

```
||(I - M)^-1||      = 200.3
||(I - Lambda)^-1|| = 10.0
```

For Λ the resolvent norm is 10, the reciprocal of the distance 0.1 from 1 to the eigenvalue 0.9. For $\mathbf{M}$ it jumps to 200.3: a tiny perturbation of size $1/200.3 \approx 0.005$ already pushes an eigenvalue of $\mathbf{M}$ onto the unit circle, whereas Λ requires a perturbation twenty times larger, of size 0.1 (right panel of Figure 33.23). Because floating-point rounding acts like a small perturbation to the matrix entries, the numerically computed eigenvalues of a strongly non-normal matrix can be surprisingly inaccurate. Section 33.6 follows this thread further with the Kreiss matrix theorem.

33.4.2.5 Products of Jacobians

So far we have multiplied a vector by the *same* matrix repeatedly. Recurrent neural networks (Section 8.5) break even that assumption. A recurrent layer updates its hidden state as $\mathbf{h}_t = \phi(\mathbf{W}\mathbf{h}_{t-1} + \cdots)$, using the same weight matrix $\mathbf{W}$ at every step, yet backpropagation through time multiplies the gradient by the sequence of per-step Jacobians $\mathbf{D}_t = \partial \mathbf{h}_t / \partial \mathbf{h}_{t-1}$:

$$\nabla_{\mathbf{h}_0} \mathcal{L} = \mathbf{D}_1^\top \mathbf{D}_2^\top \cdots \mathbf{D}_T^\top \nabla_{\mathbf{h}_T} \mathcal{L}.$$

Because the nonlinearity ϕ is evaluated at a different hidden state at each time step, the Jacobians $\mathbf{D}_1, \dots, \mathbf{D}_T$ are different matrices, and the theory of powers of a single matrix no longer applies.

In fact, the spectral radii of the individual factors tell us nothing at all about their product. Consider the two 2×2 matrices

$$\mathbf{D}_1 = \begin{pmatrix} 0 & 2 \\ 0 & 0 \end{pmatrix}, \qquad \mathbf{D}_2 = \begin{pmatrix} 0 & 0 \\ 2 & 0 \end{pmatrix}, \qquad \mathbf{D}_2 \mathbf{D}_1 = \begin{pmatrix} 0 & 0 \\ 0 & 4 \end{pmatrix}.$$

Both $\mathbf{D}_1$ and $\mathbf{D}_2$ are nilpotent with $\rho = 0$, so every power of either matrix vanishes by the second step. Yet their product has $\rho = 4$: $\mathbf{D}_1$ copies the second coordinate into the first while doubling it, and $\mathbf{D}_2$ copies it straight back while doubling it again.

It is therefore helpful to keep three distinct quantities straight when studying iterated linear maps. The *spectral radius* gives the asymptotic growth rate of powers of one fixed matrix. The *spectral norm* bounds any finite product of changing matrices, because submultiplicativity gives $\|\mathbf{D}_T \cdots \mathbf{D}_1\|_2 \le \prod_t \|\mathbf{D}_t\|_2$ (in our example, $4 \le 2 \cdot 2$). Finally, the *top Lyapunov exponent*

$$\mu = \lim_{T \to \infty} \frac{1}{T} \log \|\mathbf{D}_T \cdots \mathbf{D}_1 \mathbf{x}_0\|$$

measures the average logarithmic growth rate per step along a long product of changing matrices. When the factors are independent and identically distributed random matrices, this limit exists with probability one under mild conditions (Furstenberg and Kesten 1960).

When a gradient passes backward through a sequence of Jacobians with $\mu > 0$, it grows exponentially and explodes; when $\mu < 0$, it shrinks exponentially and vanishes (Bengio et al. 1994; Pascanu et al. 2013). Notice that the standard remedies in deep learning target norms and Jacobians rather than eigenvalues. Orthogonal initialization (Saxe et al. 2014) chooses an orthogonal weight matrix so that a linear recurrence preserves the Euclidean length of every vector at every step, a much stronger guarantee than $\rho = 1$. Gradient clipping caps the norm of the gradient directly after backpropagation. Gated architectures such as LSTMs and GRUs alter the Jacobians themselves, creating pathways along which selected directions pass through with factors close to 1 (Section 8.5.1.1).

33.4.2.6 Random Matrices at Initialization

How large are the spectral radius and spectral norm for the random weight matrices of a freshly initialized neural network? Consider an $n \times n$ matrix $\mathbf{A}$ with independent standard

Gaussian entries, known as the real *Ginibre ensemble* (Ginibre 1965). As $n \to \infty$, the eigenvalues rescaled by $1/\sqrt{n}$ spread uniformly over the unit disc in the complex plane. This is the *circular law*, which Tao and Vu (2010) proved in general for independent, identically distributed entries of mean 0 and variance 1. The circular law tells us where almost all eigenvalues live, though by itself it does not rule out a few stray outliers. A separate theorem establishes that no outlier escapes the unit circle, so $\rho(\mathbf{A})/\sqrt{n} \to 1$ with probability one (Geman 1986).

The spectral norm behaves quite differently: $\|\mathbf{A}\|_2/\sqrt{n} \to 2$ (Yin et al. 1988). This is a special case of the Marchenko–Pastur law (Section 33.3.2.4) at aspect ratio $c = 1$: the largest eigenvalue of $\frac{1}{n}\mathbf{A}^\top\mathbf{A}$ converges to the upper edge $(1+\sqrt{c})^2 = 4$, and $\|\mathbf{A}\|_2^2$ is simply n times that eigenvalue. Even at $n = 1000$, a single random draw sits right next to both asymptotic limits.

```
n = 1000
A = np.random.randn(n, n)
print(f'rho(A) / sqrt(n)       = {np.abs(np.linalg.eigvals(A)).max() / n ** 0.5:.3f}')
print(f'sigma_max(A) / sqrt(n) = {np.linalg.norm(A, 2) / n ** 0.5:.3f}')
```

```
rho(A) / sqrt(n)       = 1.030
sigma_max(A) / sqrt(n) = 1.990
```

At large n, the spectral radius lands within a few percent of $\sqrt{n}$ while the largest stretch sits close to $2\sqrt{n}$: a random Gaussian matrix stretches certain vectors twice as far as its eigenvalues would lead you to expect. Standard initialization schemes divide random weights by $\sqrt{\text{fan-in}}$ for a different reason altogether. With entries of variance $1/n$, we get $\mathbb{E}\|\mathbf{Ax}\|^2 = \|\mathbf{x}\|^2$ for every fixed vector $\mathbf{x}$, so squared lengths are preserved on average. Having a spectral radius near 1 is merely a side effect, and as Section 33.4.2.5 showed, it does not control a product of different matrices. Surprisingly, a long product of independent matrices with this exact scaling *shrinks* a typical vector, giving $\mu < 0$, even though every individual factor preserves squared length on average. The reason is the concavity of the logarithm: for a unit vector $\mathbf{x}$, the expectation of $\log \|\mathbf{Ax}\|^2$ lies strictly below the logarithm of the expectation of $\|\mathbf{Ax}\|^2$ (which is $\log 1 = 0$), and it is the logarithms that add up along a deep product (Exercise 5).

Both the non-normal matrix $\mathbf{M}$ and the random Gaussian matrix point to the same conclusion: eigenvalues tell us little about how much a matrix stretches vectors over a finite number of steps, and for rectangular matrices eigenvalues do not even exist. To measure stretching directly, for any matrix of any shape, we turn next to the singular values of Section 33.5.

33.4.3 Summary

Even without factoring a matrix, Gershgorin discs locate its eigenvalues directly from its entries, and power iteration extracts the dominant eigenpair at the geometric rate $|\lambda_2/\lambda_1|$, the mechanism behind PageRank. The spectral radius $\rho(\mathbf{A})$ governs the long-run rate of powers of a single fixed matrix and nothing more: a non-normal matrix can grow dramatically before it decays, and the eigenvalues of individual factors reveal nothing about a product of changing matrices. For products of changing maps, the spectral norm bounds finite steps and the top Lyapunov exponent gives the average long-run rate. A random $n \times n$ Gaussian matrix at initialization makes the gap between eigenvalues and stretching concrete:

its spectral radius is close to $\sqrt{n}$, whereas its spectral norm is close to $2\sqrt{n}$.

33.4.4 Exercises

1. **Spectral radius and norms.** The spectral radius bounds every power of a matrix from below, and for normal matrices it also bounds them from above.

   1. Show that $\rho(\mathbf{A}) \le \|\mathbf{A}^k\|_2^{1/k}$ for every square matrix $\mathbf{A}$ and every $k \ge 1$.
   2. The chain Equation 33.39 uses two facts without proof. Show that $\|\mathbf{U}\mathbf{X}\mathbf{U}^*\|_2 = \|\mathbf{X}\|_2$ for every unitary $\mathbf{U}$ and every square $\mathbf{X}$, and that the spectral norm of a diagonal matrix is the largest modulus of its diagonal entries.
   3. Show that the converse of Equation 33.39 fails: find a 3×3 matrix that is not normal but satisfies $\|\mathbf{A}^k\|_2 = \rho(\mathbf{A})^k$ for every $k \ge 1$.
   4. For a given $t > 0$, construct a diagonalizable 2×2 matrix with spectral radius $1/2$ and $\|\mathbf{A}\|_2 > t$.

2. code **Power-iteration rates.** Let $\mathbf{S}$ be the matrix of Figure 33.21.

   1. By Equation 33.38, the misalignment between the iterate and $\mathbf{v}_1$ decays at the rate $|\lambda_2/\lambda_1| = 1/3$. How many iterations reduce it by a factor of 100?
   2. Let $\mathbf{A}$ be symmetric with unit eigenvector $\mathbf{v}_1$ for λ_1, and let $\mathbf{x} = \mathbf{v}_1 + \delta\mathbf{u}$ for a unit vector $\mathbf{u}$ orthogonal to $\mathbf{v}_1$. Show that the Rayleigh quotient of Section 33.3.2.3 satisfies $R(\mathbf{x}) - \lambda_1 = O(\delta^2)$, with no term linear in δ. At which rate does the Rayleigh quotient of the power iterates therefore approach λ_1? Where does the argument use that $\mathbf{A}$ is symmetric?
   3. Repeat the experiment of Figure 33.21 for $\left(\begin{smallmatrix} 1 & 0.1 \\ 0.1 & 1 \end{smallmatrix}\right)$, which has the same eigenvectors as $\mathbf{S}$ and the eigenvalues 1.1 and 0.9, from a random start. Plot the sine of the angle to $\mathbf{v}_1$ and the error of the Rayleigh quotient on a logarithmic scale for 30 steps. Predict both slopes and the number of steps each error needs to shrink by a factor of 100, and explain why this matrix converges so much more slowly than $\mathbf{S}$.

3. code **Pseudospectra and transient growth.** ● Let $\mathbf{M}$ be the non-normal matrix Equation 33.41 and $\Lambda = \operatorname{diag}(0.9, 0.8)$ the normal matrix with the same eigenvalues, as in Section 33.4.2.3.

   1. Prove the equality of the two sets in Equation 33.43. First show that $\|(z\mathbf{I} - \mathbf{A})^{-1}\|_2 = 1/\sigma_{\min}(z\mathbf{I} - \mathbf{A})$, where $\sigma_{\min}(\mathbf{X}) = \min_{\|\mathbf{x}\|=1} \|\mathbf{X}\mathbf{x}\|$. For one inclusion, take unit vectors $\mathbf{u}$, $\mathbf{v}$ with $(z\mathbf{I} - \mathbf{A})\mathbf{v} = \sigma\mathbf{u}$, $\sigma = \sigma_{\min}(z\mathbf{I} - \mathbf{A})$, and show that z is an eigenvalue of $\mathbf{A} + \sigma\mathbf{u}\mathbf{v}^*$. For the other, apply $(z\mathbf{I} - \mathbf{A})\mathbf{x} = \mathbf{E}\mathbf{x}$ to an eigenvector $\mathbf{x}$ of $\mathbf{A} + \mathbf{E}$. Using the resolvent norms printed in Section 33.4.2.4, how large a perturbation makes $z = 1$ an eigenvalue of a perturbed $\mathbf{M}$, and how large for Λ?
   2. Suppose that $\|\mathbf{A}^k\|_2 \le K_0$ for all $k \ge 0$. Show that for $|z| > 1$ the series $\sum_{k \ge 0} \mathbf{A}^k / z^{k+1}$ converges to $(z\mathbf{I} - \mathbf{A})^{-1}$, and conclude that

   $$\sup_{k \ge 0} \|\mathbf{A}^k\|_2 \;\ge\; K(\mathbf{A}) = \sup_{|z|>1} \left(|z| - 1\right) \|(z\mathbf{I} - \mathbf{A})^{-1}\|_2 .$$

   The quantity $K(\mathbf{A})$ is called the Kreiss constant. Restate the bound in terms of pseudospectra: if $\Lambda_\varepsilon(\mathbf{A})$ contains a point at distance δ outside the unit circle, how large must some power of $\mathbf{A}$ be?

3. Evaluate $K(\mathbf{M})$ numerically on a grid of points z with $|z| > 1$, and compare it with the largest $\|\mathbf{M}^k\|_2$ printed in Section 33.4.2.3. Find $K(\Lambda)$ by hand.

4. code **Damping in PageRank.** ● Let $\mathbf{P}$ be a column-stochastic $n \times n$ matrix and $\mathbf{G} = \alpha\mathbf{P} + (1-\alpha)\frac{1}{n}\mathbf{1}\mathbf{1}^\top$ with $0 < \alpha < 1$, the damped matrix of Section 33.4.1.3.
   1. Let $\mathbf{P}\mathbf{x} = \lambda\mathbf{x}$. Show that $\mathbf{1}^\top\mathbf{x} = 0$ whenever $\lambda \neq 1$, and that $\mathbf{G}\mathbf{x} = \alpha\lambda\mathbf{x}$ whenever $\mathbf{1}^\top\mathbf{x} = 0$.
   2. The eigenvalue 1 of $\mathbf{P}$ can be repeated, as it is when the links split into disconnected pockets. You may use that every column-stochastic $\mathbf{P}$ has a non-negative eigenvector for the eigenvalue 1 (a stationary distribution). Show that the vectors $\mathbf{x}$ of this eigenspace with $\mathbf{1}^\top\mathbf{x} = 0$ form a subspace of dimension one less than the eigenspace. Conclude that for diagonalizable $\mathbf{P}$ the eigenvalues of $\mathbf{G}$ are 1 and $\alpha\lambda_2(\mathbf{P}), \dots, \alpha\lambda_n(\mathbf{P})$, which gives the bound $|\lambda_2(\mathbf{G})| \le \alpha$ of Haveliwala and Kamvar (2003) .
   3. Four pages link in a cycle, $1 \to 2 \to 3 \to 4 \to 1$. Find the eigenvalues of their link matrix $\mathbf{P}$. Why does undamped power iteration started at page 1 never converge?
   4. With $\alpha = 0.85$ and the cycle of (c), run power iteration on $\mathbf{G}$ from page 1, and count the iterations until the ℓ_1 distance to the stationary distribution falls below 10^{-8}. Compare with the prediction $0.85^k \le 10^{-8}$ from (b).
   5. Check numerically on the three-page example of Section 33.4.1.3 that the eigenvalues of $\mathbf{G}$ are α times those of $\mathbf{P}$, apart from the eigenvalue 1, and that the stationary distribution moves from $(0.3, 0.4, 0.3)$ to about $(0.303, 0.394, 0.303)$. Why does damping move it toward uniform?

5. code **Lyapunov exponents of random products.** ●● Let $\mathbf{D}_1, \mathbf{D}_2, \dots$ be independent $n \times n$ matrices with independent $\mathcal{N}(0, 1/n)$ entries, a linear model of the Jacobians of a deep network at fan-in initialization, and let $\mathbf{x}_0$ be a unit vector. By Section 33.4.2.6, each factor has spectral radius close to 1 for large n. Section 33.4.2.5 summarizes the growth of the product by its top Lyapunov exponent

$$\mu = \lim_{T \to \infty} \frac{1}{T} \log \|\mathbf{D}_T \cdots \mathbf{D}_1 \mathbf{x}_0\|,$$

which exists with probability one by the theorem of Furstenberg and Kesten.

   1. Show that $\mathbb{E}\|\mathbf{D}\mathbf{x}\|^2 = \|\mathbf{x}\|^2$ for every fixed $\mathbf{x}$, so that this scaling preserves squared norms on average.
   2. Let $\mathbf{u}_0 = \mathbf{x}_0$ and $\mathbf{u}_t = \mathbf{x}_t / \|\mathbf{x}_t\|$ for the iterates $\mathbf{x}_t = \mathbf{D}_t \cdots \mathbf{D}_1 \mathbf{x}_0$. Given $\mathbf{D}_1, \dots, \mathbf{D}_{t-1}$, what is the distribution of $n\|\mathbf{D}_t \mathbf{u}_{t-1}\|^2$? Use the answer to write $\log \|\mathbf{D}_T \cdots \mathbf{D}_1 \mathbf{x}_0\|$ as a sum of independent terms. With $\mathbb{E} \log \chi^2_n = \psi(n/2) + \log 2$, where ψ is the digamma function, derive the closed form of Cohen and Newman (1984)

$$\mu = \tfrac{1}{2}\big(\psi(n/2) + \log 2 - \log n\big).$$

   3. Show that $\mu < 0$ for every n, although (a) holds. Evaluate μ in closed form for $n = 1$, using $\psi(1/2) = -\gamma_{\mathrm{E}} - 2\log 2$ with Euler's constant $\gamma_{\mathrm{E}} \approx 0.5772$. Using $\psi(x) = \log x - \frac{1}{2x} + O(x^{-2})$, show that $\mu \approx -1/(2n)$ for large n.

4. Simulate $T = 10^4$ steps with renormalization for $n \in \{1, 2, 5, 50\}$, and compare the measured exponent with (b) and with the logarithms of the spectral radius and of the largest singular value of single factors. By which factor does a typical input shrink in a linear network of width 50 and depth 100 at this initialization? What does the comparison say about the three quantities of Section 33.4.2.5?

33.5 Singular Value Decomposition and Low-Rank Approximation

In Section 33.3, we saw how the eigendecomposition describes a square matrix in terms of the directions it merely stretches, turning powers of a matrix into simple powers of numbers. Most matrices in deep learning, however, are not like that. A weight matrix maps inputs of one width to outputs of another, and a data matrix has one row per example and one column per feature. Neither is applied repeatedly to its own output, and neither needs to be square. Even when a matrix *is* square, eigenvalues can still let us down. The shear $\mathbf{T} = \left(\begin{smallmatrix} 1 & 1 \\ 0 & 1 \end{smallmatrix}\right)$ is *defective*, having only a single eigenvector direction where the plane needs two, and in Section 33.4 we met a matrix whose sixth power stretches a vector ten-fold even though both of its eigenvalues are smaller than 1. We need a factorization that exists for *every* matrix, measures stretching directly, and tells us how few numbers are needed to approximate a large matrix well.

A concrete photograph makes that last requirement easy to appreciate. A grayscale picture of a cat, 400×500 pixels, is simply a matrix of 200,000 numbers. Like any matrix, it can be written as a sum of rank-one pieces, each formed by the outer product of a single column and a single row. Remarkably, just twenty well-chosen pieces (18,020 numbers, or 9% of the original data) reproduce the photograph with a relative error of only 8% (Figure 33.28). Which twenty pieces achieve this, and could any other twenty do better? That same question arises across deep learning: it tells us how many parameters a fine-tuning adapter for a large model needs, and which directions in a dataset are actually worth keeping.

The key idea is to stop asking a single basis to serve for both the inputs and the outputs. If we allow one orthonormal basis for the input space and a separate orthonormal basis for the output space, then *every* matrix reduces to a list of independent non-negative stretches $\sigma_1 \geq \sigma_2 \geq \cdots \geq 0$. From that single list we can read off a matrix's rank, its best low-rank approximation, principal component analysis, least-squares solutions, and how sensitive a linear system is to measurement error. Recall that Section 1.3 introduced two ways to measure the size of a matrix: the Frobenius norm $\|\mathbf{A}\|_F$ (the square root of the sum of squared entries) and the spectral norm $\|\mathbf{A}\|_2$ (the largest factor by which $\mathbf{A}$ can stretch a unit vector). In this section, both become simple functions of the singular values σ_i.

Two main results anchor what follows: the existence of the singular value decomposition for any matrix, and the Eckart–Young theorem, which guarantees that truncating the decomposition gives the best approximation of every rank. From there, the pseudoinverse and the condition number follow in just a few lines. The final subsection connects these tools to modern deep learning and can be skipped on a first reading. All numerical experiments below use NumPy; throughout, $\mathbf{T}$ denotes the shear matrix and $\mathbf{A}$ denotes a general matrix unless stated otherwise.

33.5.1 The Singular Value Decomposition

33.5.1.1 The Factorization and Its Geometry

In Section 1.3, we saw that the symmetric matrix $\left(\begin{smallmatrix} 2 & 1 \\ 1 & 2 \end{smallmatrix}\right)$ maps the unit circle to an ellipse whose semi-axes lie along its eigenvectors $(1,1)$ and $(-1,1)$, with half-lengths 3 and 1. Remarkably, a *non-symmetric* matrix also maps the unit circle to an ellipse (or flattens it to a line segment if the matrix is singular), as we will prove shortly. The difference is that the orthogonal input directions landing on the axes of this ellipse are no longer the axes themselves. Instead, there are two orthonormal frames, one for the input space and one for the output space, and the singular value decomposition names both.

Theorem (singular value decomposition). *Every real $m \times n$ matrix $\mathbf{A}$ can be written as*

$$\mathbf{A} = \mathbf{U}\,\Sigma\,\mathbf{V}^{\top}, \tag{33.44}$$

where $\mathbf{U} \in \mathbb{R}^{m\times m}$ and $\mathbf{V} \in \mathbb{R}^{n\times n}$ are orthogonal ($\mathbf{U}^{\top}\mathbf{U} = \mathbf{I}$, $\mathbf{V}^{\top}\mathbf{V} = \mathbf{I}$) and $\Sigma \in \mathbb{R}^{m\times n}$ is zero except for the non-negative entries $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_{\min(m,n)} \geq 0$ on its main diagonal. The columns $\mathbf{v}_i$ of $\mathbf{V}$ are the *right singular vectors*, the columns $\mathbf{u}_i$ of $\mathbf{U}$ are the *left singular vectors*, and the diagonal entries σ_i are the *singular values*.

In geometric terms, *every* matrix, regardless of its shape, is simply an orthogonal transformation, followed by a diagonal stretch, followed by a second orthogonal transformation. Writing $\mathbf{AV} = \mathbf{U}\Sigma$ and reading off column i makes the action explicit one direction at a time:

$$\mathbf{A}\mathbf{v}_i = \sigma_i\,\mathbf{u}_i, \qquad i \leq \min(m,n). \tag{33.45}$$

Each input direction $\mathbf{v}_i$ is sent to the corresponding output direction $\mathbf{u}_i$, stretched by σ_i; when $n > m$, the extra input directions $\mathbf{v}_{m+1}, \dots, \mathbf{v}_n$ are collapsed to zero. Reading Equation 33.44 from right to left, applying $\mathbf{A}$ to a vector $\mathbf{x}$ unfolds in three stages:

1. $\mathbf{V}^{\top}$ rotates the input so that each right singular vector $\mathbf{v}_i$ aligns with the coordinate axis $\mathbf{e}_i$;
2. Σ stretches axis i by σ_i and, when $m \neq n$, either appends zero coordinates or drops excess ones to match the output dimension;
3. $\mathbf{U}$ rotates the stretched axes so that axis i points along $\mathbf{u}_i$.

Here "rotation" is convenient shorthand for an orthogonal map, which may also include a reflection.

Figure 33.24 illustrates these three stages for the shear $\mathbf{T}$. Its singular values are $\sigma_1 = 1.618$ and $\sigma_2 = 0.618$; the first right singular vector $\mathbf{v}_1$ points at 58.3° above the x-axis, while its image direction $\mathbf{u}_1$ points at 31.7°. Because the input and output frames differ, $\mathbf{U} \neq \mathbf{V}$. We will derive these exact numbers in Section 33.5.1.2.

In n dimensions, the same three stages deform the unit sphere into an ellipsoid whose semi-axes are the vectors $\sigma_i\mathbf{u}_i$. If only $r < n$ of the input directions have positive singular values, the remaining $n - r$ directions collapse to zero, and the image is a flat r-dimensional ellipsoid lying inside the span of $\mathbf{u}_1, \dots, \mathbf{u}_r$.

Compact and thin forms. Let r be the number of positive singular values (in Section 33.5.1.5 we will verify that r is precisely the rank of $\mathbf{A}$). Every row and column of Σ beyond the

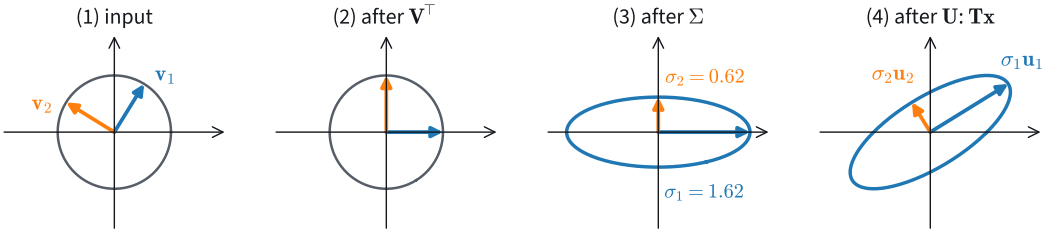


Figure 33.24: The shear $\mathbf{T} = \left(\begin{smallmatrix} 1 & 1 \\ 0 & 1 \end{smallmatrix}\right) = \mathbf{U}\Sigma\mathbf{V}^\top$ acting on the unit circle, read left to right: $\mathbf{V}^\top$ rotates by $-58.3°$, Σ stretches by $\varphi = 1.618$ along x and by $1/\varphi = 0.618$ along y, and $\mathbf{U}$ rotates by $31.7°$. The input frame $\{\mathbf{v}_i\}$ and the output frame $\{\mathbf{u}_i\}$ differ.

first r is zero, so the singular vectors paired with them contribute nothing to the product. Dropping those zero entries yields the *compact SVD*, $\mathbf{A} = \mathbf{U}_r\Sigma_r\mathbf{V}_r^\top$, where $\mathbf{U}_r \in \mathbb{R}^{m\times r}$ and $\mathbf{V}_r \in \mathbb{R}^{n\times r}$ hold the first r singular vectors and $\Sigma_r = \operatorname{diag}(\sigma_1, \dots, \sigma_r)$. Between the full and compact forms sits the *thin SVD*, the default reduced form that numerical libraries return on request, which keeps $\min(m, n)$ columns in $\mathbf{U}$ and $\mathbf{V}$ and still records any zero singular values when $r < \min(m, n)$. For a 3×2 matrix of rank 2, the full factors have shapes 3×3, 3×2, and 2×2, whereas the thin factors have shapes 3×2, 2×2, and 2×2 (matching the compact SVD here because the matrix has full column rank). The full and thin forms differ whenever the matrix is not square, and the thin and compact forms whenever it is rank-deficient. Figure 33.25 draws a 6×4 matrix of rank 3, for which all three differ.

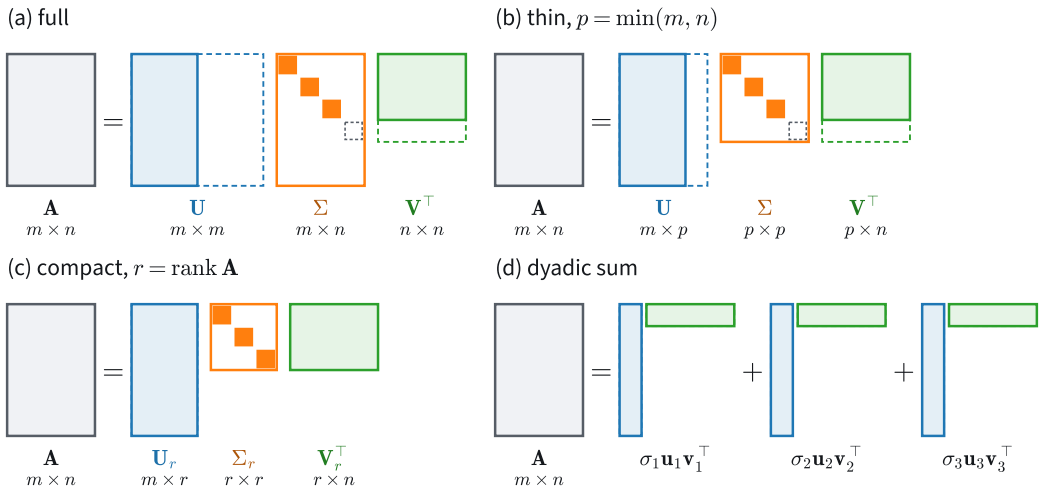


Figure 33.25: Shapes of the SVD factors of an $m \times n$ matrix of rank r, drawn to scale for $m = 6$, $n = 4$, and $r = 3$: (a) full, (b) thin, (c) compact, and (d) the dyadic sum of r rank-one terms. Filled squares on the diagonal of Σ are positive singular values, and the dashed square is a zero singular value. The dashed parts of $\mathbf{U}$ and $\mathbf{V}^\top$ multiply only zeros of Σ, so each smaller form drops some of them without changing the product.

The dyadic sum. Multiplying out the compact SVD column by row expresses $\mathbf{A}$ as a weighted sum of rank-one matrices, often called *dyads*:

$$\mathbf{A} = \sum_{i=1}^{r} \sigma_i \, \mathbf{u}_i \mathbf{v}_i^\top . \tag{33.46}$$

Each term $\mathbf{u}_i \mathbf{v}_i^\top$ is the outer product of an output direction and an input direction, scaled by its singular value, and the terms are already sorted from heaviest to lightest. For the shear, $\mathbf{T} = \varphi \, \mathbf{u}_1 \mathbf{v}_1^\top + \varphi^{-1} \, \mathbf{u}_2 \mathbf{v}_2^\top$, splitting the matrix into one piece of weight 1.618 and a second of weight 0.618. Keeping only the heaviest pieces is the idea behind low-rank approximation in Section 33.5.2.1.

33.5.1.2 Existence

Where do the σ_i, $\mathbf{v}_i$, and $\mathbf{u}_i$ come from, and why should the output directions come out orthonormal? The spectral theorem of Section 33.3.2.1 guarantees that every symmetric matrix has an orthonormal basis of eigenvectors with real eigenvalues. We can apply it to the *Gram matrix* $\mathbf{A}^\top \mathbf{A}$, the $n \times n$ matrix of dot products between the columns of $\mathbf{A}$. This matrix is symmetric, and it is positive semidefinite (Section 33.3.2.2) because $\mathbf{x}^\top \mathbf{A}^\top \mathbf{A} \mathbf{x} = \|\mathbf{A}\mathbf{x}\|^2 \geq 0$ for every vector $\mathbf{x}$.

Proof of Equation 33.44 (existence). By the spectral theorem, $\mathbf{A}^\top \mathbf{A}$ has an orthonormal basis of eigenvectors $\mathbf{v}_1, \dots, \mathbf{v}_n$ with eigenvalues $\lambda_1 \geq \cdots \geq \lambda_n$. Every eigenvalue is nonnegative because $\lambda_i = \mathbf{v}_i^\top \mathbf{A}^\top \mathbf{A} \mathbf{v}_i = \|\mathbf{A}\mathbf{v}_i\|^2$. Define

$$\sigma_i = \sqrt{\lambda_i}, \qquad r = \#\{i : \sigma_i > 0\}.$$

Thus the singular values are the square roots of the eigenvalues of the Gram matrix, and r counts the positive ones. For each $i \leq r$, set $\mathbf{u}_i = \mathbf{A}\mathbf{v}_i / \sigma_i$. The crucial check is that these pushed-through vectors are automatically orthonormal, which follows from $\mathbf{v}_j$ being an eigenvector of $\mathbf{A}^\top \mathbf{A}$ and the $\mathbf{v}_i$ being orthonormal:

$$\mathbf{u}_i^\top \mathbf{u}_j = \frac{1}{\sigma_i \sigma_j} \mathbf{v}_i^\top \mathbf{A}^\top \mathbf{A} \, \mathbf{v}_j = \frac{\lambda_j}{\sigma_i \sigma_j} \mathbf{v}_i^\top \mathbf{v}_j = \frac{\lambda_j}{\sigma_i \sigma_j} \delta_{ij} = \delta_{ij}.$$

Orthonormality of the input frame therefore passes directly through $\mathbf{A}^\top \mathbf{A}$ to the output frame. Next we check the dimensions. Since the r orthonormal vectors $\mathbf{u}_1, \dots, \mathbf{u}_r$ lie in $\mathbb{R}^m$, we have $r \leq m$, and since there are n eigenvalues in total, $r \leq n$; hence $r \leq \min(m, n)$, and $\lambda_i = 0$ for every $i > r$. Complete $\mathbf{u}_1, \dots, \mathbf{u}_r$ to an orthonormal basis $\mathbf{u}_1, \dots, \mathbf{u}_m$ of $\mathbb{R}^m$ (for example, by applying Gram–Schmidt to any completion). For $r < i \leq n$, $\|\mathbf{A}\mathbf{v}_i\|^2 = \lambda_i = 0$, so $\mathbf{A}\mathbf{v}_i = \mathbf{0}$. Comparing $\mathbf{AV}$ with $\mathbf{U}\Sigma$ column by column, column i of $\mathbf{U}\Sigma$ is $\sigma_i \mathbf{u}_i$ for $i \leq \min(m, n)$ (which vanishes when $i > r$) and $\mathbf{0}$ for $i > m$ (where Σ has no diagonal entry). In every case, column i of $\mathbf{AV}$ matches column i of $\mathbf{U}\Sigma$. Hence $\mathbf{AV} = \mathbf{U}\Sigma$, and right-multiplying by $\mathbf{V}^\top = \mathbf{V}^{-1}$ yields $\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^\top$. ■

This proof is also a recipe: diagonalize the symmetric matrix $\mathbf{A}^\top \mathbf{A}$, take square roots to get the singular values, and map the eigenvectors through $\mathbf{A}$ to obtain the left singular vectors.

Let us work through the shear by hand to see every step in action. For $\mathbf{T} = \left(\begin{smallmatrix} 1 & 1 \\ 0 & 1 \end{smallmatrix}\right)$, the Gram matrix is $\mathbf{T}^\top \mathbf{T} = \left(\begin{smallmatrix} 1 & 1 \\ 1 & 2 \end{smallmatrix}\right)$. It has trace 3 and determinant 1, so its characteristic polynomial is $\lambda^2 - 3\lambda + 1$, giving eigenvalues $(3 \pm \sqrt{5})/2$. Introducing the golden ratio $\varphi = (1 + \sqrt{5})/2$,

which satisfies $\varphi^2 = \varphi + 1$, these eigenvalues simplify neatly to φ^2 and φ^{-2}. The singular values are therefore $\sigma_1 = \varphi \approx 1.618$ and $\sigma_2 = 1/\varphi \approx 0.618$, and their product is $1 = |\det \mathbf{T}|$.

Now we find the singular vectors. For the first right singular vector (x, y), the first row of $(\mathbf{T}^\top\mathbf{T} - \varphi^2\mathbf{I})(x, y) = \mathbf{0}$ reads $(1 - \varphi^2)x + y = 0$; using $\varphi^2 - 1 = \varphi$ gives $y = \varphi x$, so $\mathbf{v}_1 \propto (1, \varphi)$. Applying $\mathbf{T}$ yields $\mathbf{T}(1, \varphi) = (1 + \varphi, \varphi) = \varphi\,(\varphi, 1)$, so $\mathbf{u}_1 \propto (\varphi, 1)$. Both vectors have length $\sqrt{1 + \varphi^2} = \sqrt{\varphi + 2}$, giving $\mathbf{v}_1 \approx (0.526, 0.851)$ at angle $\arctan\varphi = 58.3^\circ$ and $\mathbf{u}_1 \approx (0.851, 0.526)$ at 31.7°. The second pair follows by orthogonality, $\mathbf{v}_2 = (-\varphi, 1)/\sqrt{\varphi + 2}$ and $\mathbf{u}_2 = (-1, \varphi)/\sqrt{\varphi + 2}$, and indeed $\mathbf{T}\mathbf{v}_2 = (1 - \varphi, 1)/\sqrt{\varphi + 2} = \varphi^{-1}\mathbf{u}_2$ since $(1 - \varphi)\varphi = -1$. Both $\mathbf{V}$ and $\mathbf{U}$ are rotations $\mathbf{R}_\theta = \left(\begin{smallmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{smallmatrix}\right)$, so the shear factors as

$$\mathbf{T} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} = \mathbf{R}_{31.7^\circ} \begin{bmatrix} \varphi & 0 \\ 0 & 1/\varphi \end{bmatrix} \mathbf{R}_{58.3^\circ}^\top. \tag{33.47}$$

Geometrically, the shear rotates the plane by -58.3°, stretches it by 1.618 along the x-axis and 0.618 along the y-axis, and rotates it back by only 31.7°. The two angles sum to 90° because $\mathbf{u}_1$ is simply $\mathbf{v}_1$ with its coordinates swapped, reflecting the symmetry $\mathbf{T}^\top = \mathbf{J}\mathbf{T}\mathbf{J}$ for the swap matrix $\mathbf{J} = \left(\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right)$.

Does NumPy's `np.linalg.svd` agree with our hand calculation? Because singular vectors are only determined up to sign (Section 33.5.1.3), the cell below stores the shear as `A` and first fixes signs so that $\mathbf{u}_1$ lies in the first quadrant and $\mathbf{U}$ is a proper rotation.

```
A = np.array([[1., 1.], [0., 1.]])
U, s, Vt = np.linalg.svd(A)
if U[0, 0] < 0:                          # joint sign choice: u_1 in the first quadrant
    U[:, 0], Vt[0] = -U[:, 0], -Vt[0]
if np.linalg.det(U) < 0:                 # joint sign choice: U a rotation (then so is V)
    U[:, 1], Vt[1] = -U[:, 1], -Vt[1]
phi = (1 + np.sqrt(5)) / 2
print('singular values', s.round(4), '  phi, 1/phi', np.array([phi, 1 / phi]).round(4))
print('U =', U.round(3).tolist(), '  Vt =', Vt.round(3).tolist())
print('angles of u_1, v_1 (degrees):', np.degrees(np.arctan2(U[1, 0], U[0, 0])).round(1),
      np.degrees(np.arctan2(Vt[0, 1], Vt[0, 0])).round(1))
print('A v_1 = sigma_1 u_1:', np.allclose(A @ Vt[0], s[0] * U[:, 0]),
      '  sigma_1 sigma_2 =', s.prod().round(6))
```

```
singular values [1.618 0.618]   phi, 1/phi [1.618 0.618]
U = [[0.851, -0.526], [0.526, 0.851]]   Vt = [[0.526, 0.851], [-0.851, 0.526]]
angles of u_1, v_1 (degrees): 31.7 58.3
A v_1 = sigma_1 u_1: True   sigma_1 sigma_2 = 1.0
```

It matches: the library returns φ and $1/\varphi$ together with the two rotations of Equation 33.47.

The same construction works unchanged for a rectangular matrix. Consider the 3×2 matrix with rows $(3, 1)$, $(1, 3)$, and $(0, 2)$. Its Gram matrix is $\left(\begin{smallmatrix} 10 & 6 \\ 6 & 14 \end{smallmatrix}\right)$, with eigenvalues $12 \pm \sqrt{40}$. The next cell carries out the proof's three steps directly (eigenvectors of $\mathbf{A}^\top\mathbf{A}$, square roots, and $\mathbf{u}_i = \mathbf{A}\mathbf{v}_i/\sigma_i$) and compares the result against `np.linalg.svd`.

```
A = np.array([[3., 1.], [1., 3.], [0., 2.]])
lam, V = np.linalg.eigh(A.T @ A)
lam, V = lam[::-1], V[:, ::-1]                    # descending order
sigma = np.sqrt(lam)
U = A @ V / sigma                                  # u_i = A v_i / sigma_i
U_lib, s_lib, Vt_lib = np.linalg.svd(A, full_matrices=False)
print('sigma from eigh:', sigma.round(5), '  from svd:', s_lib.round(5))
print('U orthonormal:', np.allclose(U.T @ U, np.eye(2)), '  same U, V up to sign:',
      np.allclose(abs(U), abs(U_lib)), np.allclose(abs(V), abs(Vt_lib.T)))
print('rebuild error:', np.linalg.norm((U * sigma) @ V.T - A).round(12))
```

```
sigma from eigh: [4.28072 2.38232]   from svd: [4.28072 2.38232]
U orthonormal: True   same U, V up to sign: True True
rebuild error: 0.0
```

Both routes yield $\sigma_1 = \sqrt{12+\sqrt{40}} \approx 4.281$ and $\sigma_2 = \sqrt{12-\sqrt{40}} \approx 2.382$. As the proof promised, the pushed-through vectors $\mathbf{u}_i$ come out orthonormal, and both $\mathbf{U}$ and $\mathbf{V}$ agree with the library's output up to sign.

33.5.1.3 Uniqueness and the Largest Stretch

Is the SVD of a matrix unique? The singular *values* are always unique, because they are the square roots of the eigenvalues of $\mathbf{A}^\top\mathbf{A}$, which $\mathbf{A}$ determines completely. The singular *vectors*, by contrast, have three kinds of freedom:

- For a simple (unrepeated) $\sigma_i > 0$, the only choice is a joint sign flip $(\mathbf{u}_i, \mathbf{v}_i) \to (-\mathbf{u}_i, -\mathbf{v}_i)$. Both vectors must flip together, because flipping only one side would violate $\mathbf{A}\mathbf{v}_i = \sigma_i \mathbf{u}_i$.
- For a repeated singular value $\sigma > 0$ with index set $\mathcal{J}$, the columns $\mathbf{V}_\mathcal{J}$ may be rotated by any orthogonal matrix $\mathbf{Q}$ of matching size, becoming $\mathbf{V}_\mathcal{J}\mathbf{Q}$, provided $\mathbf{U}_\mathcal{J}$ is rotated by the *same* $\mathbf{Q}$ to preserve $\mathbf{U}_\mathcal{J} = \mathbf{A}\mathbf{V}_\mathcal{J}/\sigma$.
- For $\sigma = 0$, the null-space vectors $\mathbf{v}_{r+1}, \dots, \mathbf{v}_n$ and left-null-space vectors $\mathbf{u}_{r+1}, \dots, \mathbf{u}_m$ can be rotated completely independently, since they do not appear in the dyadic sum Equation 33.46.

This freedom explains why two numerical libraries, or even the same library on two different machines, may return opposite signs for the singular vectors, and why our code cells fix a sign convention before printing vectors.

Beyond factoring $\mathbf{A}$, the top singular value has a direct geometric meaning: it is the largest factor by which $\mathbf{A}$ can stretch any unit vector,

$$\sigma_1 = \max_{\|\mathbf{x}\|=1} \|\mathbf{A}\mathbf{x}\| = \|\mathbf{A}\|_2. \tag{33.48}$$

This follows immediately from the Rayleigh proposition of Section 33.3.2.3, which states that over unit vectors $\mathbf{x}$, the quadratic form $\mathbf{x}^\top\mathbf{S}\mathbf{x}$ of a symmetric matrix $\mathbf{S}$ is maximized by its top eigenvector with value equal to its top eigenvalue. Taking $\mathbf{S} = \mathbf{A}^\top\mathbf{A}$, we see that $\|\mathbf{A}\mathbf{x}\|^2 = \mathbf{x}^\top\mathbf{A}^\top\mathbf{A}\mathbf{x}$ is maximized at $\mathbf{x} = \mathbf{v}_1$ with value $\lambda_1 = \sigma_1^2$. Thus σ_1 is precisely the spectral norm $\|\mathbf{A}\|_2$ from Section 1.3: for the shear, no unit vector is stretched by more than $\varphi = 1.618$, and $\mathbf{v}_1$ achieves that stretch. The remaining singular values solve the same maximization problem once the earlier right singular vectors are ruled out, a procedure

called *deflation*: $\sigma_k = \max\{\|\mathbf{A}\mathbf{x}\| : \|\mathbf{x}\| = 1,\ \mathbf{x} \perp \mathbf{v}_1, \dots, \mathbf{v}_{k-1}\}$. In fact, Exercise 1a reverses this logic and constructs the entire SVD from the maximum stretch alone, without invoking the spectral theorem.

33.5.1.4 Singular Values and Eigenvalues

How do singular values relate to eigenvalues? Substituting Equation 33.44 into the two Gram matrices and using $\mathbf{U}^\top\mathbf{U} = \mathbf{I}$ and $\mathbf{V}^\top\mathbf{V} = \mathbf{I}$ gives

$$\mathbf{A}^\top\mathbf{A} = \mathbf{V}\,\Sigma^\top\Sigma\,\mathbf{V}^\top, \qquad \mathbf{A}\mathbf{A}^\top = \mathbf{U}\,\Sigma\Sigma^\top\,\mathbf{U}^\top. \tag{33.49}$$

Both expressions are symmetric eigendecompositions: the right singular vectors are the eigenvectors of $\mathbf{A}^\top\mathbf{A}$, the left singular vectors are the eigenvectors of $\mathbf{A}\mathbf{A}^\top$, and the squares σ_i^2 are the nonzero eigenvalues that the two Gram matrices share. When $\mathbf{A}$ is symmetric positive semidefinite, its SVD and eigendecomposition coincide, with $\mathbf{U} = \mathbf{V}$.

For any other matrix, however, the two decompositions differ, and it is a common mistake to assume that singular values equal eigenvalue magnitudes. Consider the chapter's running matrix $\mathbf{A} = \left(\begin{smallmatrix} 1 & 2 \\ -1 & 3 \end{smallmatrix}\right)$, whose eigenvalues are $2 \pm i$, both of modulus $\sqrt{5} \approx 2.236$ (Section 33.3). Its Gram matrix $\mathbf{A}^\top\mathbf{A} = \left(\begin{smallmatrix} 2 & -1 \\ -1 & 13 \end{smallmatrix}\right)$ has trace 15 and determinant 25, yielding $\sigma_{1,2}^2 = (15 \pm 5\sqrt{5})/2$ and therefore $\sigma_{1,2} = (5 \pm \sqrt{5})/2 \approx 3.618,\ 1.382$, far from 2.236. Still, one clean identity links the two lists for every square matrix:

$$|\det \mathbf{A}| = \prod_i \sigma_i = \prod_i |\lambda_i|.$$

The first equality holds because orthogonal matrices have determinant ± 1, and the second because the determinant is the product of all eigenvalues. For a 2×2 matrix with a conjugate pair of eigenvalues of common modulus $|\lambda|$, this forces $|\lambda| = \sqrt{\sigma_1\sigma_2}$; here indeed $\sqrt{3.618 \cdot 1.382} = \sqrt{5}$. The identity also recovers the volume interpretation of the determinant from Section 33.2: rotations preserve area, so the diagonal stretch Σ accounts for the entire change of area, and our running matrix maps the unit square to a parallelogram of area $3.618 \cdot 1.382 = 5$.

Computing the SVD in practice. Although our existence proof constructs the SVD from $\mathbf{A}^\top\mathbf{A}$, numerical libraries never form that product. Forming $\mathbf{A}^\top\mathbf{A}$ squares the condition number (Section 33.5.3.2), which can wipe out small singular values in floating-point arithmetic. Instead, libraries apply orthogonal transformations directly to $\mathbf{A}$ to reduce it to bidiagonal form (Golub–Kahan bidiagonalization) and then diagonalize that bidiagonal matrix (Golub and Van Loan 1996).

33.5.1.5 The Four Fundamental Subspaces

Which inputs does $\mathbf{A}$ collapse to zero, and which outputs can it reach? In Section 33.2, we named these sets the null space and the column space, and defined the rank as the number of linearly independent columns. Because $\mathbf{U}$ and $\mathbf{V}$ are invertible, multiplying by them leaves all dimensions unchanged, so $\operatorname{rank}\mathbf{A} = \operatorname{rank}\Sigma$, which is simply the count of nonzero diagonal entries:

$$\operatorname{rank}\mathbf{A} = \#\{i : \sigma_i > 0\} = r. \tag{33.50}$$

In other words, the rank counts the directions that survive without collapsing. Paired with a rounding tolerance (below), counting singular values is the most numerically reliable way to determine rank. It costs more than Gaussian elimination, but it is far more stable: a perturbation $\mathbf{E}$, such as floating-point rounding error, shifts no singular value by more than $\|\mathbf{E}\|_2$ (Weyl's inequality in Section 33.5.2.1). A clear gap between large and tiny singular values can therefore be trusted.

Even better, splitting the singular vectors at the index r gives explicit orthonormal bases for all four fundamental subspaces of $\mathbf{A}$ directly from the dyadic sum Equation 33.46:

- The **column space** (range) of $\mathbf{A}$ is $\operatorname{span}\{\mathbf{u}_1, \dots, \mathbf{u}_r\}$, because every output $\mathbf{Ax} = \sum_{i \le r} \sigma_i(\mathbf{v}_i^\top \mathbf{x})\mathbf{u}_i$ is a linear combination of these vectors, and each one is attained via $\mathbf{u}_i = \mathbf{A}(\mathbf{v}_i/\sigma_i)$.
- The **null space** of $\mathbf{A}$ is $\operatorname{span}\{\mathbf{v}_{r+1}, \dots, \mathbf{v}_n\}$, because by the same formula $\mathbf{Ax} = \mathbf{0}$ holds if and only if $\mathbf{v}_i^\top \mathbf{x} = 0$ for every $i \le r$ (the $\mathbf{u}_i$ are independent and the σ_i are positive), meaning $\mathbf{x}$ is orthogonal to $\mathbf{v}_1, \dots, \mathbf{v}_r$.
- The **row space** (column space of $\mathbf{A}^\top$) is $\operatorname{span}\{\mathbf{v}_1, \dots, \mathbf{v}_r\}$ and the **left null space** (null space of $\mathbf{A}^\top$) is $\operatorname{span}\{\mathbf{u}_{r+1}, \dots, \mathbf{u}_m\}$, by applying the same two arguments to $\mathbf{A}^\top = \mathbf{V}\Sigma^\top \mathbf{U}^\top$, which simply swaps the roles of the $\mathbf{u}_i$ and $\mathbf{v}_i$.

This gives a remarkably clean picture of any linear map. The input space splits into two orthogonal pieces: the r-dimensional row space and the $(n - r)$-dimensional null space. Likewise, the output space splits into the r-dimensional column space and the $(m - r)$-dimensional left null space. On the row space, $\mathbf{A}$ maps each $\mathbf{v}_i$ to $\sigma_i \mathbf{u}_i$, a one-to-one bijection onto the column space; everything in the null space is sent to zero.

To see this in the simplest nontrivial case, consider the singular matrix $\mathbf{B} = \left(\begin{smallmatrix} 2 & 1 \\ 4 & 2 \end{smallmatrix}\right)$ from Section 33.2. Its Gram matrix $\left(\begin{smallmatrix} 20 & 10 \\ 10 & 5 \end{smallmatrix}\right)$ has eigenvalues 25 and 0, so $\sigma_1 = 5$, $\sigma_2 = 0$, and $r = 1$. The four subspaces are the lines spanned by

$$\mathbf{v}_1 = \tfrac{1}{\sqrt{5}}(2, 1), \quad \mathbf{v}_2 = \tfrac{1}{\sqrt{5}}(-1, 2), \quad \mathbf{u}_1 = \tfrac{1}{\sqrt{5}}(1, 2), \quad \mathbf{u}_2 = \tfrac{1}{\sqrt{5}}(-2, 1), \qquad \mathbf{B} = 5\,\mathbf{u}_1\mathbf{v}_1^\top.$$

Here the column space is the line $y = 2x$ along $\mathbf{u}_1$, and the left null space is the perpendicular line along $\mathbf{u}_2$. In the input plane, the null space is the line along $(1, -2)$, perpendicular to the row $(2, 1)$ that spans the row space. Figure 33.26 illustrates the general geometry; as we will see in Section 33.5.3.1, the pseudoinverse simply runs the bijection from the row space to the column space in reverse.

Numerical rank. In floating-point arithmetic, a matrix that is mathematically rank-deficient cannot be relied on to have singular values that are literally zero; rounding error typically leaves tiny values of order $\epsilon_{\text{mach}}\sigma_1$ instead. In practice, rank is therefore decided with a threshold. NumPy's `np.linalg.matrix_rank` counts singular values above $\sigma_1 \max(m, n)\,\epsilon_{\text{mach}}$, a tolerance that scales naturally with both the largest singular value and the matrix dimensions. Let us see what the "zero" singular values of a 4×4 matrix of rank 2 actually look like.

```
A = np.random.standard_normal((4, 2)) @ np.random.standard_normal((2, 4))   # rank <= 2
s = np.linalg.svd(A, compute_uv=False)
tol = s[0] * max(A.shape) * np.finfo(float).eps
print('singular values:', np.array2string(s, formatter={'float_kind': '{:.1e}'.format}))
```

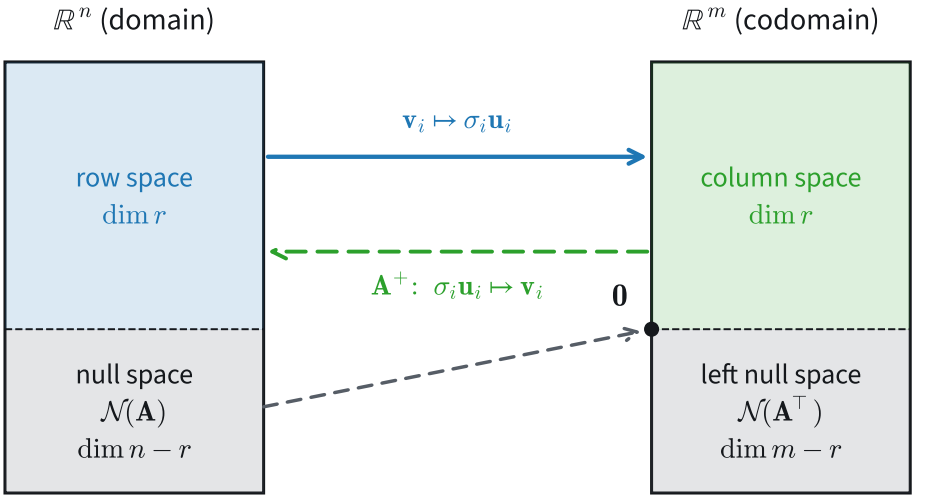


Figure 33.26: The four fundamental subspaces. $\mathbf{A}$ maps the row space $\operatorname{span}\{\mathbf{v}_1, \dots, \mathbf{v}_r\}$ one-to-one onto the column space $\operatorname{span}\{\mathbf{u}_1, \dots, \mathbf{u}_r\}$ by $\mathbf{v}_i \mapsto \sigma_i \mathbf{u}_i$, and maps the null space to zero. The pseudoinverse $\mathbf{A}^+$ (dashed) runs the map backwards, $\sigma_i \mathbf{u}_i \mapsto \mathbf{v}_i$, and sends the left null space to zero.

```
print(f'tolerance {tol:.1e};  exact zeros: {(s == 0).sum()};  '
      f'numerical rank: {np.linalg.matrix_rank(A)}')
```

```
singular values: [4.4e+00 2.2e+00 4.7e-17 0.0e+00]
tolerance 3.9e-15;  exact zeros: 1;  numerical rank: 2
```

Two singular values are of order one, while the remaining two are pure rounding noise: 10^{-15} or smaller, well below the tolerance. Rounding may happen to leave one of them at exactly zero, but it rarely does so for both. Counting exact zeros would therefore misreport the rank, whereas the threshold recovers the true rank of 2.

33.5.2 Low-Rank Approximation

33.5.2.1 Best Low-Rank Approximation

We can now return to the cat photograph from the start of the section: which twenty rank-one pieces should we keep? The dyadic sum Equation 33.46 already orders the pieces of $\mathbf{A}$ from heaviest to lightest, so the natural candidate is to keep the first k terms. This gives the *truncated SVD*,

$$\mathbf{A}_k = \sum_{i=1}^{k} \sigma_i \, \mathbf{u}_i \mathbf{v}_i^\top. \tag{33.51}$$

The matrix $\mathbf{A}_k$ has rank at most k, and storing it requires only k singular values, k columns of length m, and k columns of length n, for a total of $k(m + n + 1)$ numbers instead of

mn. For our 400×500 photograph with $k = 20$, that is 18,020 numbers out of 200,000. Two questions immediately arise: how large is the approximation error, and could some cleverer rank-k matrix do even better?

Matrix norms as singular values. To measure approximation error, we need a matrix norm. The Frobenius norm from Section 1.3, $\|\mathbf{A}\|_F^2 = \sum_{ij} a_{ij}^2$, sums the squared lengths of the columns of $\mathbf{A}$, which are the diagonal entries of $\mathbf{A}^\top\mathbf{A}$; thus $\|\mathbf{A}\|_F^2 = \operatorname{tr}(\mathbf{A}^\top\mathbf{A})$. Substituting the SVD and using the cyclic property of the trace (Section 33.2) to move $\mathbf{V}$ across, we find

$$\|\mathbf{A}\|_F^2 = \operatorname{tr}(\mathbf{V}\Sigma^\top\Sigma\mathbf{V}^\top) = \operatorname{tr}(\Sigma^\top\Sigma\,\mathbf{V}^\top\mathbf{V}) = \sum_i \sigma_i^2. \tag{33.52}$$

The Frobenius norm is therefore the square root of the sum of squared singular values, which immediately tells us that multiplying on either side by an orthogonal matrix never changes it. For the shear, $1 + 1 + 0 + 1 = 3 = \varphi^2 + \varphi^{-2}$. Meanwhile, the spectral norm is simply the largest singular value, $\|\mathbf{A}\|_2 = \sigma_1$, by Equation 33.48.

With both norms expressed in terms of singular values, we can state the central approximation theorem of linear algebra.

Theorem (Eckart–Young–Mirsky). *Let $\sigma_i = 0$ for $i > r$. For every $k \geq 0$, the smallest spectral-norm error attainable by a matrix of rank at most k is*

$$\min_{\operatorname{rank}\mathbf{B}\leq k} \|\mathbf{A} - \mathbf{B}\|_2 = \sigma_{k+1}, \tag{33.53}$$

and the smallest squared Frobenius error is $\min_{\operatorname{rank}\mathbf{B}\leq k} \|\mathbf{A} - \mathbf{B}\|_F^2 = \sum_{i>k} \sigma_i^2$. Both minima are attained at the truncated SVD $\mathbf{B} = \mathbf{A}_k$.

In short, no rank-k matrix can beat the truncated SVD in either norm: the best rank-k approximation misses by the first singular value it omits (in the spectral norm), or by the combined energy of all omitted singular values (in the Frobenius norm). Eckart and Young proved the Frobenius case (Eckart and Young 1936), and Mirsky later extended it to every norm that is invariant under orthogonal multiplication on both sides (Mirsky 1960). We prove the spectral-norm bound first; the Frobenius bound then follows cleanly from one extra ingredient, Weyl's inequality, which drops right out of the same argument.

Before writing the general proof, picture $n = 3$ and $k = 1$. Any matrix $\mathbf{B}$ of rank one collapses an entire plane of inputs to zero, while the top two right singular vectors of $\mathbf{A}$ span another plane. In $\mathbb{R}^3$, two planes through the origin must intersect in at least a line. Along that line, $\mathbf{B}$ outputs zero while $\mathbf{A}$ stretches every vector by at least σ_2, so $\mathbf{A} - \mathbf{B}$ must stretch some unit vector by at least σ_2 (Figure 33.27). The proof below simply carries this dimension count into n dimensions.

Proof (spectral norm). *First, the truncation achieves σ_{k+1}.* The residual $\mathbf{A} - \mathbf{A}_k = \sum_{i>k} \sigma_i \mathbf{u}_i \mathbf{v}_i^\top$ is already in compact SVD form with largest singular value σ_{k+1}, so $\|\mathbf{A} - \mathbf{A}_k\|_2 = \sigma_{k+1}$ by Equation 33.48. Likewise, Equation 33.52 gives $\|\mathbf{A} - \mathbf{A}_k\|_F^2 = \sum_{i>k} \sigma_i^2$.

Second, no matrix of rank at most k can do better. If $k \geq r$ the error is zero and there is nothing to prove, so assume $k < r$ and let $\operatorname{rank}\mathbf{B} \leq k$. By rank–nullity (Section 33.2), the null space $\mathcal{K} = \ker\mathbf{B}$ has dimension at least $n - k$. Meanwhile, the span $\mathcal{V} = \operatorname{span}\{\mathbf{v}_1, \dots, \mathbf{v}_{k+1}\}$ of

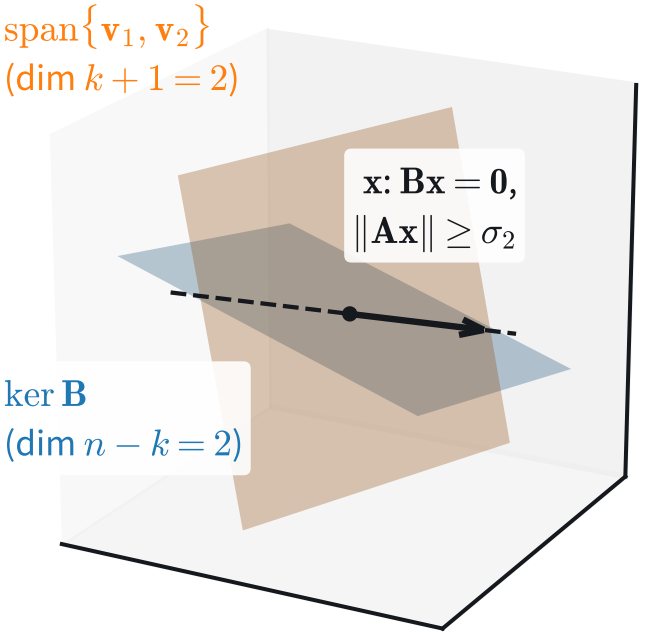


Figure 33.27: The dimension count behind Eckart–Young for $n = 3$, $k = 1$: the plane $\ker \mathbf{B}$ and the plane spanned by the top two right singular vectors must meet in a line. On that line $\mathbf{B}$ vanishes while $\mathbf{A}$ still stretches by at least σ_2.

the top $k+1$ right singular vectors has dimension $k+1$. Because these two dimensions add to more than n, the subspaces must share a nonzero vector:

$$\dim(\mathcal{K} \cap \mathcal{V}) = \dim \mathcal{K} + \dim \mathcal{V} - \dim(\mathcal{K} + \mathcal{V}) \geq (n-k) + (k+1) - n = 1,$$

where the equality is the dimension formula for two subspaces and the inequality uses $\dim(\mathcal{K}+\mathcal{V}) \leq n$. (To verify the dimension formula, extend a basis of $\mathcal{K} \cap \mathcal{V}$ separately to a basis of $\mathcal{K}$ and a basis of $\mathcal{V}$; the combined vectors are linearly independent, span $\mathcal{K}+\mathcal{V}$, and number $\dim \mathcal{K} + \dim \mathcal{V} - \dim(\mathcal{K} \cap \mathcal{V})$.) Pick a unit vector $\mathbf{x} \in \mathcal{K} \cap \mathcal{V}$ and expand it as $\mathbf{x} = \sum_{i \leq k+1} c_i \mathbf{v}_i$ with $\sum_i c_i^2 = 1$. Then $\mathbf{B}\mathbf{x} = \mathbf{0}$, and using $\mathbf{A}\mathbf{v}_i = \sigma_i \mathbf{u}_i$ together with the orthonormality of the $\mathbf{u}_i$,

$$\|\mathbf{A}\mathbf{x}\|^2 = \Big\| \sum_{i \leq k+1} c_i \sigma_i \mathbf{u}_i \Big\|^2 = \sum_{i \leq k+1} c_i^2 \sigma_i^2 \geq \sigma_{k+1}^2 \sum_{i \leq k+1} c_i^2 = \sigma_{k+1}^2.$$

Hence $\|\mathbf{A} - \mathbf{B}\|_2 \geq \|(\mathbf{A}-\mathbf{B})\mathbf{x}\| = \|\mathbf{A}\mathbf{x}\| \geq \sigma_{k+1}$. ■

The geometry is worth repeating in one sentence: any rank-k competitor $\mathbf{B}$ must annihilate at least one direction inside the span of the top $k+1$ right singular vectors, and $\mathbf{A}$ stretches every direction in that span by at least σ_{k+1}. For the shear $\mathbf{T}$ with $k=1$, the truncation $\mathbf{T}_1 = \varphi\, \mathbf{u}_1 \mathbf{v}_1^\top = \frac{1}{\sqrt{5}} \left(\begin{smallmatrix} \varphi & \varphi^2 \\ 1 & \varphi \end{smallmatrix}\right) \approx \left(\begin{smallmatrix} 0.724 & 1.171 \\ 0.447 & 0.724 \end{smallmatrix}\right)$ misses by $1/\varphi = 0.618$ in both norms (the residual has rank one, so its spectral and Frobenius norms coincide), and no rank-one matrix can get closer.

To complete the theorem for the Frobenius norm, we first harvest a general perturbation inequality from the spectral-norm bound we just proved.

Lemma (Weyl's inequality for singular values). *For any two matrices $\mathbf{X}, \mathbf{Y}$ of the same shape and indices $i, j \geq 1$ with $i + j - 1 \leq \min(m, n)$,*

$$\sigma_{i+j-1}(\mathbf{X} + \mathbf{Y}) \leq \sigma_i(\mathbf{X}) + \sigma_j(\mathbf{Y}). \tag{33.54}$$

Proof. Let $\mathbf{X}_{i-1}$ and $\mathbf{Y}_{j-1}$ be the truncated SVDs Equation 33.51 (with $\mathbf{X}_0 = \mathbf{0}$). Their sum $\mathbf{X}_{i-1} + \mathbf{Y}_{j-1}$ has rank at most $i+j-2$, so applying the spectral-norm lower bound to $\mathbf{X}+\mathbf{Y}$ with $k = i+j-2$ gives $\sigma_{i+j-1}(\mathbf{X}+\mathbf{Y}) \leq \|\mathbf{X}+\mathbf{Y}-\mathbf{X}_{i-1}-\mathbf{Y}_{j-1}\|_2$. By the triangle inequality and the achievability half of Eckart–Young, the right-hand side is at most $\|\mathbf{X} - \mathbf{X}_{i-1}\|_2 + \|\mathbf{Y} - \mathbf{Y}_{j-1}\|_2 = \sigma_i(\mathbf{X}) + \sigma_j(\mathbf{Y})$. ■

In words, adding $\mathbf{Y}$ to $\mathbf{X}$ can lift the $(i+j-1)$-th singular value above the i-th singular value of $\mathbf{X}$ by at most $\sigma_j(\mathbf{Y})$. Setting $j = 1$ and applying the inequality in both directions shows that perturbing a matrix by $\mathbf{Y}$ moves no singular value by more than $\|\mathbf{Y}\|_2$ (Horn and Johnson 2012); Exercise 2 works out the details.

Proof (Frobenius norm). Apply Weyl's lemma with $\mathbf{X} = \mathbf{A} - \mathbf{B}$, $\mathbf{Y} = \mathbf{B}$, and $j = k+1$. Because $\operatorname{rank} \mathbf{B} \leq k$, we have $\sigma_{k+1}(\mathbf{B}) = 0$, which yields $\sigma_{k+i}(\mathbf{A}) \leq \sigma_i(\mathbf{A}-\mathbf{B})$ for every $i \geq 1$. Squaring and summing over i, then applying Equation 33.52,

$$\|\mathbf{A}-\mathbf{B}\|_F^2 = \sum_{i \geq 1} \sigma_i(\mathbf{A}-\mathbf{B})^2 \geq \sum_{i \geq 1} \sigma_{k+i}(\mathbf{A})^2 = \sum_{j > k} \sigma_j(\mathbf{A})^2,$$

which the truncation $\mathbf{A}_k$ attains. ■

Uniqueness of the minimizer. One subtle difference separates the two norms. In the spectral norm, the best rank-k approximation is never unique when $k \geq 1$ and $\sigma_{k+1} > 0$: for $\mathbf{A} = \operatorname{diag}(3, 2, 1)$ and $k = 1$, every matrix $\mathbf{B} = \operatorname{diag}(3 + t, 0, 0)$ with $|t| \leq 1$ gives $\|\mathbf{A} - \mathbf{B}\|_2 = \max(|t|, 2, 1) = 2 = \sigma_2$. In the Frobenius norm, by contrast, $\mathbf{A}_k$ is the *unique* minimizer whenever $\sigma_k > \sigma_{k+1}$ or $\sigma_{k+1} = 0$ (in which case $\mathbf{A}_k = \mathbf{A}$) (Golub and Van Loan 1996). This uniqueness is one reason principal component analysis, which we meet next, is formulated in the Frobenius norm.

Let us now test the theorem on our 400×500 cat photograph. Eckart–Young fixes the spectral error of every truncation at σ_{k+1}, but it does not tell us how quickly the singular values of a real image decay, nor how much worse a natural rank-k competitor performs. The cell below loads `img/cat1.jpg`, converts it to grayscale using standard luminance weights for red, green, and blue, and builds the truncations with `np.einsum` (which multiplies its inputs and sums over every index omitted after the arrow; since row i of `Vt` is $\mathbf{v}_i^\top$, `'i,mi,in->mn'` computes the dyadic sum).

```
img = d2l.plt.imread('../img/cat1.jpg') @ np.array([0.299, 0.587, 0.114])  # 400 x 500
U, s, Vt = np.linalg.svd(img, full_matrices=False)
m, n = img.shape
fro = np.linalg.norm(img)
energy = np.cumsum(s**2) / (s**2).sum()                          # energy kept at each rank
slope = np.polyfit(np.log(np.arange(1, 101)), np.log(s[:100]), 1)[0]
print(f'{m} x {n} = {m * n} numbers;  slope of log sigma_i on log i: {slope:.2f}')
print(f'energy kept at rank 1: {energy[0]:.3f}, at rank 4: {energy[3]:.3f}')
for k in [1, 5, 20, 50]:
    A_k = np.einsum('i,mi,in->mn', s[:k], U[:, :k], Vt[:k])    # first k dyads
    print(f'k={k:2d}: ||A-A_k||_2={np.linalg.norm(img - A_k, 2):6.0f}  '
          f'sigma_{k + 1}={s[k]:6.0f}  rel. err {np.linalg.norm(img - A_k) / fro:.3f}  '
          f'stores {k * (m + n + 1) / (m * n):.1%}')
cols = np.argsort(-np.linalg.norm(img, axis=0))[:20]           # 20 largest columns
C = img[:, cols]
proj = C @ np.linalg.pinv(C) @ img
print(f'projection onto the 20 largest columns: rel. err {np.linalg.norm(img - proj) /
↪  fro:.3f}')
```

```
400 x 500 = 200000 numbers;  slope of log sigma_i on log i: -1.11
energy kept at rank 1: 0.886, at rank 4: 0.955
k= 1: ||A-A_k||_2=  9234  sigma_2=  9234  rel. err 0.337  stores 0.5%
k= 5: ||A-A_k||_2=  4048  sigma_6=  4048  rel. err 0.191  stores 2.3%
k=20: ||A-A_k||_2=   874  sigma_21=   874  rel. err 0.080  stores 9.0%
k=50: ||A-A_k||_2=   342  sigma_51=   342  rel. err 0.045  stores 22.5%
projection onto the 20 largest columns: rel. err 0.310
```

The spectral error matches σ_{k+1} at every rank, as guaranteed, and rank 20 stores just 9% of the numbers at an 8% relative error. In Figure 33.28, the cat appears as a bright blur at rank 1, a recognizable silhouette with four paws at rank 5, and unmistakably a cat lying on its back by rank 20. The left panel reveals why so few pieces suffice: over the first hundred indices, the singular values decay roughly as a power law, $\sigma_i \propto i^{-1.1}$.

The energy ratio. The Frobenius error formula gives a practical rule for choosing k. The fraction of total squared "energy" $\sum_i \sigma_i^2$ retained by the rank-k truncation is

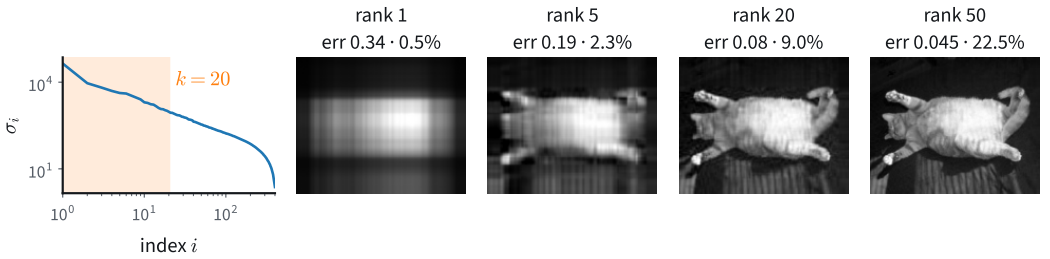


Figure 33.28: Eckart–Young on a 400×500 photograph. Left: the singular values fall roughly as a power law (log–log axes; the shaded band marks the twenty kept in the rank-20 panel). Right: the truncations $\mathbf{A}_k$ with their relative Frobenius error and the fraction $k(m+n+1)/mn$ of the original numbers they store.

$$\frac{\sum_{i \le k} \sigma_i^2}{\sum_i \sigma_i^2} = 1 - \frac{\|\mathbf{A} - \mathbf{A}_k\|_F^2}{\|\mathbf{A}\|_F^2}, \tag{33.55}$$

meaning the relative Frobenius error is the square root of one minus the energy ratio. Retaining 95% of the energy thus corresponds to a relative error of $\sqrt{0.05} \approx 22\%$, while retaining 99% brings the error down to 10%. One must read this ratio with care, however. On the photograph, σ_1^2 alone accounts for 89% of the energy because a constant image has rank one (all rows identical) and the photograph's average brightness is large, so the first dyad mostly captures that mean. Rank 4 already retains 95% of the energy (at 21% relative error), whereas once we subtract the mean pixel value, reaching 95% requires rank 8 instead of 4 (as we verify in Section 33.5.4.3). Always check whether σ_1 is dominated by a nonzero mean, or subtract the mean first, just as principal component analysis does.

To see how much Eckart–Young saves over a naive rank-20 scheme, the last lines of the cell select the 20 actual columns of the photograph with the largest norms and project every column onto their span (the idea behind CUR decompositions, which approximate a matrix using subsets of its own rows and columns). Its relative error is 0.31, nearly four times worse than the 0.08 of the truncated SVD.

Truncation as denoising. Compression is not the only reason to truncate an SVD; denoising is another. Suppose an observed $m \times n$ matrix ($m \le n$) consists of a low-rank signal plus random noise, $\mathbf{A} = \mathbf{B} + \mathbf{N}$, where the entries of $\mathbf{N}$ are independent with mean zero and standard deviation σ_{noise}. By the Marchenko–Pastur law of Section 33.3.2.4 applied to $\mathbf{N}\mathbf{N}^\top/n$, the singular values of a large noise matrix concentrate inside the interval $[(\sqrt{n} - \sqrt{m})\sigma_{\text{noise}}, (\sqrt{n} + \sqrt{m})\sigma_{\text{noise}}]$. Signal components whose singular values rise well above the upper edge stand out cleanly from the noise bulk, while weaker ones disappear into it. For square matrices with a known noise level, Gavish and Donoho proved that the asymptotically optimal hard threshold is $\frac{4}{\sqrt{3}}\sqrt{n}\,\sigma_{\text{noise}} \approx 2.309\sqrt{n}\,\sigma_{\text{noise}}$, sitting slightly above the noise edge $2\sqrt{n}\,\sigma_{\text{noise}}$ (Gavish and Donoho 2014). Exercise 8 derives the constant $4/\sqrt{3}$.

33.5.2.2 Principal Component Analysis

Suppose we are given a dataset whose examples are the rows of $\mathbf{X} \in \mathbb{R}^{n\times d}$, with n data points and d features. Often most of the variation in the data lies along just a few directions. Two natural questions arise: which unit direction captures the largest variance of the data, and which k-dimensional affine subspace approximates the points with the smallest squared error? Surprisingly, both questions have the exact same answer, and it is simply the truncated SVD of the centered data.

Centering. First fix any k-dimensional subspace with orthogonal projector $\mathbf{P}$, and ask which offset $\mathbf{c}$ minimizes the total squared reconstruction error $\sum_i \|(\mathbf{I}-\mathbf{P})(\mathbf{x}_i-\mathbf{c})\|^2$. This objective is a convex quadratic in $\mathbf{c}$, and its gradient $-2(\mathbf{I}-\mathbf{P})\sum_i(\mathbf{x}_i-\mathbf{c})$ vanishes at the sample mean $\mathbf{c}=\bar{\mathbf{x}}$, regardless of $\mathbf{P}$. The best affine subspace therefore always passes through the mean, so we begin by *centering* the data: $\tilde{\mathbf{X}} = \mathbf{X} - \mathbf{1}\bar{\mathbf{x}}^\top$. Its empirical covariance is the symmetric positive semidefinite matrix $\mathbf{C} = \frac{1}{n}\tilde{\mathbf{X}}^\top\tilde{\mathbf{X}}$. (Using the unbiased factor $\frac{1}{n-1}$ rescales every eigenvalue identically and changes neither the principal directions nor the fraction of variance they explain.)

Proposition (PCA via Rayleigh). *Over all unit vectors* $\mathbf{w}$*, the projected variance* $\frac{1}{n}\|\tilde{\mathbf{X}}\mathbf{w}\|^2 = \mathbf{w}^\top\mathbf{C}\mathbf{w}$ *is maximized by the top eigenvector of* $\mathbf{C}$*, which is the top right singular vector* $\mathbf{v}_1$ *of* $\tilde{\mathbf{X}}$*, with maximal variance* σ_1^2/n*.*

Proof. Projecting a centered point onto $\mathbf{w}$ yields the scalar coordinate $\tilde{\mathbf{x}}^\top\mathbf{w}$, whose empirical variance is $\frac{1}{n}\sum_i(\tilde{\mathbf{x}}_i^\top\mathbf{w})^2 = \frac{1}{n}\|\tilde{\mathbf{X}}\mathbf{w}\|^2 = \mathbf{w}^\top\mathbf{C}\mathbf{w}$. By the Rayleigh proposition of Section 33.3.2.3, this quadratic form is maximized by the top eigenvector of $\mathbf{C}$, with value $\lambda_1(\mathbf{C})$. By Equation 33.49, $\mathbf{C} = \frac{1}{n}\mathbf{V}\Sigma^\top\Sigma\mathbf{V}^\top$, so the eigenvectors of $\mathbf{C}$ are the right singular vectors $\mathbf{v}_i$ of $\tilde{\mathbf{X}}$ and its eigenvalues are σ_i^2/n. ■

Among unit vectors orthogonal to $\mathbf{v}_1$, the variance is maximized by $\mathbf{v}_2$, and continuing inductively gives the top k *principal directions* $\mathbf{v}_1,\dots,\mathbf{v}_k$, with component i explaining variance σ_i^2/n. Now check the reconstruction error. Projecting every centered point onto $\operatorname{span}\{\mathbf{v}_1,\dots,\mathbf{v}_k\}$ corresponds to right-multiplying $\tilde{\mathbf{X}}$ by $\mathbf{V}_k\mathbf{V}_k^\top$. Since $\mathbf{V}^\top\mathbf{V}_k$ selects the first k columns of the identity,

$$\tilde{\mathbf{X}}\mathbf{V}_k\mathbf{V}_k^\top = \mathbf{U}\Sigma\mathbf{V}^\top\mathbf{V}_k\mathbf{V}_k^\top = \mathbf{U}_k\Sigma_k\mathbf{V}_k^\top = \tilde{\mathbf{X}}_k.$$

Projecting onto the top k principal directions is therefore identical to taking the rank-k truncated SVD of the centered data. By Eckart–Young in the Frobenius norm, no other projection, and indeed no other rank-k matrix at all, can reconstruct the centered points with smaller squared error. The discarded variance $\sum_{i>k}\sigma_i^2/n$ equals the mean squared reconstruction error, and the explained-variance ratio is simply the energy ratio Equation 33.55 of $\tilde{\mathbf{X}}$.

Figure 33.29 shows a cloud of 300 points with standard deviations 3 and 0.8 along axes rotated by $\pi/5 = 36^\circ$, together with its two principal axes. In the population, the first direction holds $9/9.64 \approx 93.4\%$ of the variance and aligns with the long axis. The cell below checks how accurately a finite sample of 300 points recovers these values.

```
theta = np.pi / 5
rot = np.array([[np.cos(theta), -np.sin(theta)], [np.sin(theta), np.cos(theta)]])
```

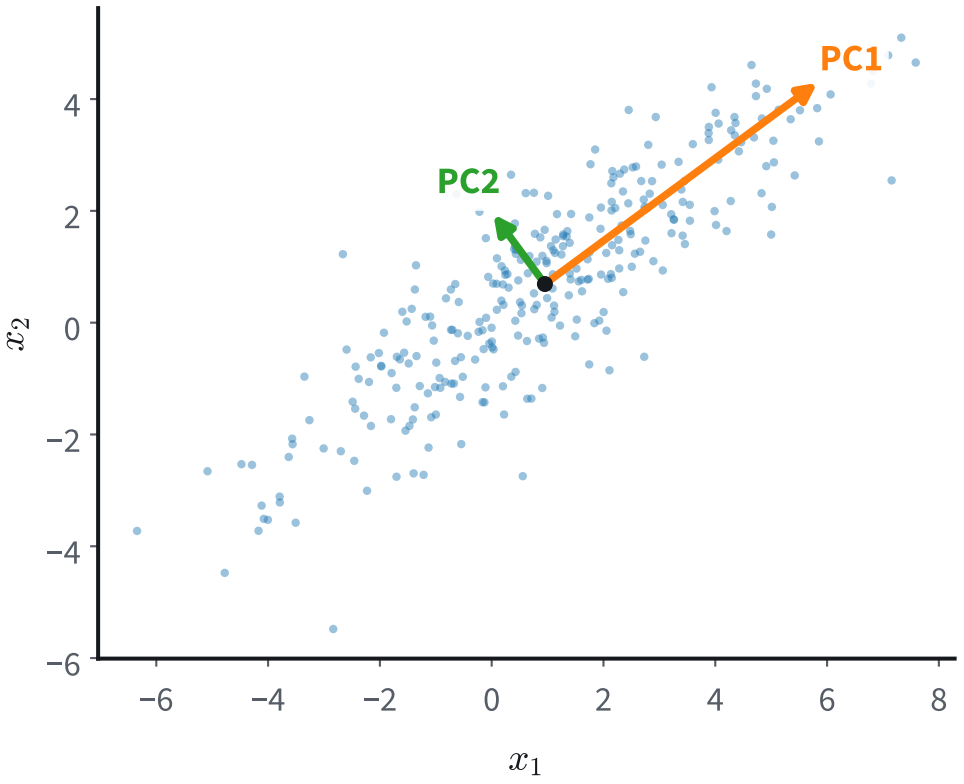


Figure 33.29: A cloud of 300 points centered at $(1, 0.6)$ with standard deviations 3 and 0.8 along axes rotated by $36°$, and its principal axes drawn from the mean, each two standard deviations $2\sigma_i/\sqrt{n}$ long. The first axis points along the direction of largest variance.

```
X = np.random.standard_normal((300, 2)) * np.array([3.0, 0.8]) @ rot.T   # the cloud
Xc = X - X.mean(0)
U, s, Vt = np.linalg.svd(Xc, full_matrices=False)
X1 = np.outer(U[:, 0] * s[0], Vt[0])                                    # rank-1 reconstruction
print('explained-variance ratio:', (s**2 / (s**2).sum()).round(3), ' (population 0.934,
↪  0.066)')
print('angle of v_1 to the long axis:',
      np.degrees(np.arccos(abs(Vt[0] @ rot[:, 0]))).round(1), 'degrees')
print('mean squared rank-1 error:', (((Xc - X1)**2).sum() / 300).round(4),
      '  sigma_2^2 / n:', (s[1]**2 / 300).round(4))
```

```
explained-variance ratio: [0.923 0.077]  (population 0.934, 0.066)
angle of v_1 to the long axis: 0.6 degrees
mean squared rank-1 error: 0.7111   sigma_2^2 / n: 0.7111
```

Across random draws, the first principal direction explains roughly 91% to 95% of the variance and lands within a few degrees of the true long axis. If we forget to center the data, however, the first singular vector swings toward the mean of the cloud as soon as the mean's offset exceeds the spread of the points; it then describes *where* the cloud sits rather than *how* it varies. Exercise 3 quantifies this effect.

33.5.3 Solving Linear Systems with the SVD

33.5.3.1 The Pseudoinverse and Least Squares

A linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$ with more equations than unknowns generally has no exact solution, while one with fewer equations than unknowns has infinitely many. In both situations, we look for the least-squares solution introduced in Section 33.1 (writing the target as $\mathbf{b}$ instead of $\mathbf{y}$),

$$\min_{\mathbf{x}} \|\mathbf{A}\mathbf{x} - \mathbf{b}\|^2,$$

and whenever multiple $\mathbf{x}$ achieve the same minimum residual, we choose the one with the smallest norm.

Consider the simplest underdetermined example first: a single equation in two unknowns, $x_1 + x_2 = 2$. Its solutions form an entire line, and the residual is zero everywhere on it. Which point on the line should we pick? Here $\mathbf{A} = \begin{pmatrix} 1 & 1 \end{pmatrix}$, which has one nonzero singular value $\sigma_1 = \sqrt{2}$, right singular vector $\mathbf{v}_1 = (1,1)/\sqrt{2}$, and $u_1 = 1$. The equation fixes the coordinate of $\mathbf{x}$ along $\mathbf{v}_1$ to $\sqrt{2}$, while the coordinate along the null-space direction $\mathbf{v}_2 = (1,-1)/\sqrt{2}$ does not affect the residual at all. Setting that free coordinate to zero yields $(1,1)$, the foot of the perpendicular from the origin to the line, with norm $\sqrt{2}$; every other solution $(1+t, 1-t)$ has strictly larger squared norm $2 + 2t^2$ (Figure 33.30).

The general rule simply inverts every nonzero singular value and transposes the orthogonal factors. The *Moore–Penrose pseudoinverse* of $\mathbf{A}$ is

$$\mathbf{A}^{+} = \mathbf{V}\Sigma^{+}\mathbf{U}^{\top}, \qquad \Sigma^{+}_{ii} = \begin{cases} 1/\sigma_i & \sigma_i > 0, \\ 0 & \sigma_i = 0, \end{cases} \tag{33.56}$$

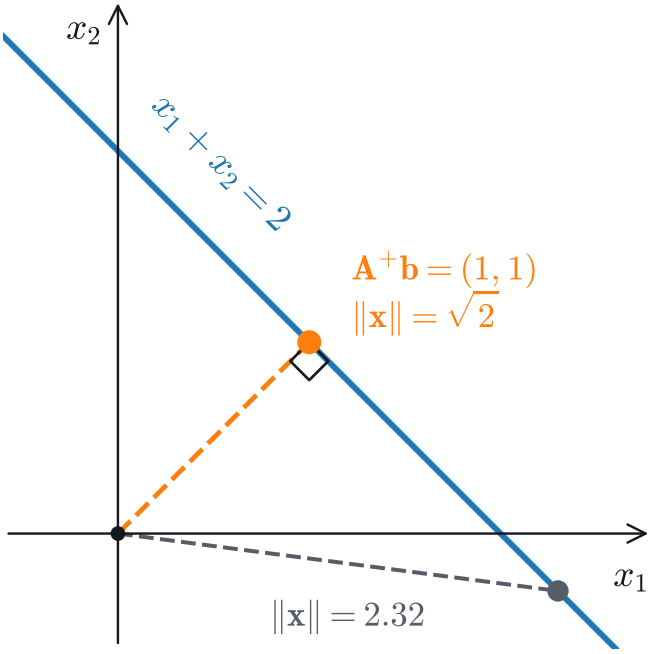


Figure 33.30: Least squares with one equation and two unknowns. Every point of the line solves $x_1 + x_2 = 2$; the pseudoinverse returns the one closest to the origin, the foot of the perpendicular.

where Σ^{+} has shape $n \times m$, the transpose shape of Σ. Geometrically, $\mathbf{A}^{+}$ reverses every stretch that can be reversed and sends the left null space to zero. For the shear, $\mathbf{T}^{+} = \mathbf{T}^{-1} = \left(\begin{smallmatrix} 1 & -1 \\ 0 & 1 \end{smallmatrix}\right)$; for $\mathbf{A} = (1 \;\; 1)$, $\mathbf{A}^{+} = \mathbf{v}_1 \sigma_1^{-1} u_1 = (\tfrac{1}{2}, \tfrac{1}{2})^{\top}$, so $\mathbf{A}^{+} \cdot 2 = (1, 1)$.

Proposition (minimum-norm least squares). *Among all minimizers of $\|\mathbf{A}\mathbf{x} - \mathbf{b}\|$, the unique one of smallest norm is $\hat{\mathbf{x}} = \mathbf{A}^{+}\mathbf{b}$.*

Proof. Change variables to the singular bases by setting $\mathbf{y} = \mathbf{V}^{\top}\mathbf{x}$ and $\mathbf{c} = \mathbf{U}^{\top}\mathbf{b}$. Because $\mathbf{U}$ is orthogonal, it preserves Euclidean norms, so

$$\|\mathbf{A}\mathbf{x} - \mathbf{b}\|^2 = \|\mathbf{U}^{\top}(\mathbf{A}\mathbf{x} - \mathbf{b})\|^2 = \|\Sigma\mathbf{y} - \mathbf{c}\|^2 = \sum_{i \le r} (\sigma_i y_i - c_i)^2 + \sum_{i > r} c_i^2.$$

Rotating into the singular bases decouples the system into independent scalar equations. The first sum is minimized term by term by setting $y_i = c_i / \sigma_i$ for $i \le r$. The second sum is a fixed constant, the component of $\mathbf{b}$ outside the column space, which no choice of $\mathbf{x}$ can reduce. Meanwhile, the coordinates y_i for $i > r$ do not enter the residual at all, and since $\|\mathbf{x}\| = \|\mathbf{y}\|$, setting them to zero gives the unique minimizer of smallest norm. That choice is $\mathbf{y} = \Sigma^{+}\mathbf{c}$, which translates back to $\hat{\mathbf{x}} = \mathbf{V}\Sigma^{+}\mathbf{U}^{\top}\mathbf{b} = \mathbf{A}^{+}\mathbf{b}$. $\blacksquare$

In the singular bases, minimizing the residual and minimizing the norm govern completely disjoint coordinates ($i \le r$ versus $i > r$), so both goals are achieved simultaneously. When $\mathbf{A}$ is square and invertible, every σ_i is positive and $\mathbf{A}^{+} = \mathbf{V}\Sigma^{-1}\mathbf{U}^{\top} = \mathbf{A}^{-1}$. Yet the same formula also exposes the main danger of least squares: dividing by a tiny σ_i multiplies any noise in $\mathbf{b}$ along $\mathbf{u}_i$ by $1/\sigma_i$.

Two standard remedies tame these unstable directions. *Ridge regression* minimizes $\|\mathbf{A}\mathbf{x} - \mathbf{b}\|^2 + \lambda\|\mathbf{x}\|^2$ for $\lambda > 0$, giving $\mathbf{x}_\lambda = (\mathbf{A}^{\top}\mathbf{A} + \lambda\mathbf{I})^{-1}\mathbf{A}^{\top}\mathbf{b}$. The *truncated pseudoinverse* $\mathbf{x}_k$ instead keeps only the first k singular terms of $\mathbf{A}^{+}\mathbf{b}$. Substituting the SVD and noting that $\mathbf{A}^{\top}\mathbf{A} + \lambda\mathbf{I} = \mathbf{V}(\Sigma^{\top}\Sigma + \lambda\mathbf{I})\mathbf{V}^{\top}$, both solutions take a transparent filter form:

$$\mathbf{x}_\lambda = \sum_{i \le r} \frac{\sigma_i^2}{\sigma_i^2 + \lambda} \, \frac{\mathbf{u}_i^{\top}\mathbf{b}}{\sigma_i} \, \mathbf{v}_i, \qquad \mathbf{x}_k = \sum_{i \le k} \frac{\mathbf{u}_i^{\top}\mathbf{b}}{\sigma_i} \, \mathbf{v}_i. \tag{33.57}$$

Both start from the unregularized pseudoinverse terms $\mathbf{u}_i^{\top}\mathbf{b}/\sigma_i$. Ridge regression scales component i by the smooth filter factor $\sigma_i^2/(\sigma_i^2 + \lambda)$, which stays near 1 when $\sigma_i^2 \gg \lambda$ and shrinks toward 0 when $\sigma_i^2 \ll \lambda$; truncation applies a sharp step function of 1 or 0. For $\lambda = 1$, singular values 3, 1, and 0.3 receive filter factors 0.9, 0.5, and 0.083. For a linear model trained with squared loss, weight decay (Section 2.7) is precisely ridge regression, and these filter factors show exactly which directions it damps.

The normal equations. The classical textbook route to least squares solves the normal equations $\mathbf{A}^{\top}\mathbf{A}\mathbf{x} = \mathbf{A}^{\top}\mathbf{b}$. For any $\mathbf{A}$, its solutions are the least-squares minimizers (since $\mathbf{A}^{\top}(\mathbf{A}\mathbf{x} - \mathbf{b}) = \mathbf{0}$ states that the residual is orthogonal to the column space), and the solution is unique whenever $\mathbf{A}$ has full column rank. Numerically, however, forming $\mathbf{A}^{\top}\mathbf{A}$ is a bad idea because it squares the condition number (Section 33.5.3.2). Modern libraries therefore solve least-squares problems via the SVD or a QR factorization instead.

Let us test the pseudoinverse on one overdetermined and one underdetermined system. The first fits a line $b \approx x_1 + x_2 t$ to the four observations $\mathbf{b} = (6, 5, 7, 10)$ at $t = 1, 2, 3, 4$; the second is our single equation $x_1 + x_2 = 2$.

```
A = np.array([[1., 1.], [1., 2.], [1., 3.], [1., 4.]])     # 4 equations, 2 unknowns
b = np.array([6., 5., 7., 10.])
x_pinv = np.linalg.pinv(A) @ b
x_lstsq = np.linalg.lstsq(A, b, rcond=None)[0]
print('overdetermined: pinv', x_pinv.round(4), ' lstsq', x_lstsq.round(4),
      ' residual', np.linalg.norm(A @ x_pinv - b).round(4))
A1 = np.array([[1., 1.]])                                   # 1 equation: x1 + x2 = 2
x1 = np.linalg.pinv(A1) @ np.array([2.])
print('underdetermined: pinv gives', x1.round(4), ' norm', np.linalg.norm(x1).round(4))
```

```
overdetermined: pinv [3.5 1.4]  lstsq [3.5 1.4]  residual 2.0494
underdetermined: pinv gives [1. 1.]  norm 1.4142
```

The pseudoinverse and `np.linalg.lstsq` agree on the fitted line $3.5 + 1.4t$, with residual $\sqrt{4.2} \approx 2.049$, and for the single equation the pseudoinverse returns the foot of the perpendicular, $(1, 1)$.

33.5.3.2 The Condition Number

In practice, the entries of a linear system come from noisy measurements or finite-precision rounding. Before trusting a computed solution, we need to know how much a tiny change in the data can alter the answer. A simple 2×2 system reveals how dramatic the amplification can be. Consider $\mathbf{K}\mathbf{x} = \mathbf{b}$ with $\mathbf{K} = \left(\begin{smallmatrix} 1 & 1 \\ 1 & 1.0001 \end{smallmatrix}\right)$ and $\mathbf{b} = (2, 2.0001)$. Its exact solution is $\mathbf{x} = (1, 1)$. Now nudge b_2 by just 10^{-4}, to 2.0002: the solution jumps all the way to $(0, 2)$. A perturbation of roughly three parts in 10^5 in the data has completely changed the answer. The singular values of $\mathbf{K}$, roughly 2.00005 and 5×10^{-5}, tell us why.

The *condition number* of a matrix $\mathbf{A}$ of rank r is the ratio of its largest to its smallest nonzero singular value,

$$\kappa(\mathbf{A}) = \frac{\sigma_1}{\sigma_r}. \tag{33.58}$$

Geometrically, $\kappa(\mathbf{A})$ measures how unevenly $\mathbf{A}$ stretches different directions: for $\mathbf{K}$ it is about 4×10^4, whereas for the shear it is only $\varphi^2 \approx 2.618$. When $\mathbf{A}$ is symmetric positive definite, its singular values coincide with its eigenvalues, so $\kappa = \lambda_{\max}/\lambda_{\min}$, recovering the condition number for quadratic bowls in Section 33.3.2.3. The next proposition shows that $\kappa(\mathbf{A})$ is the exact worst-case amplification factor for relative errors in a linear solve.

Proposition (error amplification). *Let $\mathbf{A}$ be square and invertible, $\mathbf{A}\mathbf{x} = \mathbf{b}$ with $\mathbf{b} \neq \mathbf{0}$, and let a perturbation $\delta\mathbf{b}$ shift the solution to $\mathbf{x} + \delta\mathbf{x}$. Then*

$$\frac{\|\delta\mathbf{x}\|}{\|\mathbf{x}\|} \le \kappa(\mathbf{A}) \, \frac{\|\delta\mathbf{b}\|}{\|\mathbf{b}\|}, \tag{33.59}$$

and for every $\mathbf{A}$ there exist $\mathbf{b}$ and $\delta\mathbf{b}$ that attain equality.

Proof. From $\mathbf{A}\,\delta\mathbf{x} = \delta\mathbf{b}$ we have $\delta\mathbf{x} = \mathbf{A}^{-1}\delta\mathbf{b}$. The largest singular value of $\mathbf{A}^{-1} = \mathbf{V}\Sigma^{-1}\mathbf{U}^{\top}$ is $1/\sigma_n$, so Equation 33.48 gives $\|\delta\mathbf{x}\| \le \sigma_n^{-1}\|\delta\mathbf{b}\|$. Likewise, $\mathbf{b} = \mathbf{A}\mathbf{x}$ gives $\|\mathbf{b}\| \le \sigma_1\|\mathbf{x}\|$. Combining the two inequalities yields

$$\frac{\|\delta\mathbf{x}\|}{\|\mathbf{x}\|} \le \frac{\sigma_1}{\sigma_n}\,\frac{\|\delta\mathbf{b}\|}{\|\mathbf{b}\|} = \kappa(\mathbf{A})\,\frac{\|\delta\mathbf{b}\|}{\|\mathbf{b}\|}.$$

Equality holds in both bounds simultaneously when $\mathbf{b} = \mathbf{u}_1$ (making $\mathbf{x} = \sigma_1^{-1}\mathbf{v}_1$ as short as possible for a unit right-hand side) and $\delta\mathbf{b} = \varepsilon\mathbf{u}_n$ (making $\delta\mathbf{x} = \varepsilon\sigma_n^{-1}\mathbf{v}_n$ as large as possible for a perturbation of size ε). ■

The relative error in the data is therefore amplified by at most κ. With $\kappa = 10^4$, a relative error of 10^{-8} in $\mathbf{b}$ can grow into 10^{-4} in $\mathbf{x}$, costing four digits of accuracy. In our 2×2 example, the relative change in $\mathbf{b}$ is $10^{-4}/\|\mathbf{b}\| \approx 3.5 \times 10^{-5}$, so the bound allows a relative change of 1.41 in $\mathbf{x}$, and the realized change is 1.0. The perturbation $(0, 10^{-4})$ places $1/\sqrt{2}$ of its length along the most amplified direction $\mathbf{u}_2 \approx (1, -1)/\sqrt{2}$, while $\mathbf{b}$ points almost directly along $\mathbf{u}_1$.

At the ideal extreme, an orthogonal matrix has all singular values equal to 1, so $\kappa = 1$: rotations and reflections never amplify error. That is why computing with the orthogonal factors $\mathbf{U}$ and $\mathbf{V}$ is numerically safe, and why an orthogonally initialized weight matrix preserves the norm of every input vector. At the opposite extreme, forming the Gram matrix squares the condition number,

$$\kappa(\mathbf{A}^\top\mathbf{A}) = \kappa(\mathbf{A})^2,$$

because Equation 33.49 is the SVD of $\mathbf{A}^\top\mathbf{A}$, with singular values σ_i^2 (for any matrix $\mathbf{A}$ of any rank). For the shear, κ jumps from $\varphi^2 \approx 2.62$ to $\varphi^4 \approx 6.85$. The cell below verifies our 2×2 example and compares $\kappa(\mathbf{A})$ with $\kappa(\mathbf{A}^\top\mathbf{A})$ for the line-fitting matrix above.

```
K = np.array([[1., 1.], [1., 1.0001]])
b = np.array([2., 2.0001])
x = np.linalg.solve(K, b)
x2 = np.linalg.solve(K, b + np.array([0., 1e-4]))
print('x =', x.round(4), '  after changing b_2 by 1e-4:', x2.round(4) + 0.)
kappa = np.linalg.cond(K)
print(f'kappa(K) = {kappa:.0f};  bound {kappa * 1e-4 / np.linalg.norm(b):.2f};  '
      f'realized {np.linalg.norm(x2 - x) / np.linalg.norm(x):.2f}')
A = np.array([[1., 1.], [1., 2.], [1., 3.], [1., 4.]])
print(f'kappa(A) = {np.linalg.cond(A):.3f};  kappa(A^T A) = {np.linalg.cond(A.T @ A):.2f};  '
      f'kappa(A)^2 = {np.linalg.cond(A)**2:.2f}')
```

```
x = [1. 1.]   after changing b_2 by 1e-4: [0. 2.]
kappa(K) = 40002;  bound 1.41;  realized 1.00
kappa(A) = 7.469;  kappa(A^T A) = 55.78;  kappa(A)^2 = 55.78
```

The output confirms our hand calculation: the solution jumps from $(1, 1)$ to $(0, 2)$, with $\kappa(\mathbf{K}) \approx 40{,}002$. Even for the modest 4×2 line-fitting matrix, $\kappa = 7.47$, so forming the normal equations would already inflate the condition number to $\kappa^2 = 55.8$.

Conditioning and the shape of a bowl. A large condition number also shows up directly in optimization as elongated level sets. For a full-column-rank least-squares objective $\frac{1}{2}\|\mathbf{A}\mathbf{x} - \mathbf{b}\|^2$, the Hessian is $\mathbf{H} = \mathbf{A}^\top\mathbf{A}$, whose level sets are ellipsoids with semi-axes proportional to $1/\sigma_i$ along the $\mathbf{v}_i$. The contours therefore have axis ratio $\kappa(\mathbf{A})$, while the Hessian itself has condition number $\kappa(\mathbf{A})^2$. On such a narrow valley, gradient descent zig-zags across the

steep walls and makes slow progress along the floor (Figure 33.31), at a convergence rate governed by $\kappa(\mathbf{H})$ that we derive in Section 35.1.

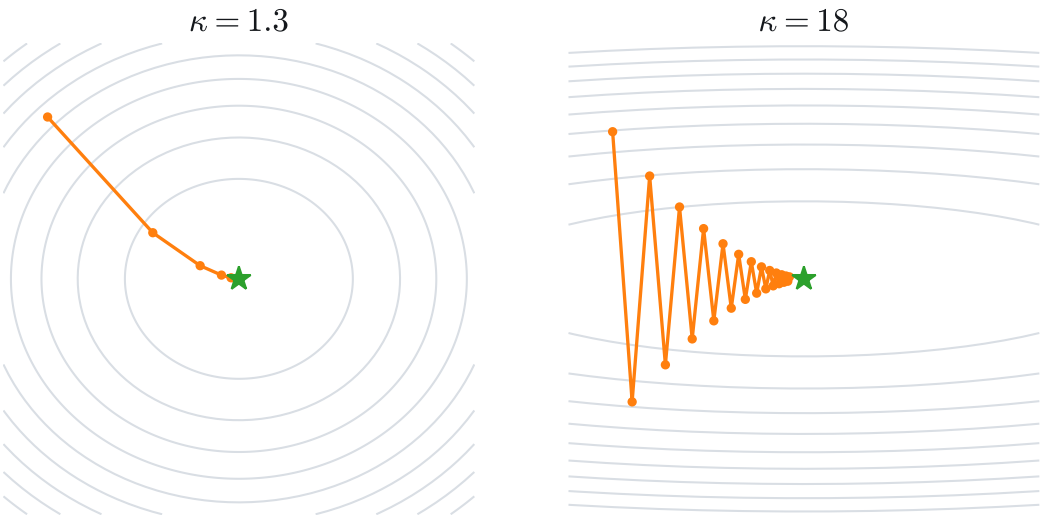


Figure 33.31: Gradient descent on a well-conditioned quadratic bowl ($\kappa(\mathbf{H}) = 1.3$, near-circular contours, an almost straight path) versus an ill-conditioned one ($\kappa(\mathbf{H}) = 18$, a zig-zag trajectory across the narrow valley). The panel titles give the condition number $\kappa(\mathbf{H})$ of the Hessian; the contours are elongated by its square root, a factor of about 4.2 in the second panel.

33.5.4 The SVD in Deep Learning

Three recurring questions in modern large-model training come down directly to the singular values of a weight matrix: how many parameters a fine-tuning adapter needs, how to construct an update that treats every direction of weight space equally, and how much a single layer can stretch its input.

33.5.4.1 Low-Rank Updates

Updating every parameter of a large pretrained model during fine-tuning is costly in both memory and compute. LoRA (Hu et al. 2021) avoids this by freezing each pretrained weight matrix $\mathbf{W} \in \mathbb{R}^{m\times n}$ and learning only a rank-k correction $\Delta\mathbf{W} = \mathbf{BA}$, where $\mathbf{B} \in \mathbb{R}^{m\times k}$ and $\mathbf{A} \in \mathbb{R}^{k\times n}$ (here $\mathbf{A}$ follows the paper's notation for the $k\times n$ down-projection, not the generic matrix of earlier subsections). The adapter uses $k(m+n)$ trainable parameters instead of mn. For a 4096×4096 layer with $k = 8$, that drops the parameter count from $16{,}777{,}216$ to $8 \cdot 8192 = 65{,}536$, or just 0.39%. As Figure 33.32 illustrates, the frozen weight and the low-rank bypass act in parallel on the same input and add their outputs.

What does Eckart–Young tell us here, and what does it *not* tell us? If full fine-tuning would produce an update with singular values $\sigma_1 \geq \sigma_2 \geq \cdots$, then Eckart–Young guarantees that no rank-k adapter can approximate that specific update closer than σ_{k+1} in the spectral norm or $(\sum_{i>k} \sigma_i^2)^{1/2}$ in the Frobenius norm. It does *not* dictate which adapter gradient descent actually learns. LoRA succeeds whenever *some* low-rank update achieves low loss—

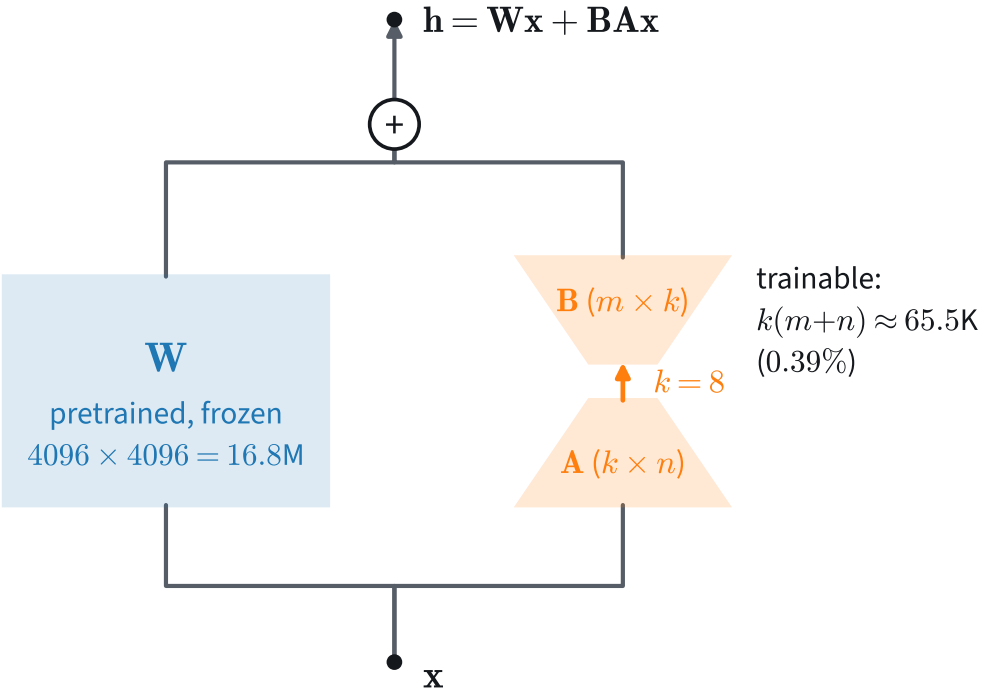


Figure 33.32: The LoRA architecture for one layer. The pretrained $\mathbf{W}$ (all $mn = 16.8\text{M}$ parameters for $m = n = 4096$) stays frozen; a parallel bypass first compresses the input to $k = 8$ dimensions with $\mathbf{A}$ and then expands back with $\mathbf{B}$, so only $k(m + n) \approx 65.5\text{K}$ parameters (about 0.39%) are trained. The two branches add: $\mathbf{h} = \mathbf{W}\mathbf{x} + \mathbf{B}\mathbf{A}\mathbf{x}$. After training, $\mathbf{B}\mathbf{A}$ can be merged into $\mathbf{W}$, so deployment costs nothing extra.

even if it is not the closest rank-k approximation to the full fine-tuning update. That such low-rank solutions exist for many downstream tasks is the empirical discovery behind LoRA. Exercise 2 bounds how much a rank-k adapter can perturb the singular values of $\mathbf{W}$.

33.5.4.2 Orthogonalized Updates

The Muon optimizer (Jordan et al. 2024) updates a weight matrix not along its raw momentum matrix $\mathbf{M}$, but along the *polar factor* of $\mathbf{M}$, the matrix that keeps the singular vectors of $\mathbf{M}$ while setting every nonzero singular value to 1. Section 9.6 motivates and develops the full optimizer; here we supply the two linear-algebraic tools it relies on: what the polar factor is, and how to compute it using only matrix multiplications.

The polar decomposition. For a square matrix $\mathbf{A}$, inserting $\mathbf{V}^\top\mathbf{V} = \mathbf{I}$ into the SVD splits it into an orthogonal factor times a symmetric stretch:

$$\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^\top = (\mathbf{U}\mathbf{V}^\top)\,(\mathbf{V}\Sigma\mathbf{V}^\top) = \mathbf{Q}\,\mathbf{P}. \tag{33.60}$$

Here $\mathbf{Q} = \mathbf{U}\mathbf{V}^\top$ is orthogonal (as a product of orthogonal matrices) and $\mathbf{P} = \mathbf{V}\Sigma\mathbf{V}^\top$ is symmetric positive semidefinite with eigenvalues σ_i. Every square matrix thus decomposes into a symmetric stretch followed by an orthogonal map, the matrix analogue of writing a complex number in polar form as $z = e^{i\theta}|z|$. Note that $\mathbf{Q}$ need not be a pure rotation: when $\mathbf{A}$ is invertible, $\det\mathbf{Q} = \operatorname{sign}\det\mathbf{A}$, so $\mathbf{Q}$ includes a reflection whenever $\det\mathbf{A} < 0$. For the shear $\mathbf{T}$, Equation 33.47 yields $\mathbf{Q} = \mathbf{R}_{31.7^\circ}\mathbf{R}_{58.3^\circ}^\top = \mathbf{R}_{-26.6^\circ} = \frac{1}{\sqrt{5}}\left(\begin{smallmatrix} 2 & 1 \\ -1 & 2 \end{smallmatrix}\right)$ (since $\tan 26.6^\circ = \frac{1}{2}$) and $\mathbf{P} = \frac{1}{\sqrt{5}}\left(\begin{smallmatrix} 2 & 1 \\ 1 & 3 \end{smallmatrix}\right)$. Checking the product confirms $\mathbf{Q}\mathbf{P} = \frac{1}{5}\left(\begin{smallmatrix} 5 & 5 \\ 0 & 5 \end{smallmatrix}\right) = \mathbf{T}$ and $\mathbf{P}^2 = \left(\begin{smallmatrix} 1 & 1 \\ 1 & 2 \end{smallmatrix}\right) = \mathbf{T}^\top\mathbf{T}$.

For a rectangular matrix $\mathbf{M}$ with thin SVD $\mathbf{M} = \mathbf{U}\Sigma\mathbf{V}^\top$, the polar factor is still $\mathbf{U}\mathbf{V}^\top$: it shares the singular vectors of $\mathbf{M}$ and has all singular values equal to 1, so its columns (when $m \geq n$) or its rows (when $m \leq n$) are orthonormal. When $\mathbf{M}$ is rank-deficient ($r < \min(m, n)$), the polar factor is not unique, and the natural canonical choice is $\mathbf{U}_r\mathbf{V}_r^\top$ from the compact SVD. Exercise 6 shows that for a square matrix, $\mathbf{U}\mathbf{V}^\top$ is also the closest orthogonal matrix to $\mathbf{A}$ in the Frobenius norm.

Odd polynomials of a matrix. Computing a full SVD of every weight matrix at every training step would be prohibitively slow. Fortunately, we can transform the singular values without touching the singular vectors using nothing more than matrix multiplications.

Proposition (odd polynomials act on singular values). *Let $\mathbf{X} = \mathbf{U}\Sigma\mathbf{V}^\top$ be a thin SVD and let $p(\sigma) = \sum_j c_j\sigma^{2j+1}$ be an odd polynomial. Then*

$$p(\mathbf{X}) := \sum_j c_j\,\mathbf{X}(\mathbf{X}^\top\mathbf{X})^j = \mathbf{U}\,p(\Sigma)\,\mathbf{V}^\top. \tag{33.61}$$

Proof. Since $\mathbf{X}^\top\mathbf{X} = \mathbf{V}\Sigma^2\mathbf{V}^\top$, taking powers gives $(\mathbf{X}^\top\mathbf{X})^j = \mathbf{V}\Sigma^{2j}\mathbf{V}^\top$, and left-multiplying by $\mathbf{X} = \mathbf{U}\Sigma\mathbf{V}^\top$ gives $\mathbf{X}(\mathbf{X}^\top\mathbf{X})^j = \mathbf{U}\Sigma^{2j+1}\mathbf{V}^\top$. Summing over j completes the proof. ■

Any odd matrix polynomial therefore leaves $\mathbf{U}$ and $\mathbf{V}$ untouched while applying the scalar function p independently to each singular value. To reach the polar factor $\mathbf{U}\mathbf{V}^\top$, we simply need an odd polynomial p whose repeated application drives every positive singular value to 1.

Newton–Schulz. The classic *Newton–Schulz iteration* $\mathbf{X} \leftarrow \frac{1}{2}(3\mathbf{X} - \mathbf{X}\mathbf{X}^\top\mathbf{X})$ applies the cubic polynomial $p(\sigma) = \frac{1}{2}(3\sigma - \sigma^3)$ to every singular value. Its fixed points are 0 and ± 1. On $(0, 1]$, p pushes every value upward without ever overshooting 1, since $p(\sigma) - \sigma = \frac{1}{2}\sigma(1 - \sigma^2) \geq 0$ and p reaches its maximum $p(1) = 1$. In fact, p maps the entire interval $(0, \sqrt{3})$ into $(0, 1]$, so every starting value in this *immediate basin* of 1 converges to 1, whereas larger starting values can behave erratically (Exercise 5). Scaling the initial matrix as $\mathbf{X}_0 = \mathbf{M}/\|\mathbf{M}\|_F$ guarantees that every singular value starts inside $(0, 1]$, because $\|\mathbf{M}\|_F \geq \sigma_1(\mathbf{M})$. Any zero singular value stays zero (since $p(0) = 0$), so a rank-deficient $\mathbf{M}$ converges cleanly to $\mathbf{U}_r\mathbf{V}_r^\top$.

How many iterations does this take? Near 0, $p(\sigma) \approx \frac{3}{2}\sigma$, so a small singular value s grows by a factor of $3/2$ per step and needs roughly $\log_{3/2}(1/s)$ steps to reach order one. Near 1, expanding gives $p(1 - e) = 1 - \frac{3}{2}e^2 + \frac{1}{2}e^3$: the error roughly squares at every step, shrinking from order one to 10^{-6} in about four additional steps. Gluing the two phases together predicts roughly $\log_{3/2}(1/s_{\min}) + 4$ steps, governed by the smallest normalized singular value $s_{\min} = \sigma_{\min}(\mathbf{M})/\|\mathbf{M}\|_F$. Let us test this rough estimate on a random 6×4 matrix standing in for the momentum.

```
M = np.random.standard_normal((6, 4))                # a stand-in momentum matrix
X = M / np.linalg.norm(M)                           # Frobenius normalization: sigma in (0, 1]
s_min = np.linalg.svd(X, compute_uv=False)[-1]
for k in range(1, 41):
    X = 1.5 * X - 0.5 * X @ X.T @ X                  # p(sigma) = (3 sigma - sigma^3) / 2
    if np.abs(np.linalg.svd(X, compute_uv=False) - 1).max() < 1e-6:
        break
U, s, Vt = np.linalg.svd(M, full_matrices=False)
print(f'smallest scaled sigma {s_min:.3f};  predicted steps '
      f'{np.log(1 / s_min) / np.log(1.5) + 4:.1f}')
print(f'steps until max|sigma_i - 1| < 1e-6: {k};  distance to U V^T: '
      f'{np.linalg.norm(X - U @ Vt):.1e}')
```

```
smallest scaled sigma 0.146;  predicted steps 8.7
steps until max|sigma_i - 1| < 1e-6: 9;  distance to U V^T: 7.3e-11
```

The actual step count matches the prediction to within a single iteration, and the iterate lands within 10^{-5} of the exact polar factor from a full SVD. Because $s_{\min}$ varies across random draws, so does the step count; a nearly rank-deficient $\mathbf{M}$ takes longer. In practice, Muon replaces the cubic with a tuned quintic polynomial that trades exact convergence to 1 for speed (Section 9.6.2).

33.5.4.3 Measuring and Controlling the Spectrum

Our third question asks how much a linear layer can stretch its input, and the largest singular value answers it directly. More broadly, the full singular spectrum tells us whether a matrix is truly close to low rank. Modern training recipes therefore bound, inspect, and approximate singular values without ever paying for a full SVD.

Spectral normalization. By Equation 33.48, a linear layer $\mathbf{x} \mapsto \mathbf{W}\mathbf{x}$ satisfies $\|\mathbf{W}\mathbf{x} - \mathbf{W}\mathbf{x}'\| \leq \sigma_1\|\mathbf{x} - \mathbf{x}'\|$, with equality whenever $\mathbf{x} - \mathbf{x}'$ aligns with $\mathbf{v}_1$. The smallest constant bounding how much a map can amplify input differences is its *Lipschitz constant*, so here the Lipschitz constant is simply $\sigma_1 = \|\mathbf{W}\|_2$. Dividing $\mathbf{W}$ by σ_1 caps the layer's stretch at 1, a technique that

stabilizes the training of GAN discriminators (Miyato et al. 2018). Rather than computing a full SVD, σ_1 is estimated via power iteration on $\mathbf{W}^\top\mathbf{W}$ (Section 33.4.1.2). In practice, one alternates $\mathbf{u} \leftarrow \mathbf{W}\mathbf{v}$ and $\mathbf{v} \leftarrow \mathbf{W}^\top\mathbf{u}$, normalizing at each half-step; together they perform one multiplication by $\mathbf{W}^\top\mathbf{W}$, with $\|\mathbf{W}\mathbf{v}\|$ as the estimate of σ_1. Each round pulls $\mathbf{v}$ closer to $\mathbf{v}_1$, shrinking the error by roughly $(\sigma_2/\sigma_1)^2$. Miyato et al. (2018) found empirically that a single round per training step suffices when warm-started from the previous step's vector. (For a convolutional layer, reshaping the kernel into a 2D matrix gives an approximation to the convolution's spectral norm, though not its exact value.)

Measuring the spectrum. Our earlier 0.39% LoRA figure assumed that rank 8 suffices. Whether a matrix is truly close to low rank depends entirely on how fast its singular values decay, so one should measure before truncating. The cell below compares two 400×500 matrices: a random Gaussian matrix with no structure at all, and our cat photograph with its mean pixel value subtracted.

```
G = np.random.standard_normal(img.shape)                        # no structure
for name, W in [('Gaussian', G), ('photograph', img - img.mean())]:
    s = np.linalg.svd(W, compute_uv=False)
    r95 = int(np.searchsorted(np.cumsum(s**2) / (s**2).sum(), 0.95)) + 1
    print(f'{name:10s}: sigma from {s[0]:.3g} to {s[-1]:.2g};  95% energy at rank {r95:3d};  '
          f'adapter {r95 * (m + n) / (m * n):.1%} of W')
```

```
Gaussian  : sigma from 42.2 to 2.5;  95% energy at rank 269;  adapter 121.0% of W
photograph: sigma from 2.65e+04 to 2.4;  95% energy at rank   8;  adapter 3.6% of W
```

The contrast is stark. The singular values of the Gaussian matrix spread across the Marchenko–Pastur interval $[\sqrt{n} - \sqrt{m}, \sqrt{n} + \sqrt{m}] \approx [2.4, 42.4]$ from Section 33.5.2.1, with no drop-off: capturing 95% of its energy takes rank roughly 270 out of 400, so a low-rank adapter would actually use *more* parameters than the original matrix. The centered photograph, by contrast, reaches 95% of its energy at rank 8, corresponding to an adapter with under 4% of the parameters. Neither matrix is a trained neural-network weight, so this pair only illustrates the two extremes; for LoRA, the matrix that matters is the task-specific update, whose spectrum must be measured rather than assumed.

Randomized SVD. When a matrix is too large to factor directly, *randomized SVD* (Halko et al. 2011) computes an accurate rank-k approximation using only a handful of matrix–vector products with $\mathbf{A}$. Its core ingredient is a *range finder*: multiply $\mathbf{A}$ by a random Gaussian matrix Ω with slightly more than k columns, orthonormalize the columns of $\mathbf{A}\Omega$ into $\mathbf{Q}$, and compute the SVD of the much smaller matrix $\mathbf{Q}^\top\mathbf{A}$. Whenever the singular values decay, the top singular directions dominate every random linear combination of the columns of $\mathbf{A}$, so $\mathbf{Q}$ captures them with high probability. Exercise 7 implements this algorithm along with a power-iteration refinement for slowly decaying spectra, and Section 33.6 places it in the broader context of randomized sketching.

Finally, keep three limits of the SVD in mind. First, a full SVD costs $O(mn\min(m, n))$ operations (roughly $4096^3 \approx 7 \times 10^{10}$ for a single 4096×4096 weight matrix), which is why practical algorithms like Newton–Schulz and randomized range finders rely solely on matrix products. Second, low rank is a property of the data, never a free guarantee, as the Gaussian matrix just demonstrated. Third, the SVD captures only *linear* structure: points lying on a circle in the plane form a rank-2 data matrix even though the circle is inherently one-dimensional. Understanding curved functions and nonlinear structure requires derivatives, which is where we turn in the next chapter.

33.5.5 Summary

Every real matrix $\mathbf{A}$, regardless of its shape, admits an orthonormal input basis $\{\mathbf{v}_i\}$ and an orthonormal output basis $\{\mathbf{u}_i\}$ between which it acts as a simple list of non-negative stretches, $\mathbf{A}\mathbf{v}_i = \sigma_i \mathbf{u}_i$ with $\sigma_1 \geq \sigma_2 \geq \cdots \geq 0$. The factorization follows from applying the spectral theorem to the Gram matrix $\mathbf{A}^\top \mathbf{A}$, with $\sigma_i = \sqrt{\lambda_i(\mathbf{A}^\top \mathbf{A})}$.

From that single ordered list of singular values, five core quantities fall out at once: the rank (the number of positive σ_i), the spectral and Frobenius norms ($\|\mathbf{A}\|_2 = \sigma_1$ and $\|\mathbf{A}\|_F^2 = \sum_i \sigma_i^2$), the optimal rank-$k$ approximation given by the truncated SVD $\mathbf{A}_k$ (Eckart–Young, which becomes principal component analysis on centered data), the Moore–Penrose pseudoinverse $\mathbf{A}^+$ that returns the minimum-norm least-squares solution, and the condition number $\kappa = \sigma_1/\sigma_r$ that bounds relative error amplification. In deep learning, the same singular spectrum sizes low-rank fine-tuning adapters, defines the polar factor used by Muon, and gives the Lipschitz constant controlled by spectral normalization.

33.5.6 Exercises

1. **Norms and the maximum stretch.** The text obtained Equation 33.48 from the spectral theorem. The maximum stretch can also build the SVD on its own.
    1. Let $\sigma_1 = \max_{\|\mathbf{x}\|=1} \|\mathbf{A}\mathbf{x}\|$, assume $\sigma_1 > 0$, and build an SVD without the spectral theorem. (i) Show that the maximum is attained at some unit $\mathbf{v}_1$, and set $\mathbf{u}_1 = \mathbf{A}\mathbf{v}_1/\sigma_1$. (ii) For a unit $\mathbf{w} \perp \mathbf{v}_1$, the function $t \mapsto \|\mathbf{A}(\mathbf{v}_1 + t\mathbf{w})\|^2/(1+t^2)$ has a maximum at $t = 0$, so its derivative there vanishes. Deduce $\mathbf{u}_1^\top \mathbf{A}\mathbf{w} = 0$: $\mathbf{A}$ maps $\mathbf{v}_1^\perp$ into $\mathbf{u}_1^\perp$. (iii) Recurse on the restriction of $\mathbf{A}$ to $\mathbf{v}_1^\perp$. Where did compactness of the unit sphere enter?
    2. Show that $\|\mathbf{A}\|_2 \leq \|\mathbf{A}\|_F \leq \sqrt{r}\,\|\mathbf{A}\|_2$ with $r = \operatorname{rank}\mathbf{A}$. Which matrices attain equality on each side?
    3. The ratio $\|\mathbf{A}\|_F^2/\|\mathbf{A}\|_2^2$ is called the *stable rank* of $\mathbf{A}$; by (b) it lies between 1 and $\operatorname{rank}\mathbf{A}$. Compute it in closed form for the decaying spectrum $\sigma_i = e^{-i/12}$, $i = 0, \ldots, 255$ (a geometric series), and show that rank 18 is needed to hold 95% of the energy of this spectrum. Then estimate the stable rank of the 400×500 Gaussian matrix of Section 33.5.4.3 from $\|\mathbf{G}\|_F^2 \approx mn$ and $\sigma_1 \approx \sqrt{m} + \sqrt{n}$, and compare it with the rank of about 270 that holds 95% of its energy. Why can the stable rank be far below the rank?
2. **Weyl perturbation bound.** An update Δ, for instance a rank-k LoRA update $\mathbf{BA}$ (the paper's letters), changes the singular values of a weight matrix $\mathbf{W}$. The Weyl inequality Equation 33.54, proved in Section 33.5.2.1, limits how far they move.
    1. Set $j = 1$ in Equation 33.54 and apply the inequality in both directions to derive $|\sigma_i(\mathbf{W}+\Delta) - \sigma_i(\mathbf{W})| \leq \sigma_1(\Delta)$ for every i.
    2. Take $\mathbf{W} = \operatorname{diag}(1, 2)$ and $\Delta = 0.1\,\mathbf{1}\mathbf{1}^\top$. Compute the singular values of $\mathbf{W} + \Delta$ and compare each change with $\sigma_1(\Delta)$.
    3. Now let $\operatorname{rank}\Delta \leq k$. The Frobenius proof in Section 33.5.2.1 applies Equation 33.54 with $j = k + 1$ to a matrix of rank at most k. Apply the same step to Δ to obtain the interlacing inequalities $\sigma_{i+k}(\mathbf{W}) \leq \sigma_i(\mathbf{W}+\Delta)$ and $\sigma_{i+k}(\mathbf{W}+\Delta) \leq \sigma_i(\mathbf{W})$. Use them to show that the ranks of $\mathbf{W}+\Delta$ and $\mathbf{W}$ differ by at most k, and that at most k singular values of $\mathbf{W}+\Delta$ exceed $\sigma_1(\mathbf{W})$, however large $\sigma_1(\Delta)$ is. Why does the bound of
    (a) not give these conclusions?

3. code **Centering in PCA, and Fashion-MNIST.** Let the rows of $\mathbf{X} \in \mathbb{R}^{n\times d}$ be data points with mean row $\bar{\mathbf{x}}$ and covariance $\mathbf{C}$ as in the principal component analysis of Section 33.5.2.2.
   1. Show that $\frac{1}{n}\mathbf{X}^\top\mathbf{X} = \mathbf{C} + \bar{\mathbf{x}}\bar{\mathbf{x}}^\top$.
   2. Generate a cloud like the one of Section 33.5.2.2 (300 points with standard deviations 3 and 0.8 along axes rotated by $\pi/5$) and shift every point by $c\,(1,-1)/\sqrt{2}$ for $c \in \{0, 1, 3, 10, 30\}$. For each c, compute the top right singular vector of the *uncentered* data, its angle to the principal axis $(\cos\frac{\pi}{5}, \sin\frac{\pi}{5})$, and the ratio $\sigma_1^2/\sum_i \sigma_i^2$.
   3. Use (a) to predict, from the eigenvalues of $\mathbf{C}$, the offset at which the top singular vector turns from the principal axis toward the mean.
   4. Use (a) again to approximate the ratio $\sigma_1^2/\sum_i \sigma_i^2$ for large c, and find its limit as $c \to \infty$. At $c = 10$ the ratio is close to the centered explained-variance ratio. Does this agreement reflect the data or the chosen offset?
   5. Load the Fashion-MNIST test images of t-shirts and trousers through `d2l.FashionMNIST`, as in Section 33.1, flatten them to vectors of length 784, center them, and compute the top two right singular vectors. Plot the two classes in the plane of the first two principal components. What fraction of the variance do these two directions hold? Compute the angle between $\mathbf{v}_1$ and the difference of the class means $\bar{\mathbf{x}}_{\text{trouser}} - \bar{\mathbf{x}}_{\text{t-shirt}}$, and explain why the two directions are close but not equal. Finally, reconstruct one t-shirt from its first 5, 20 and 50 components and report the relative error of each.

4. code **Ridgeless limit and gradient descent.** An overparameterized linear model has more parameters than equations, so least squares has infinitely many minimizers. Part (a) holds for every shape and rank;

   (b) and (c) use $\mathbf{A} \in \mathbb{R}^{m\times n}$ with $m < n$.

   1. Use the filter form Equation 33.57 to show that $(\mathbf{A}^\top\mathbf{A} + \lambda\mathbf{I})^{-1}\mathbf{A}^\top \to \mathbf{A}^+$ as $\lambda \to 0^+$, for a matrix $\mathbf{A}$ of any shape and rank. Why does $(\mathbf{A}^\top\mathbf{A})^{-1}\mathbf{A}^\top$ not exist when $m < n$?
   2. Gradient descent on $\frac{1}{2}\|\mathbf{A}\mathbf{x} - \mathbf{b}\|^2$ with step size $0 < \eta < 2/\sigma_1^2$ updates $\mathbf{x}_{t+1} = \mathbf{x}_t - \eta\,\mathbf{A}^\top(\mathbf{A}\mathbf{x}_t - \mathbf{b})$. Show that $\mathbf{x}_t - \mathbf{x}_0$ stays in the row space of $\mathbf{A}$ and that $\mathbf{x}_t$ converges to $\mathbf{A}^+\mathbf{b} + (\mathbf{I} - \mathbf{A}^+\mathbf{A})\mathbf{x}_0$. Which solution does the initialization $\mathbf{x}_0 = \mathbf{0}$ select?
   3. Check (a) and (b) for a 20×50 Gaussian $\mathbf{A}$ and a Gaussian $\mathbf{b}$: run gradient descent from $\mathbf{x}_0 = \mathbf{0}$ and from a random $\mathbf{x}_0$, and compare both limits and their norms with $\mathbf{A}^+\mathbf{b}$.

5. code **Newton–Schulz basin.** The Newton–Schulz iteration $\mathbf{X} \leftarrow \frac{1}{2}(3\mathbf{X} - \mathbf{X}\mathbf{X}^\top\mathbf{X})$ of Section 33.5.4.2 starts from $\mathbf{X} = \mathbf{M}/\|\mathbf{M}\|_F$.
   1. Let $\mathbf{X} = \mathbf{U}\Sigma\mathbf{V}^\top$ be a thin SVD. Show directly, without Equation 33.61, that one step produces $\mathbf{U}\,p(\Sigma)\,\mathbf{V}^\top$ with $p(\sigma) = \frac{1}{2}(3\sigma - \sigma^3)$ applied to every diagonal entry.
   2. Show that $(0, \sqrt{3})$ is the immediate basin of 1: p maps it into $(0, 1]$, and the iterates $p(p(\cdots p(\sigma_0)))$ converge to 1 for every $\sigma_0 \in (0, \sqrt{3})$. Show that some starting values beyond $\sqrt{3}$ also reach 1 (try 2.15) and others do not (try 2). Expand $p(1 - e)$ in powers of e. How fast does the distance to 1 shrink once it is small?
   3. For a 6×4 Gaussian $\mathbf{M}$, count the steps until $\|\mathbf{X} - \mathbf{U}\mathbf{V}^\top\|_F < 10^{-6}$ when the iteration starts from $\mathbf{M}/c$ with $c = \|\mathbf{M}\|_F$ and with $c = \sigma_1(\mathbf{M})$. Compare both

counts with $\log_{3/2}(1/s_{\min}) + 4$, where $s_{\min}$ is the smallest singular value after scaling. Why does Muon divide by $\|\mathbf{M}\|_F$ although $\sigma_1(\mathbf{M})$ saves a step?

4. Start the iteration from $\mathbf{M}/c$ with $c = \sigma_1(\mathbf{M})/2$ and with $c = \sigma_1(\mathbf{M})/2.5$. What does it converge to in each case? Explain both outcomes with the polynomial p of (b).

6. code **Polar factor and Procrustes.** ● The polar decomposition Equation 33.60 writes a square matrix as $\mathbf{A} = \mathbf{QP}$ with $\mathbf{Q}$ orthogonal and $\mathbf{P}$ symmetric positive semidefinite.

   1. Show that $\mathbf{P} = (\mathbf{A}^\top\mathbf{A})^{1/2}$ is determined by $\mathbf{A}$, and that $\mathbf{Q}$ is determined by $\mathbf{A}$ when $\mathbf{A}$ is invertible. Give two different polar decompositions of $\operatorname{diag}(1, 0)$.
   2. Show that $\mathbf{Q} = \mathbf{UV}^\top$ is an orthogonal matrix closest to $\mathbf{A}$ in the Frobenius norm. Reduce the problem to maximizing $\operatorname{tr}(\mathbf{R}^\top\mathbf{A})$ over orthogonal $\mathbf{R}$.
   3. Two embedding spaces that differ by a rotation can be aligned from paired points $\mathbf{X}, \mathbf{Y} \in \mathbb{R}^{d\times N}$ by minimizing $\|\mathbf{RX} - \mathbf{Y}\|_F$ over orthogonal $\mathbf{R}$. Use (b) to show that the minimizer is the polar factor of $\mathbf{YX}^\top$. Test it with $d = 50$, $N = 1000$, a random orthogonal $\mathbf{R}$ (the $\mathbf{Q}$ factor of a QR decomposition of a Gaussian matrix), Gaussian $\mathbf{X}$, and $\mathbf{Y} = \mathbf{RX} + \mathbf{E}$, where $\mathbf{E}$ has i.i.d. $\mathcal{N}(0, s^2)$ entries for $s \in \{0, 0.1, 1, 3\}$. Compare $\|\hat{\mathbf{R}} - \mathbf{R}\|_F/\sqrt{d}$ for this estimate and for the unconstrained least-squares estimate $\mathbf{YX}^+$.

7. code **Randomized range finder.** ● The randomized SVD of Section 33.5.4.3 multiplies $\mathbf{A}$ by a Gaussian $\Omega \in \mathbb{R}^{n\times(k+p)}$, orthonormalizes $\mathbf{Y} = \mathbf{A}\Omega$ into $\mathbf{Q}$, and truncates the SVD of $\mathbf{Q}^\top\mathbf{A}$ to rank k.

   1. Implement it. Build $\mathbf{A} \in \mathbb{R}^{1000\times500}$ as $\mathbf{U}\operatorname{diag}(\sigma_1, \dots, \sigma_{500})\mathbf{V}^\top$ with random orthonormal $\mathbf{U}$ and $\mathbf{V}$, once with the fast decay $\sigma_i = e^{-i/5}$ and once with the slow decay $\sigma_i = 1/i$. For $k = 20$ and $p \in \{0, 5, 10\}$, divide the spectral-norm error of the rank-k result by σ_{k+1}, the minimum of Equation 33.53, and average over 10 draws of Ω. Why does oversampling alone suffice for one spectrum but not for the other?
   2. Replace $\mathbf{Y}$ by $(\mathbf{AA}^\top)^q\mathbf{A}\Omega$ for $q \in \{1, 2\}$. How do the ratios change for each spectrum? Explain the effect of q with the singular values of $(\mathbf{AA}^\top)^q\mathbf{A}$. How many products with $\mathbf{A}$ or $\mathbf{A}^\top$ does each variant cost?

8. code **Optimal hard threshold.** ●● The threshold $\frac{4}{\sqrt{3}}\sqrt{n}\,\sigma_{\text{noise}}$ of Gavish and Donoho (2014) follows from the behavior of a single signal component. Let $\mathbf{A} = \theta\,\mathbf{uv}^\top + \mathbf{N}$ be $n \times n$, with unit vectors $\mathbf{u}, \mathbf{v}$, $\theta > 0$, and $\mathbf{N}$ with i.i.d. Gaussian entries $\mathcal{N}(0, \sigma^2_{\text{noise}})$. Write $x = \theta/(\sqrt{n}\,\sigma_{\text{noise}})$ and let $(\sigma_1, \hat{\mathbf{u}}, \hat{\mathbf{v}})$ be the top singular triple of $\mathbf{A}$, with the sign of the pair chosen so that $\mathbf{u}^\top\hat{\mathbf{u}} \geq 0$. As $n \to \infty$, Benaych-Georges and Nadakuditi (2012) show that σ_1 tends to the bulk edge $2\sqrt{n}\,\sigma_{\text{noise}}$ and the singular vectors become orthogonal to $\mathbf{u}$ and $\mathbf{v}$ when $x \leq 1$, and that for $x > 1$

$$\frac{\sigma_1}{\sqrt{n}\,\sigma_{\text{noise}}} \to x + \frac{1}{x}, \qquad (\mathbf{u}^\top\hat{\mathbf{u}})^2 \to 1 - \frac{1}{x^2}, \qquad (\mathbf{v}^\top\hat{\mathbf{v}})^2 \to 1 - \frac{1}{x^2}.$$

   The transition at $x = 1$ is the singular-value counterpart of the eigenvalue transition at $\beta = \sqrt{c}$ in exercise 5 of Section 33.3.

   1. Two estimates of the signal $\theta\,\mathbf{uv}^\top$ are the rank-one truncation $\sigma_1\hat{\mathbf{u}}\hat{\mathbf{v}}^\top$ and the zero matrix. Express the squared Frobenius error of each in terms of θ, σ_1, $\mathbf{u}^\top\hat{\mathbf{u}}$ and $\mathbf{v}^\top\hat{\mathbf{v}}$.

2. Use the limits to show that the truncation has the smaller asymptotic error exactly when $x > \sqrt{3}$, and that the singular value at $x = \sqrt{3}$ is $\frac{4}{\sqrt{3}}\sqrt{n}\,\sigma_{\text{noise}}$. Which signal strengths produce a singular value above the noise edge that should nevertheless be discarded?
3. Check (a) and (b) for $n = 500$ and $\sigma_{\text{noise}} = 1/\sqrt{n}$, with $x \in \{0.5, 1.2, 1.5, \sqrt{3}, 2, 3\}$ and 10 draws each: compare σ_1, the squared cosines, and both errors with their limits.

33.6 Further Reading

This section connects the linear algebra developed in this chapter to numerical analysis, random matrix theory, and deep-learning applications, followed by a curated list of textbooks and courses covering the chapter's core material. If your main interest is deep learning, start with random projections, iterative solvers, random matrices and trainable depth, and sketching, then jump to the final two entries on low-rank attention and spectral-norm optimizers. Nothing elsewhere in the book depends on this section.

33.6.1 Geometry and Linear Maps

33.6.1.1 Other Inner Products: Kernel Methods

The nearest-mean rule of Section 33.1.4.2 classifies a test image by comparing it with two class averages through a single dot product, which forces the decision boundary to be a flat hyperplane. That simplicity breaks down whenever a class is curved or split into clusters, as happens when distinguishing t-shirts from shirts. Kernel methods keep the exact same geometric rule and change only the inner product: we replace $\mathbf{x} \cdot \mathbf{x}_i$ by a similarity function $k(\mathbf{x}, \mathbf{x}_i)$ that corresponds to a genuine inner product in some other, possibly infinite-dimensional, feature space. The class average becomes a mean in that hidden space, every training example votes with weight $k(\mathbf{x}, \mathbf{x}_i)$, and the resulting boundary can bend freely in the original input space. Asking which training points actually need nonzero weights leads directly to the support vector machine. Schölkopf and Smola (2002) open their book with this very classifier and build the full theory of kernel methods from it.

33.6.1.2 Random Projections

In Section 33.1, we saw that random directions in high-dimensional space $\mathbb{R}^d$ are almost orthogonal to one another. Surprisingly, that geometric quirk lets us shrink the dimension of a dataset without looking at the data at all. The Johnson–Lindenstrauss lemma (Johnson and Lindenstrauss 1984) guarantees that for any collection of n points, multiplying by a suitably scaled random Gaussian matrix into k dimensions of order $\varepsilon^{-2} \log n$ preserves all $\binom{n}{2}$ pairwise distances within a factor of $1 \pm \varepsilon$, with high probability. Unlike principal component analysis (Section 33.5.2.2), a random projection is completely data-independent, and its target dimension k does not even depend on the ambient dimension d. The trick therefore pays off most when d is vastly larger than $\varepsilon^{-2} \log n$. Dasgupta and Gupta (2003) give a remarkably short proof with explicit constants, while Vershynin (2018) situates the lemma inside the broader theory of concentration for random vectors.

33.6.1.3 Solving Linear Systems at Scale

In Section 33.2.2.4, we solved linear systems via LU factorization and noted that the inverse of a large sparse matrix is typically dense. Why does Gaussian elimination with partial pivoting work so reliably in floating-point arithmetic, even though pathological worst-case matrices exist? Trefethen and Bau (1997) give an illuminating explanation of this gap between worst-case and practical stability, and Higham (2002) provides the definitive rounding-error analysis.

At large scale, sparse solvers never form $\mathbf{A}^{-1}$, and often avoid forming dense LU factors altogether. Sparse direct methods carefully reorder the unknowns so that elimination introduces as few new nonzeros as possible. Iterative methods go a step further and access the matrix solely through matrix–vector products $\mathbf{A}\mathbf{x}$, much like power iteration (Section 33.4.1.2). When $\mathbf{A}$ is symmetric positive definite, the conjugate gradient method of Hestenes and Stiefel (1952) shrinks the error after k steps by at least a factor of $2\big((\sqrt{\kappa}-1)/(\sqrt{\kappa}+1)\big)^{k}$ in the norm induced by $\mathbf{A}$, where κ is the condition number from Section 33.5.3.2. Golub and Van Loan (1996) treat conjugate gradient and its relatives in full detail.

33.6.1.4 Contraction Order and Tensor Decompositions

The `einsum` notation of Section 33.2.1.4 specifies the mathematical result of contracting several tensors, but it leaves open the order in which pairwise contractions are carried out. That choice can make a dramatic difference: for $n \times n$ matrices, computing $(\mathbf{A}\mathbf{B})\mathbf{x}$ takes roughly n^3 multiplications, whereas $\mathbf{A}(\mathbf{B}\mathbf{x})$ takes only $2n^2$. When many factors are involved, choosing the cheapest order becomes a search over parenthesizations, which `np.einsum(..., optimize=True)` and the `opt_einsum` package (Smith and Gray 2018) automate.

The idea of low-rank approximation also extends beyond matrices to higher-order tensors, though with a sobering catch. A three-way tensor can be written as a sum of rank-one outer products $\mathbf{a}_i \otimes \mathbf{b}_i \otimes \mathbf{c}_i$ (the CP decomposition), as a small core tensor multiplied by a factor matrix along each mode (the Tucker decomposition), or as a chain of three-way cores (the tensor train of Oseledets (2011)). Unfortunately, nothing as clean as the Eckart–Young theorem holds in general for tensors: determining the CP rank is NP-hard, and a best approximation of a given CP rank may not even exist. The survey by Kolda and Bader (2009) is the standard guide to CP and Tucker decompositions and their algorithms.

33.6.2 Eigenvalues and Spectra

33.6.2.1 Computing Eigenvalues in Practice

There is no formula in radicals for the roots of a general polynomial of degree five or higher, so every practical eigensolver must iterate. Worse still, no production library forms the characteristic polynomial of Section 33.3.1.2, because the roots of a polynomial can be extraordinarily sensitive to tiny rounding errors in its coefficients. Instead, `eig` first applies orthogonal similarity transformations to reduce the matrix to Hessenberg form (zero everywhere below the first subdiagonal), which takes $O(n^3)$ operations. It then runs the shifted QR algorithm: factor $\mathbf{H} - \mu\mathbf{I} = \mathbf{Q}\mathbf{R}$, form $\mathbf{R}\mathbf{Q} + \mu\mathbf{I}$, and repeat. Every step is a similarity transformation, so the eigenvalues never change, while the subdiagonal entries shrink toward zero and the diagonal reveals the spectrum (complex conjugate pairs of a real matrix settle into 2×2 diagonal blocks). For symmetric matrices, the Hessenberg form is tridiagonal, and `eigh` exploits that structure for major speedups. Trefethen and Bau (1997) derive the

QR algorithm cleanly from power iteration on subspaces, and Golub and Van Loan (1996) supply the production engineering details.

33.6.2.2 Pseudospectra and the Kreiss Matrix Theorem

In Section 33.4.2.4, we saw that for a non-normal matrix, eigenvalues inside the unit disc do not prevent large transient growth in $\|\mathbf{A}^k\|_2$. The Kreiss matrix theorem turns the visual intuition of pseudospectra into a quantitative bound. Defining the Kreiss constant as $K(\mathbf{A}) = \sup_{|z|>1} (|z|-1)\|(z\mathbf{I}-\mathbf{A})^{-1}\|_2$, the theorem brackets the worst-case power norm for any $n \times n$ matrix: $K(\mathbf{A}) \le \sup_k \|\mathbf{A}^k\|_2 \le e\, n\, K(\mathbf{A})$. You can prove the lower bound in an exercise in Section 33.4; the upper bound guarantees that a matrix whose pseudospectra hug the unit disc closely cannot experience unbounded transient growth. Trefethen and Embree (2005) develop this theory thoroughly, including its classic application to fluid flows that transition to turbulence even when every linearized eigenvalue is strictly stable.

33.6.2.3 Mixing Times and the Spectral Gap

In the two-page Markov chain of Section 33.3, the distance to the stationary distribution shrinks by a factor of 4 at every click because the second eigenvalue is $1/4$. How many steps does a general Markov chain need to mix? For a reversible chain, let $\lambda_\star$ be the largest modulus among all eigenvalues other than 1. The number of steps required to come within total variation distance ε of the stationary distribution is bounded both above and below by multiples of $1/(1-\lambda_\star)$, where the constants depend on ε and on the smallest stationary probability. For a lazy chain—one that stays at its current state with probability at least $1/2$—all eigenvalues are nonnegative, so $\lambda_\star$ is simply the second eigenvalue λ_2. Cheeger's inequality then links the spectral gap $1-\lambda_2$ from both sides to the chain's *conductance*, the smallest probability flow leaving any subset relative to its stationary mass. In other words, you can read the convergence rate directly from the bottlenecks of the graph. Levin and Peres (2017) present this theory with great clarity, alongside the convergence results that Section 33.4.1.3 cites from Horn and Johnson (2012) .

33.6.2.4 Random Matrices and Trainable Depth

In Section 33.4.2.5, we observed that the spectral radii of individual layer Jacobians do not control the behavior of a long product of Jacobians. What, then, determines whether a very deep network can be trained from initialization? Saxe et al. (2014) solve the learning dynamics of deep linear networks exactly and show that orthogonal initial weights keep the training time independent of depth, whereas scaled Gaussian weights cause learning to slow down as depth grows. Extending the analysis to nonlinear networks, Pennington et al. (2017) use free probability to compute the full singular value distribution of the input–output Jacobian. They show that *dynamical isometry*, the regime in which all singular values cluster tightly near 1, is achievable with orthogonal weights and tanh activations but not with ReLU, and that networks initialized in this regime train substantially faster at large depth.

33.6.2.5 Spectra of Trained Weights

The Marchenko–Pastur law of Section 33.3.2.4 describes the singular value spectrum of a matrix of pure independent noise, which is what the weight matrices of a freshly initialized network look like. Martin and Mahoney (2021) compare the spectra of trained neural network weights against this random-matrix baseline and observe that training produces both

sharp outliers and a heavy, power-law tail. They interpret the tail exponent as a signature of implicit self-regularization. While this departure from Marchenko–Pastur appears consistently across many architectures, using the tail exponent to rank model quality without evaluating on test data remains an empirical correlation rather than a proven theorem.

33.6.3 Low Rank Beyond the Truncated SVD

33.6.3.1 Matrix Completion and the Nuclear Norm

A movie-ratings matrix might have only a few percent of its entries observed, whereas the Eckart–Young theorem (Section 33.5.2.1) requires knowing the entire matrix up front. Matrix completion bridges this gap by minimizing the *nuclear norm* $\|\mathbf{Z}\|_* = \sum_i \sigma_i$, which serves as a convex stand-in for matrix rank in the same way that the ℓ_1 norm stands in for vector sparsity, subject to matching all observed entries. Candès and Recht (2009) prove that this convex optimization problem recovers an underlying low-rank matrix exactly, with high probability, from far fewer than n^2 randomly observed entries, provided its singular vectors are spread out across the coordinates rather than concentrated on a few spikes. Soft-Impute (Mazumder et al. 2010) solves the penalized version of this problem by alternating a fill-in step with soft-thresholding of the singular values.

33.6.3.2 CUR Decompositions and Sketching

Although the truncated SVD is optimal in approximation error, its singular vectors mix together all rows and columns, making them difficult to interpret, and computing a full SVD of an $m \times n$ matrix costs $O(mn \min(m, n))$ operations. A CUR decomposition instead builds an approximation from actual columns $\mathbf{C}$ and actual rows $\mathbf{R}$ of the data matrix, linked by a small core matrix $\mathbf{U}$. In Section 33.5.2.1, we saw that the naive strategy of picking the columns with the largest norms loses badly to the truncated SVD. Mahoney and Drineas (2009) show how to fix this: sampling columns and rows according to their *leverage scores*, the squared lengths of their projections onto the top singular vectors, gets within a small constant factor of the Eckart–Young error with high probability.

More broadly, the randomized range finder of Section 33.5.4.3 (Halko et al. 2011) is one example of *sketching*: multiply a large matrix by a thin random matrix and solve the problem on the much smaller sketch. Woodruff (2014) surveys how this same idea accelerates least squares and regression, returning a solution within a $1 + \varepsilon$ factor of optimal from a sketch with far fewer rows than the original dataset.

33.6.3.3 Adapter Variants

LoRA (Section 33.5.4.1) initializes its low-rank update $\mathbf{BA}$ with $\mathbf{B} = \mathbf{0}$ and a random $\mathbf{A}$, which means the earliest gradient steps see the input only through a random k-dimensional subspace unrelated to the pretrained weight. Several variants change this initialization or parameterization in simple ways. PiSSA (Meng et al. 2024) initializes $\mathbf{BA}$ to the rank-k truncated SVD of the pretrained weight and freezes the residual, so fine-tuning immediately adapts the principal directions of the original matrix. DoRA (S.-Y. Liu et al. 2024) decomposes each weight column into a scalar magnitude and a unit direction, trains the magnitudes directly, and applies the low-rank update only to the directions.

33.6.3.4 Low Rank in Attention

Self-attention over a sequence of n tokens forms an $n \times n$ score matrix, and several efficient architectures in the attention chapters replace either that matrix or its key–value cache with a low-rank factorization. Linear attention (Section 10.6) expresses the attention scores as feature dot products $\phi(\mathbf{q}) \cdot \phi(\mathbf{k})$, capping the rank of the score matrix at the feature dimension. Multi-head latent attention (Section 11.3.3.2) (DeepSeek-AI 2024) compresses the cache by storing a single low-dimensional latent vector per token instead of separate keys and values for every head. Linformer (Wang et al. 2020) starts from the empirical observation that trained attention matrices have rapidly decaying singular values, and projects the keys and values along the sequence dimension down to a fixed length.

33.6.3.5 Steepest Descent under Other Norms

In Section 33.5.4.2, we computed the polar factor $\mathbf{U}\mathbf{V}^\top$ of a gradient matrix. Geometrically, this matrix is the steepest-descent direction when step sizes are measured in the spectral norm rather than the Euclidean norm. Bernstein and Newhouse (2024) turn that observation into a unifying lens for deep-learning optimizers: with their moving averages turned off, Adam reduces to sign descent (steepest descent under the ℓ_∞ norm), while Shampoo reduces to steepest descent under the spectral norm. From this viewpoint, choosing the right norm for each layer is the core design decision, and the update rule follows automatically. The spectral-norm optimizer built on this idea is developed in Section 9.6.

33.6.4 Books, Courses, and Notes

Books

- Mathematics for Machine Learning by Deisenroth, Faisal, and Ong: a free textbook whose Chapters 2 (Linear Algebra), 3 (Analytic Geometry), and 4 (Matrix Decompositions) cover the material of this chapter from a machine-learning perspective.
- Introduction to Applied Linear Algebra: Vectors, Matrices, and Least Squares by Boyd and Vandenberghe: a free book with slides and code organized around least squares and data, pairing well with Section 33.1 and Section 33.5.
- Introduction to Linear Algebra, 6th ed. and Linear Algebra and Learning from Data by Gilbert Strang: the classic introduction built around the column picture and the four fundamental subspaces, and its SVD-centered companion for data analysis and optimization.
- Linear Algebra Done Right, 4th ed. by Sheldon Axler: a free, determinant-free treatment of eigenvalues, operators, and the spectral theorem (Section 33.3).
- Numerical Linear Algebra by Trefethen and Bau: an accessible treatment of conditioning, stability, QR, and the floating-point algorithms behind `solve`, `eig`, and `svd`.

Courses and Tutorials

- MIT 18.06 Linear Algebra and MIT 18.065 Matrix Methods in Data Analysis, Signal Processing, and Machine Learning by Gilbert Strang: complete lecture courses with notes and problem sets.
- Essence of Linear Algebra by 3Blue1Brown and Singular Value Decomposition by Steve Brunton: visual lectures on linear maps, determinants, eigenvectors, and the SVD.
- Linear Algebra Review and Reference by Kolter and Do (Stanford CS229) and Immersive Linear Algebra by Ström, Åström, and Akenine-Möller.

Chapter 34

Calculus and Automatic Differentiation

Throughout this book, training a neural network follows the same rhythm: evaluate a loss, ask the framework for its gradient, and nudge every weight a small step downhill. We have relied on that loop without pausing to ask two natural questions about how it works. First, suppose a training run suddenly diverges after we double the learning rate. The gradient itself is still correct, yet the step has carried us past the small neighborhood where a local slope reliably predicts the loss. How far can that linear approximation be trusted? Second, notice how remarkably cheap the gradient is. Estimating every slope by perturbing a hundred million weights one at a time would require a hundred million forward passes, yet a single backward call returns all of them for roughly the cost of a few forward passes. How is that possible?

Both questions belong to calculus, a subject born in the seventeenth century when Newton and Leibniz asked how to measure continuous change and how to add up infinitely many tiny pieces. Their two tools, the derivative and the integral, turn out to be inverses of each other. In deep learning, the derivative supplies the best linear model of a loss surface near our current weights, and aside from the loss value itself, it is the only information an optimizer ever sees. The integral enters as soon as data or weights are random: a probability or an expected loss is an area under a curve, and methods such as policy gradients (Section 14.7) must differentiate that integral with respect to the very distribution they sample from.

Earlier in the book, we stated the basic rules of differentiation (Section 1.4) and treated automatic differentiation as a ready-made tool (Section 1.5). The main chapters require nothing more, so this appendix chapter is optional reading for readers who want to see why the machinery works and where its guarantees come from. Three results do most of the heavy lifting: the descent lemma, which converts a bound on curvature into the safe step-size rule quoted in Section 9.1; the chain rule written as a product of Jacobians, which becomes far cheaper when multiplied from the loss backward to give backpropagation (Section 4.3); and the two complementary estimators for the gradient of an expectation, which trade generality against variance.

The chapter unfolds in three parts. We first build linear and quadratic approximations of a function, in one variable (Section 34.1) and then in many (Section 34.2), so we can bound

safe step sizes and tell a minimum from a saddle point. Next, we follow the chain rule from a sum over paths in a computation graph (Section 34.3) to products of layer Jacobians (Section 34.4), and finally to the algorithms that multiply them automatically (Section 34.5). The closing part introduces integration (Section 34.6) and brings sampling and differentiation together to estimate gradients of expectations (Section 34.7).

If you are already comfortable with introductory calculus, feel free to skim the warm-up pages that open several sections. We omit vector calculus beyond the gradient (such as divergence and Stokes' theorem) as well as measure theory; for differential equations, see Chapter 38.

34.1 Single Variable Calculus

Every model in the Basics part was trained by gradient descent. At each step, the optimizer evaluates the loss and its slope at the current weights, nudges the weights a little downhill, and repeats. Because it never inspects the loss anywhere else, every step rests on a local prediction: the slope at the current point is assumed to describe the loss a short distance away. How accurate is that prediction, and how large a step can it safely support?

A single weight already shows what the answer looks like. Consider one training example, one scalar weight w, and the logistic loss $\ell(w) = \log(1 + e^{-w})$. At $w = 0$, the loss is 0.693147 and its slope is $-\frac{1}{2}$. If we move the weight to $w = 0.1$, a straight line through $w = 0$ with that slope predicts a loss of $0.693147 - 0.05 = 0.643147$. The actual loss is 0.644397, so the linear prediction is off by only 0.00125. Remarkably, that small gap has a simple structure: to the digits shown, it equals $\frac{1}{2} \cdot \frac{1}{4} \cdot 0.1^2$, where the factor $\frac{1}{4}$ measures how sharply the graph of ℓ bends at $w = 0$. The slope makes the prediction, and the bend tells us how quickly that prediction drifts off as the step grows. We will derive both constants shortly, once we have the rules of differentiation in hand.

This section develops the local straight-line model from the first derivative and extends it, using the second derivative, to a parabola that also captures the bend. Three results anchor the development: the Mean Value Theorem, which connects slopes at points to changes over intervals; the Lagrange remainder, which bounds the error of both the line and the parabola; and the descent lemma, which turns a bound on curvature into a guaranteed safe step size for gradient descent. We also meet Newton's method and the corner of ReLU, where no tangent line exists yet backpropagation must still return a number. Here we study all of these ideas in one variable; their multivariable optimization theory belongs to Chapter 35.

Earlier, Section 1.4 introduced the derivative, listed the rules of differentiation, and used automatic differentiation as a practical tool. Here we review those rules briefly and focus on the proofs, geometric intuition, and error bounds. If you are already comfortable with the differentiation rules, you can jump straight to Section 34.1.3. One variable is the natural starting point because every multivariable loss contains it: holding all weights of $\ell(w_1, \dots, w_n)$ fixed except w_i yields a one-dimensional slice, and single-variable calculus is the study of such slices.

Our running example throughout this section is the *softplus* function, $s(x) = \log(1 + e^x)$. It is the one-variable slice $\log(e^0 + e^x)$ of the log-sum-exp function used across this chapter, and the logistic loss above is simply $\ell(w) = s(-w)$.

34.1.1 The Derivative

34.1.1.1 The Difference Quotient

Zoom in far enough on the graph of a smooth function f near a point x, and the curve becomes indistinguishable from a straight line (Figure 34.1). The slope of that line is the derivative.

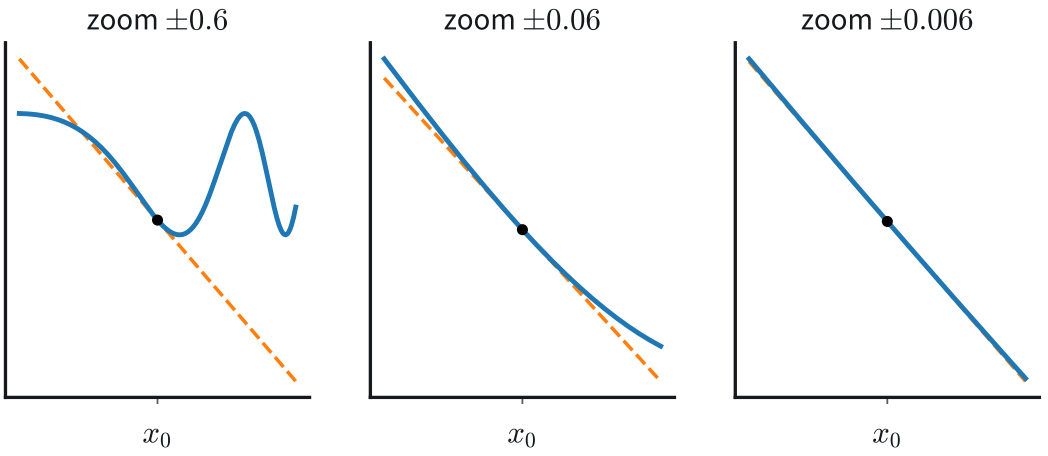


Figure 34.1: The curve $\sin(x^x)$ near $x_0 = 2$, shown in windows of half-width 0.6, 0.06, and 0.006. As the window shrinks, the curve (solid) becomes indistinguishable from its tangent line (dashed).

To capture that slope algebraically, we follow Section 1.4 and start with the *secant* line through $(x, f(x))$ and $(x+\epsilon, f(x+\epsilon))$. Its slope is the *difference quotient* $(f(x+\epsilon)-f(x))/\epsilon$. As ϵ shrinks toward zero, the secant rotates toward the tangent line (Figure 34.2).

How small must ϵ be before the secant slope is a trustworthy stand-in for the tangent slope? Unfortunately, there is no universal threshold: it depends entirely on the function. Consider the polynomial $f(x) = x^2 + 1701(x-4)^3$, whose true slope at $x = 4$ is 8, and observe how slowly the secant slope settles as ϵ shrinks.

```
def f(x):
    return x**2 + 1701*(x-4)**3

# Slope of the secant through x=4, for shrinking epsilon
for epsilon in [0.1, 0.001, 0.0001, 0.00001]:
    print(f'epsilon = {epsilon:.5f} -> {(f(4+epsilon) - f(4)) / epsilon:.5f}')
```

```
epsilon = 0.10000 -> 25.11000
epsilon = 0.00100 -> 8.00270
epsilon = 0.00010 -> 8.00012
epsilon = 0.00001 -> 8.00001
```

At $\epsilon = 0.1$, the secant slope is 25.11, more than three times the true limit, and the printed values only settle near 8 once ϵ reaches 0.001. Expanding $f(4+\epsilon)-f(4) = 8\epsilon+\epsilon^2+1701\epsilon^3$ shows why: the difference quotient equals $8 + \epsilon + 1701\epsilon^2$, so the cubic term adds 17 at

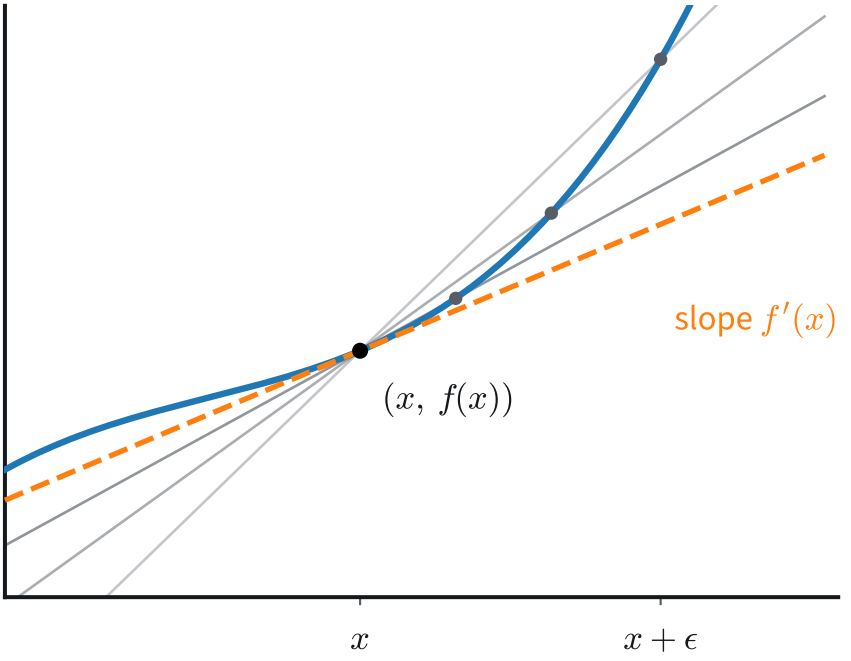


Figure 34.2: The difference quotient is the slope of the secant through $(x, f(x))$ and $(x+\epsilon, f(x+\epsilon))$. As $\epsilon \to 0$, this slope approaches the tangent slope $f'(x)$.

$\epsilon = 0.1$ but only 1.7×10^{-7} at $\epsilon = 10^{-5}$. The large coefficient delays the approach to the limit without changing it. Automatic differentiation sidesteps this problem altogether: it never chooses an ϵ, instead applying the rules of differentiation directly (Section 34.5).

34.1.1.2 The Derivative and the Linear Approximation

The **derivative** of f at x is the limit of the secant slopes,

$$f'(x) = \frac{df}{dx}(x) = \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon}, \tag{34.1}$$

whenever this limit exists, in which case we say f is *differentiable* at x. For the cubic above, $f'(4) = 8$. We write f' or $\frac{df}{dx}$, and $\frac{d}{dx}[\ldots]$ when differentiating a bulky expression. The limit can also fail to exist, as it does for $|x|$ at 0; we return to such corners in Section 34.1.6.

It is instructive to rearrange this definition. Let $r(\epsilon) = f(x+\epsilon) - f(x) - \epsilon f'(x)$ denote the error of the straight-line prediction; the limit Equation 34.1 says that $r(\epsilon)/\epsilon \to 0$. Writing $o(\epsilon)$ for any quantity $r(\epsilon)$ satisfying $r(\epsilon)/\epsilon \to 0$ as $\epsilon \to 0$, we obtain the **linear approximation**

$$f(x+\epsilon) = f(x) + \epsilon f'(x) + o(\epsilon). \tag{34.2}$$

In plain words, a small step ϵ changes the output by $\epsilon f'(x)$, up to an error that vanishes faster than ϵ. The converse holds too: if $f(x+\epsilon) = f(x) + a\epsilon + o(\epsilon)$ for some constant a,

then dividing by ϵ and letting $\epsilon \to 0$ proves that f is differentiable at x with $f'(x) = a$. The expansion Equation 34.2 and the limit Equation 34.1 therefore carry the same information, and our proofs of the sum and chain rules will lean on this equivalence.

Plotted as a function of $x+\epsilon$, the line $f(x)+\epsilon f'(x)$ is the *tangent* to the graph at x, the limit of the rotating secants. This is the line that predicted the logistic loss at $w = 0.1$, and the gap of 0.00125 was its $o(\epsilon)$ remainder. By itself, $o(\epsilon)$ promises very little: an error of order $o(\epsilon)$ could shrink only slightly faster than ϵ. Here, however, the gap shrank as ϵ^2, and as we will see in Section 34.1.5.2, quadratic error decay is the rule for every twice differentiable function.

34.1.2 Computing Derivatives

In practice, nobody computes derivatives by taking limits. Instead, as in Section 1.4, we combine a short table of elementary derivatives with rules for sums, products, and compositions. Here we prove each rule directly from the linear approximation. Among the table entries, we derive the exponential, the logarithm, and the power rule; we take the sine and cosine entries, along with the existence of a base whose exponential has slope 1 at 0, as known from a first calculus course.

34.1.2.1 A Table of Common Derivatives

The following formulas hold wherever the right-hand sides are defined: the power rule for every real n when $x > 0$ and for integer n wherever x^n is defined, and the logarithm for $x > 0$.

- **Derivative of constants.** $\frac{d}{dx} c = 0$.
- **Derivative of linear functions.** $\frac{d}{dx}(ax) = a$.
- **Power rule.** $\frac{d}{dx} x^n = n x^{n-1}$.
- **Derivative of exponentials.** $\frac{d}{dx} e^x = e^x$.
- **Derivative of the logarithm.** $\frac{d}{dx} \log(x) = \frac{1}{x}$.
- **Derivative of sine.** $\frac{d}{dx} \sin(x) = \cos(x)$.
- **Derivative of cosine.** $\frac{d}{dx} \cos(x) = -\sin(x)$.

The exponential entry rests on a one-line observation that also reveals what makes the base e special. For any $a > 0$, the difference quotient of a^x factors cleanly as

$$\frac{a^{x+\epsilon} - a^x}{\epsilon} = a^x \, \frac{a^\epsilon - 1}{\epsilon},$$

where the second factor is simply the difference quotient at 0. Thus, if a^x is differentiable at 0 with slope c_a, then $\frac{d}{dx} a^x = c_a \, a^x$ for every x: the slope of an exponential is always proportional to its height, with a single proportionality constant per base. Numerically, at $\epsilon = 10^{-6}$ we find $(2^\epsilon - 1)/\epsilon = 0.69315$ for base 2 and $(10^\epsilon - 1)/\epsilon = 2.30259$ for base 10. The number e is defined as the unique base whose constant is 1, which immediately yields $\frac{d}{dx} e^x = e^x$. Once we have the derivative of the logarithm, the chain rule will identify c_a for every other base.

34.1.2.2 Rules for Combining Derivatives

A loss function builds on the table above using sums, products, quotients, and compositions. Let g and h be differentiable at x (for the chain rule, assume h is differentiable at x and g is differentiable at $h(x)$). Then:

- **Sum rule.** $\frac{d}{dx}\big(g(x)+h(x)\big) = g'(x)+h'(x)$.
- **Product rule.** $\frac{d}{dx}\big(g(x)\,h(x)\big) = g(x)h'(x)+g'(x)h(x)$.
- **Quotient rule.** $\frac{d}{dx}\frac{g(x)}{h(x)} = \frac{g'(x)h(x)-g(x)h'(x)}{h(x)^2}$, where $h(x)\neq 0$.
- **Chain rule.** $\frac{d}{dx}g(h(x)) = g'(h(x))\,h'(x)$.

The sum and product rules are routine; only the chain rule requires real care, because the naive argument divides by $h(x+\epsilon)-h(x)$, which can vanish even when $\epsilon \neq 0$. The proofs all rely on one basic fact: **a function differentiable at x is continuous at x**, because $g(x+\epsilon)-g(x) = \epsilon\cdot\frac{g(x+\epsilon)-g(x)}{\epsilon}$ tends to $0\cdot g'(x)=0$ as $\epsilon\to 0$.

Sum rule. Adding the linear approximations for g and h gives

$$g(x+\epsilon)+h(x+\epsilon) = g(x)+h(x)+\epsilon\big(g'(x)+h'(x)\big)+o(\epsilon),$$

since the sum of two quantities that vanish faster than ϵ also vanishes faster than ϵ. By the converse of the linear approximation, the coefficient of ϵ is the derivative: first-order changes simply add. ■

Product rule. Adding and subtracting $g(x+\epsilon)h(x)$ in the numerator splits the difference quotient of the product into two quotients we already know:

$$\frac{g(x+\epsilon)h(x+\epsilon)-g(x)h(x)}{\epsilon} = g(x+\epsilon)\,\frac{h(x+\epsilon)-h(x)}{\epsilon} + h(x)\,\frac{g(x+\epsilon)-g(x)}{\epsilon}.$$

As $\epsilon\to 0$, continuity gives $g(x+\epsilon)\to g(x)$, so the right-hand side converges to $g(x)h'(x)+h(x)g'(x)$. ■

Chain rule. Rather than dividing by $h(x+\epsilon)-h(x)$, we feed the change of the inner function directly into the linear approximation of the outer function. Because g is differentiable at $h(x)$, the error ratio $\rho(t) = \frac{g(h(x)+t)-g(h(x))}{t} - g'(h(x))$ for $t\neq 0$, extended by $\rho(0)=0$, tends to 0 as $t\to 0$. For every t, including $t=0$, we can therefore write

$$g(h(x)+t) = g(h(x)) + \big(g'(h(x))+\rho(t)\big)\,t.$$

Now set $t = h(x+\epsilon)-h(x) = \epsilon h'(x)+o(\epsilon)$. By continuity of h, $t\to 0$ as $\epsilon\to 0$, so $\rho(t)\to 0$ as well. Substituting this t yields

$$\begin{aligned} g(h(x+\epsilon))-g(h(x)) &= \big(g'(h(x))+\rho(t)\big)\big(\epsilon h'(x)+o(\epsilon)\big)\\ &= \epsilon\,g'(h(x))\,h'(x)+o(\epsilon), \end{aligned}$$

because every term other than $g'(h(x))\,\epsilon h'(x)$ contains a factor that vanishes faster than ϵ. Applying the converse of the linear approximation finishes the proof. ■

Rates multiply along a composition. Applied layer by layer to a neural network, the chain rule produces a product of local slopes, and evaluating that product efficiently is precisely what backpropagation does. The quotient rule follows immediately by applying the product and chain rules to $g \cdot h^{-1}$.

34.1.2.3 Inverse Functions

The chain rule also differentiates a function that undoes another. A function g satisfying $g(f(x)) = x$ for all x near a point is an *inverse* of f there. Suppose such a g exists and is differentiable at $f(x)$; by the inverse function theorem (Rudin 1976), which we use without proof, both hold whenever f' is continuous and nonzero near x. Differentiating both sides of $g(f(x)) = x$ with the chain rule gives

$$g'(f(x))\, f'(x) = 1, \qquad \text{so} \qquad g'(y) = \frac{1}{f'(g(y))} \quad \text{at } y = f(x).$$

In other words, the slope of the inverse is the reciprocal of the slope of f, evaluated at the matching point. Geometrically, the graph of g is the graph of f reflected across the diagonal, and reflecting a line across the diagonal turns rise over run upside down.

Since the logarithm undoes e^x, this rule gives $\log'(y) = 1/e^{\log y} = 1/y$, the fifth entry in our table. Notice that the condition $f'(x) \neq 0$ is strictly necessary. For $f(x) = x^3$ we have $f'(0) = 0$, so the formula breaks down at $y = 0$; indeed, its inverse $x^{1/3}$ is not differentiable at 0, because the difference quotient $\epsilon^{1/3}/\epsilon = \epsilon^{-2/3}$ grows without bound and the graph has a vertical tangent there.

With the logarithm in hand, the chain rule settles the remaining two entries of the table. Writing $a^x = e^{x \log a}$ gives $\frac{d}{dx} a^x = (\log a)\, a^x$, identifying the constant $c_a = \log a$; indeed, $\log 2 = 0.69315$ and $\log 10 = 2.30259$ are the constants we measured numerically above. Similarly, writing $x^n = e^{n \log x}$ for $x > 0$ yields $\frac{d}{dx} x^n = e^{n \log x} \cdot \frac{n}{x} = n x^{n-1}$ for every real n.

34.1.2.4 The Softplus, the Sigmoid, and the Logit

We can now apply these rules to our running function, $s(x) = \log(1 + e^x)$. Using the chain rule with outer function log and inner function $1 + e^x$ gives

$$s'(x) = \frac{e^x}{1 + e^x} = \frac{1}{1 + e^{-x}} = \sigma(x),$$

so the derivative of the softplus is the *sigmoid* σ. Applying the chain rule a second time to $\sigma(x) = (1 + e^{-x})^{-1}$, with outer function u^{-1} and inner function $1 + e^{-x}$ of slope $-e^{-x}$, yields

$$\sigma'(x) = \frac{e^{-x}}{(1 + e^{-x})^2} = \sigma(x)(1 - \sigma(x)),$$

since $1 - \sigma(x) = e^{-x}/(1 + e^{-x})$. Because a product $p(1 - p)$ is maximized at $p = \frac{1}{2}$, the sigmoid is steepest at the origin, where $\sigma(0) = \frac{1}{2}$ and $\sigma'(0) = \frac{1}{4}$. The inverse of σ

is the *logit* function $\operatorname{logit}(p) = \log \frac{p}{1-p}$, and the inverse-function rule immediately gives $\operatorname{logit}'(p) = 1/\sigma'(\operatorname{logit} p) = \frac{1}{p(1-p)}$. At $p = \frac{1}{2}$, this equals $4 = 1/\frac{1}{4}$, matching the reciprocal of the sigmoid's slope.

We can now verify the two constants from the opening logistic-loss example. Since $\ell(w) = s(-w)$, the chain rule gives $\ell'(w) = -\sigma(-w)$, so $\ell'(0) = -\frac{1}{2}$. Differentiating once more gives $\ell''(w) = \sigma(-w)(1 - \sigma(-w))$ and thus $\ell''(0) = \frac{1}{4}$, the curvature at the origin. Let us check numerically whether the gap between $\ell(\epsilon)$ and its linear approximation $\ell(0) - \frac{1}{2}\epsilon$ indeed shrinks like ϵ^2 with leading constant $\frac{1}{2}\ell''(0) = \frac{1}{8}$.

```
# The logistic loss l(w) = log(1 + e^{-w}) near w = 0, where l'(0) = -1/2
def ell(w):
    return np.log(1 + np.exp(-w))

for eps in [0.1, 0.01, 0.001]:
    line = ell(0) - 0.5 * eps          # the linear approximation
    gap = ell(eps) - line
    print(f'eps = {eps:<5}: l = {ell(eps):.6f}, line = {line:.6f}, '
          f'gap = {gap:.2e}, gap / eps^2 = {gap / eps**2:.4f}')
```

```
eps = 0.1  : l = 0.644397, line = 0.643147, gap = 1.25e-03, gap / eps^2 = 0.1249
eps = 0.01 : l = 0.688160, line = 0.688147, gap = 1.25e-05, gap / eps^2 = 0.1250
eps = 0.001: l = 0.692647, line = 0.692647, gap = 1.25e-07, gap / eps^2 = 0.1250
```

Sure enough, the gap drops a hundredfold every time we cut ϵ by a factor of ten, and gap/ϵ^2 settles to $0.125 = \frac{1}{2}\ell''(0)$. The error of the linear approximation is indeed of second order, with a leading constant governed by the curvature. Notice also that the first line prints 0.1249 rather than 0.1250: the gap is not quite $\epsilon^2/8$, and Section 34.1.5.2 will identify the higher-order term responsible for that small difference.

So far, every calculation has lived in an infinitesimal neighborhood of a single point. Asking what the slope tells us about a function across an entire interval is a different kind of question.

34.1.3 The Mean Value Theorem

In optimization, we routinely act on two intuitions: a positive slope means the function is climbing, and a point where the slope vanishes is a candidate minimum. Gradient descent relies on both. Yet each claim infers the behavior of f over an interval from its derivative at individual points, and neither is built into the definition of the derivative. We prove the second claim first, as it is the simpler one.

Lemma (Fermat). *If f attains a local maximum or a local minimum at an interior point ξ of an interval and is differentiable at ξ, then $f'(\xi) = 0$.*

Proof. At a local maximum, $f(\xi+\epsilon) - f(\xi) \le 0$ for all sufficiently small ϵ of either sign. The difference quotient is therefore ≤ 0 for $\epsilon > 0$ and ≥ 0 for $\epsilon < 0$. Because f is differentiable at ξ, both one-sided limits must equal $f'(\xi)$, and the only number that is simultaneously ≤ 0 and ≥ 0 is zero. A local minimum follows by the same argument with inequalities reversed. ■

The first intuition requires a deeper tool: the Mean Value Theorem. Every error bound in

the remainder of this section is obtained by applying this theorem to a well-chosen auxiliary function. Geometrically (Figure 34.3), it states that the chord joining the endpoints of the graph over $[a, b]$ is parallel to the tangent line at some interior point.

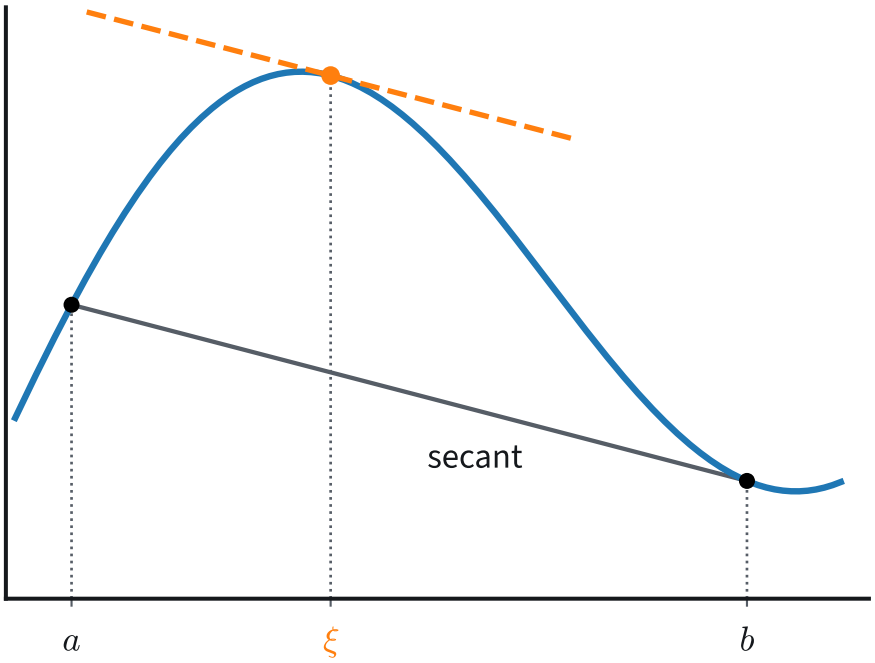


Figure 34.3: Over the interval from a to b, the secant chord through the endpoints has slope equal to the average rate of change; the Mean Value Theorem guarantees that some interior point ξ has a tangent of that same slope, parallel to the chord.

On a simple example like $f(x) = x^2$ over $[0, 2]$, the chord has slope $(4 - 0)/2 = 2$, and the tangent slope $f'(\xi) = 2\xi$ matches it at the midpoint $\xi = 1$. Notice, however, that the theorem truly requires differentiability at every interior point. The absolute value $|x|$ on $[-1, 1]$ satisfies $f(-1) = f(1)$, so its chord is horizontal, yet no interior point has slope 0: the slope is -1 on the left, $+1$ on the right, and undefined at 0.

Proposition (Mean Value Theorem). *If f is continuous on $[a, b]$ and differentiable on (a, b), then there exists a point $\xi \in (a, b)$ such that*

$$f'(\xi) = \frac{f(b) - f(a)}{b - a}. \tag{34.3}$$

Proof. Consider first the flat case $f(a) = f(b)$, known as Rolle's theorem. We invoke one foundational result from real analysis, the **extreme value theorem**: a continuous function on a closed interval attains both a maximum and a minimum (Rudin 1976). If both extrema occur at the endpoints, then f is constant and $f' \equiv 0$ everywhere inside. Otherwise, at least one extremum occurs at an interior point ξ, where $f'(\xi) = 0$ by Fermat's lemma.

For the general case, we subtract the chord to flatten the endpoints. Define the tilted function $\tilde{f}(x) = f(x) - \frac{f(b)-f(a)}{b-a}(x - a)$. Since $\tilde{f}(a) = \tilde{f}(b) = f(a)$, Rolle's theorem provides

an interior point ξ with $\tilde{f}'(\xi) = 0$, which rearranges directly to Equation 34.3. ■

In words, the average rate of change over an interval is always realized as an instantaneous rate of change at some point inside it. Two consequences will appear repeatedly throughout the chapter:

1. *If $f' > 0$ on an interval, then f is strictly increasing there.* For any $x_1 < x_2$ in the interval, Equation 34.3 gives $f(x_2) - f(x_1) = f'(\xi)(x_2 - x_1) > 0$ for some ξ between them. By the same token, $f' < 0$ implies f is strictly decreasing, and $f' = 0$ implies f is constant.
2. *If $f' < 0$ on $(x_0 - \delta, x_0)$ and $f' > 0$ on $(x_0, x_0 + \delta)$ for some $\delta > 0$, and f is continuous at x_0, then x_0 is a strict local minimum.* Applying the first consequence on each side shows that f falls into x_0 from the left and rises out of it to the right.

Returning to our running function, $s' = \sigma > 0$ everywhere, so the softplus is strictly increasing on all of $\mathbb{R}$.

34.1.4 Gradient Descent in One Variable

34.1.4.1 The Gradient-Descent Step

The linear approximation predicts how f responds to any small step ϵ: $f(x + \epsilon) \approx f(x) + \epsilon f'(x)$. To decrease f, we should choose ϵ with the sign opposite to the slope. Setting $\epsilon = -\eta f'(x)$ for a positive *step size* $\eta > 0$, the linear model predicts the change

$$f(x - \eta f'(x)) - f(x) \approx -\eta\, f'(x)^2,$$

which is strictly negative whenever $f'(x) \neq 0$. Regardless of whether the slope is positive or negative, stepping opposite to it lowers the linear model by an amount proportional to the squared slope (Figure 34.4).

Iterating this update gives one-dimensional **gradient descent**:

$$x_{t+1} = x_t - \eta\, f'(x_t). \tag{34.4}$$

Each step moves against the local slope, and the iteration comes to rest at a *stationary point*, where $f'(x_t) = 0$. In a neural network, every coordinate of the weight update $w_i \leftarrow w_i - \eta\, \partial \ell / \partial w_i$ takes this form, with all coordinates updated simultaneously. On our logistic loss starting at $w = 0$, the slope is $\ell'(0) = -\frac{1}{2}$, so a step of size η moves the weight by $\epsilon = \eta/2$ and predicts a loss drop of $\eta/4$.

Notice a glaring limitation of the linear model: its predicted change $-\eta f'(x)^2$ is negative for every $\eta > 0$, no matter how huge. In reality, too large a step can overshoot the minimum and land higher than where it began. The tangent line cannot warn us of this danger because it knows nothing about how the curve bends. The simplest bowl, $f(x) = x^2$, already reveals where the linear prediction breaks down.

34.1.4.2 The Quadratic Case

For $f(x) = x^2$, the slope is $f'(x) = 2x$, so the gradient-descent update simplifies to

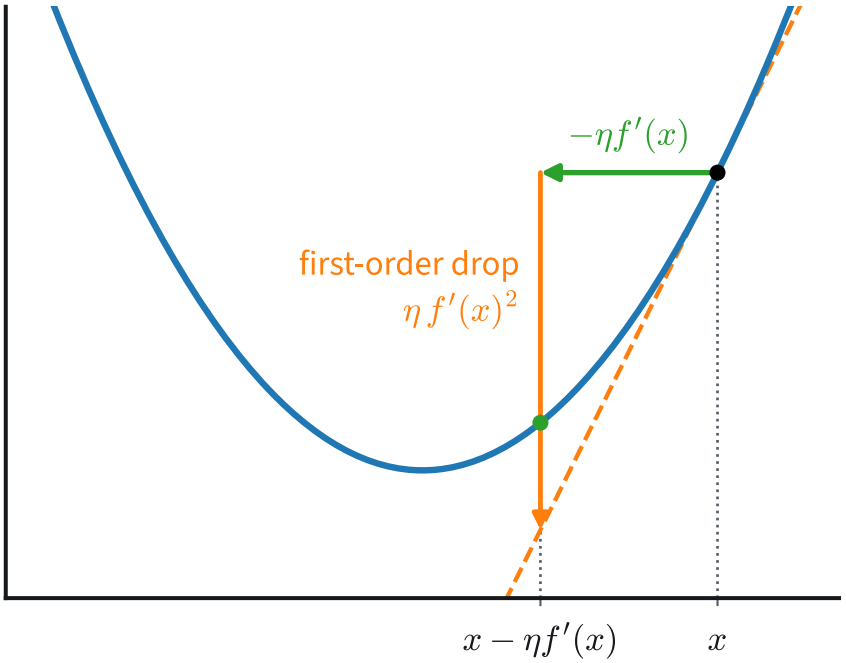


Figure 34.4: One gradient-descent step on a 1-D bowl. At x the tangent line has slope $f'(x)$; the step $-\eta f'(x)$ moves opposite the slope and lands at $x - \eta f'(x)$, lower on the curve, with the drop equal to $\eta f'(x)^2$ to first order.

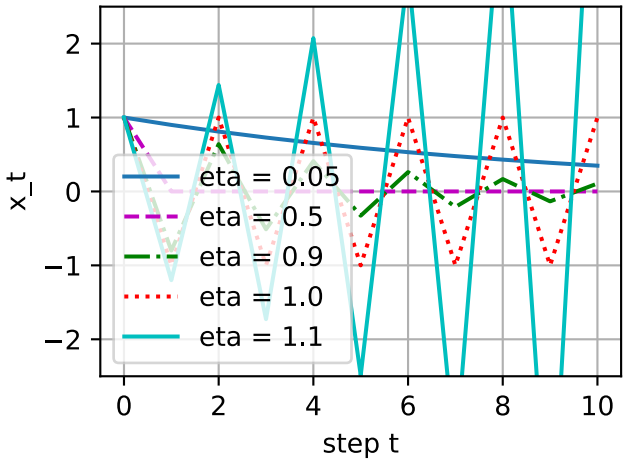
$$x_{t+1} = x_t - \eta\,(2x_t) = (1 - 2\eta)\,x_t.$$

Every step multiplies the current iterate by the same factor $1 - 2\eta$, giving the closed form $x_t = (1 - 2\eta)^t x_0$. The iterates therefore converge to the minimum $x = 0$ if and only if $|1 - 2\eta| < 1$, which is $0 < \eta < 1$. Within this convergent range, a small η creeps monotonically toward 0, $\eta = \frac{1}{2}$ lands on the minimum in a single step, and $\frac{1}{2} < \eta < 1$ overshoots and oscillates inward. Right at the boundary $\eta = 1$, the iterates bounce forever between $+x_0$ and $-x_0$, and for $\eta > 1$ they blow up.

Let us run ten steps from $x_0 = 1$ across five step sizes, one from each regime, to see these trajectories side by side.

```
# Gradient descent on f(x) = x^2 from x0 = 1, for several step sizes
def gd(eta, steps=10, x0=1.0):
    xs = [x0]
    for _ in range(steps):
        xs.append(xs[-1] - eta * (2 * xs[-1]))  # x <- x - eta * f'(x)
    return xs

etas = [0.05, 0.5, 0.9, 1.0, 1.1]
trajectories = [gd(eta) for eta in etas]
for eta, xs in zip(etas, trajectories):
    print(f'eta = {eta:.2f} -> x_10 = {xs[-1]:+.5f}')
d2l.plot(list(range(11)), trajectories, 'step t', 'x_t', ylim=[-2.5, 2.5],
         legend=[f'eta = {eta}' for eta in etas],
         fmts=('-', 'm--', 'g-.', 'r:', 'c-'))
```

```
eta = 0.05 -> x_10 = +0.34868
eta = 0.50 -> x_10 = +0.00000
eta = 0.90 -> x_10 = +0.10737
eta = 1.00 -> x_10 = +1.00000
eta = 1.10 -> x_10 = +6.19174
```



The printed values match the closed-form powers $(1-2\eta)^{10}$: the cautious step $\eta = 0.05$ has only crawled down to 0.349, $\eta = 0.9$ has zig-zagged down to 0.107, $\eta = 1$ is right back at 1,

and $\eta = 1.1$ has already diverged to 6.19. Notice that the stability threshold $\eta = 1$ equals $2/f''$, because $f'' \equiv 2$ for x^2. As we will see in Section 34.1.5.4, this is no coincidence: any bound L on the curvature gives the safe step-size threshold $2/L$ for every smooth function.

34.1.5 Curvature and Taylor Series

How far can we trust the linear approximation, and how large a step size is safe? Both questions depend on how fast the slope itself changes.

34.1.5.1 The Second Derivative

Because f' is itself a function of x, we can differentiate it again. The result is the *second derivative*, $f'' = \frac{d^2 f}{dx^2} = \frac{d}{dx}(\frac{df}{dx})$, which measures the rate of change of the slope. Repeating this process n times yields the n-th derivative,

$$f^{(n)}(x) = \frac{d^n f}{dx^n} = \left(\frac{d}{dx}\right)^n f.$$

Geometrically, $f'' > 0$ means the slope is increasing, so the graph curves upward like a bowl; $f'' < 0$ means the slope is decreasing, so the graph curves downward like a dome. At a stationary point, this sign tells a local minimum apart from a local maximum.

Proposition (second-derivative test). *Let $f'(x_0) = 0$ and suppose $f''(x_0)$ exists. If $f''(x_0) > 0$, then x_0 is a strict local minimum; if $f''(x_0) < 0$, it is a strict local maximum.*

Proof. Because $f'(x_0) = 0$, the definition of $f''(x_0)$ simplifies to $f''(x_0) = \lim_{\epsilon \to 0} f'(x_0 + \epsilon)/\epsilon$. If this limit is positive, then the quotient $f'(x_0 + \epsilon)/\epsilon$ must be positive for all sufficiently small $\epsilon \neq 0$. Hence $f'(x_0 + \epsilon)$ shares the sign of ϵ: negative just to the left of x_0 and positive just to the right. By the second consequence of the Mean Value Theorem, x_0 is a strict local minimum. The case $f''(x_0) < 0$ follows identically with all signs reversed. ■

Notice where the real work happened in that proof: turning a local sign pattern of f' into a minimum over an interval requires the Mean Value Theorem. Note also that the test certifies only a neighborhood of x_0, not the entire domain. When we move to many variables, the signs of the eigenvalues of the matrix of second derivatives will play this same role.

When $f''(x_0) = 0$ at a stationary point, the test is silent (Figure 34.5, bottom row): x^3, x^4, and $-x^4$ all satisfy $f'(0) = f''(0) = 0$, yet 0 is an inflection point for the first, a minimum for the second, and a maximum for the third; higher-order derivatives decide such flat stationary points.

34.1.5.2 The Taylor Polynomial

The tangent line matches the value and the slope of f at a base point x_0. To capture the bend as well, we can fit a parabola. A general quadratic $q(x) = ax^2 + bx + c$ has derivatives

$$q'(x) = 2ax + b, \qquad q''(x) = 2a,$$

so its curvature is the constant $2a$. Matching the curvature of f at x_0 fixes $a = f''(x_0)/2$, and matching the value and slope fixes b and c. For $f(x) = \sin(x)$, where $\sin' = \cos$ and $\sin'' = -\sin$, the best-fitting parabola at x_0 is

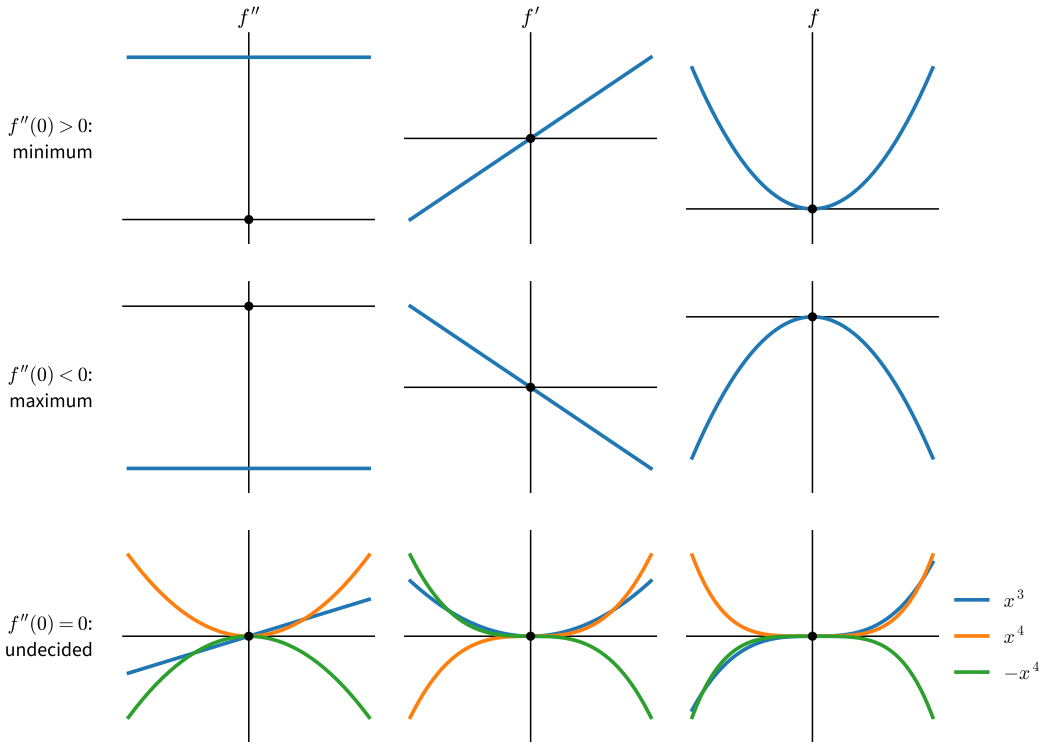


Figure 34.5: The second-derivative test. Rows: $f''(0) > 0$ (a bowl, minimum), $f''(0) < 0$ (a dome, maximum), and $f''(0) = 0$, where x^3, x^4, and $-x^4$ share $f'(0) = f''(0) = 0$ yet have an inflection point, a minimum, and a maximum at 0. Columns: f'', f', f.

$$q(x) = \sin(x_0) + \cos(x_0)\,(x - x_0) - \tfrac{1}{2}\sin(x_0)\,(x - x_0)^2.$$

Because this parabola follows the bend of the sine curve, it stays accurate over a noticeably wider interval than the tangent line (Figure 34.6, drawn at a base point where $\sin x_0 \neq 0$ so the parabola has nonzero curvature).

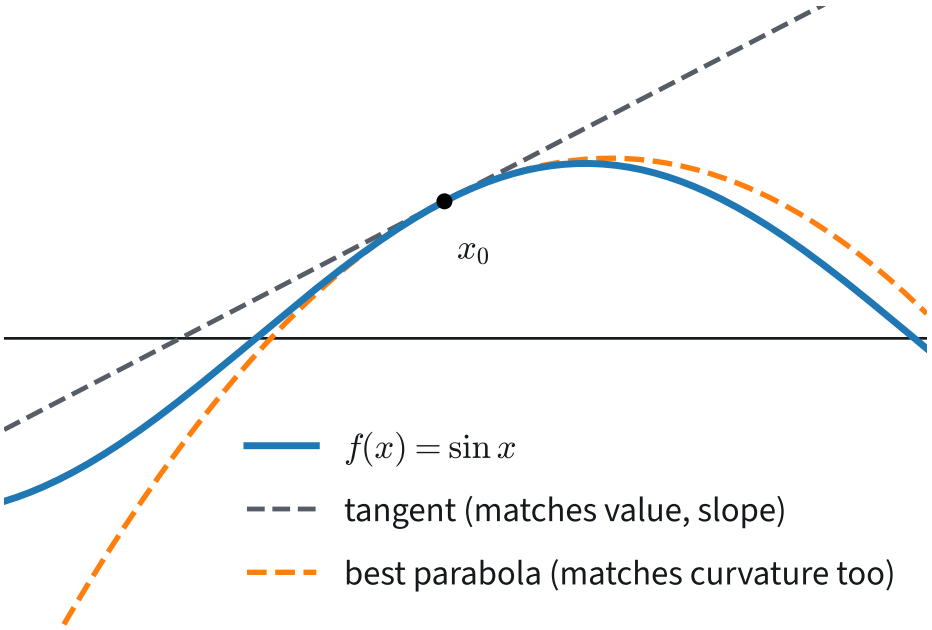


Figure 34.6: The sine curve with its two local models at x_0: the tangent line matches value and slope, while the best parabola also matches curvature and remains accurate over a wider interval.

For our running function at $x_0 = 0$, the values $s(0) = \log 2$, $s'(0) = \frac{1}{2}$, and $s''(0) = \frac{1}{4}$ produce the best parabola $\log 2 + \frac{x}{2} + \frac{x^2}{8}$. Its quadratic coefficient $\frac{1}{8}$ is the exact constant to which gap$/\epsilon^2$ converged for the logistic loss $\ell(w) = s(-w)$.

The same idea extends to any degree n. The degree-n **Taylor polynomial** of f at x_0 is

$$P_n(x) = \sum_{i=0}^{n} \frac{f^{(i)}(x_0)}{i!}(x - x_0)^i,$$

which is the unique polynomial of degree at most n whose value and first n derivatives agree with those of f at x_0. The factorials appear because differentiating $(x - x_0)^i$ exactly i times at x_0 produces $i!$, while every other term vanishes. In this notation, P_1 is the tangent line and P_2 is the best parabola. For e^x at $x_0 = 0$, where every derivative equals 1, we obtain $P_n(x) = \sum_{i=0}^{n} x^i/i!$, and each new term lets the polynomial hug the exponential curve a little farther from the origin. To know just how far we can trust P_n, and how rapidly its error grows beyond that range, we need an explicit formula for the remainder.

34.1.5.3 The Lagrange Remainder

The Mean Value Theorem yields an exact formula for the error of P_n, expressed in terms of a single unknown interior point. This is the centerpiece of the section: both the descent lemma and Newton's quadratic convergence rate fall out of its $n = 1$ case.

Proposition (Lagrange remainder). *If f is $(n+1)$-times differentiable on an open interval containing x_0 and x, with $x \neq x_0$, then*

$$f(x) = P_n(x) + R_n(x), \qquad R_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}\,(x - x_0)^{n+1}, \tag{34.5}$$

for some ξ strictly between x_0 and x. Note that the case $n = 0$ recovers the Mean Value Theorem, $f(x) = f(x_0) + f'(\xi)(x - x_0)$.

Proof. The one clever step is choosing the right auxiliary function; everything after that is straightforward differentiation. Fix x and treat the base point t as the variable: for t between x_0 and x, define

$$G(t) = f(x) - \sum_{k=0}^{n} \frac{f^{(k)}(t)}{k!}\,(x - t)^k - M\,\frac{(x - t)^{n+1}}{(n+1)!},$$

which is the error at x of the Taylor polynomial centered at t, minus a term of the form we expect that error to take. Choose the constant M so that $G(x_0) = 0$. The proof now finishes in three steps.

First, G vanishes at both endpoints. Our choice of M ensures $G(x_0) = 0$, while $G(x) = 0$ because every positive power of $(x - t)$ vanishes at $t = x$ and the $k = 0$ term leaves $f(x) - f(x) = 0$. Rolle's theorem therefore guarantees a point ξ strictly between x_0 and x where $G'(\xi) = 0$.

Second, differentiate $G(t)$ with respect to t. By the product rule, each summand with $k \geq 1$ contributes

$$-\frac{f^{(k+1)}(t)}{k!}(x - t)^k + \frac{f^{(k)}(t)}{(k-1)!}(x - t)^{k-1}$$

to $G'(t)$, while the $k = 0$ term contributes $-f'(t)$. Adjacent terms cancel in pairs (the sum telescopes), leaving only the highest-order derivative and the derivative of the M term:

$$G'(t) = -\frac{f^{(n+1)}(t)}{n!}\,(x - t)^n + M\,\frac{(x - t)^n}{n!}.$$

Third, evaluate at $t = \xi$. Because ξ lies strictly between x_0 and x, we have $(x - \xi)^n \neq 0$, so $G'(\xi) = 0$ forces $M = f^{(n+1)}(\xi)$. Plugging this back into $G(x_0) = 0$ yields Equation 34.5. ∎

The remainder looks just like the next term of the Taylor polynomial, except that the $(n+1)$-st derivative is evaluated at an unknown interior point ξ rather than at x_0. Whenever $f^{(n+1)}$

stays bounded near x_0, the error therefore scales as $|x - x_0|^{n+1}$: each additional matched derivative raises the order of accuracy by one.

Let us test this on our running function. For the softplus, $s''' = \sigma(1-\sigma)(1-2\sigma)$ vanishes at 0 and $s''''(0) = -\frac{1}{8}$, giving the fourth-degree polynomial

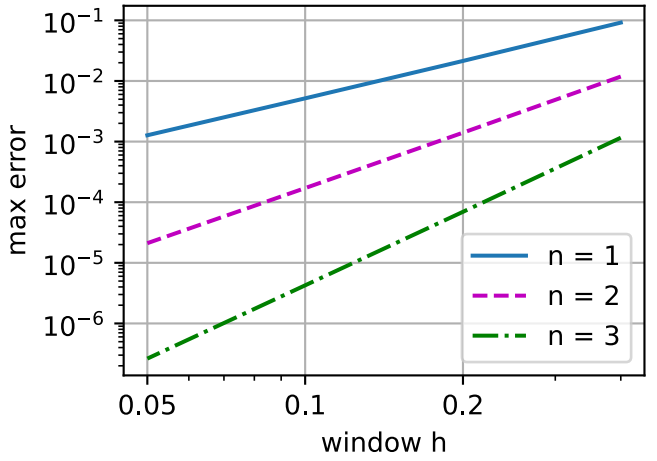
$$P_4(x) = \log 2 + \frac{x}{2} + \frac{x^2}{8} - \frac{x^4}{192}.$$

At $x = 0.5$, this evaluates to 0.974072 against the true value $s(0.5) = 0.974077$, an error of only 5×10^{-6}. In fact, the error is even smaller than an x^5 estimate would suggest, because $s(x) - \frac{x}{2} = \log\left(2\cosh\frac{x}{2}\right)$ is an even function, so all higher odd terms vanish and the next nonzero term is sixth order. That quartic term also explains why our earlier script printed 0.1249 instead of 0.1250 at $\epsilon = 0.1$: we were seeing $\frac{1}{8} - \frac{\epsilon^2}{192} = 0.12495$.

We can check the h^{n+1} scaling directly in code. If the worst-case error of P_n on the window $|x| \le h$ behaves like h^{n+1}, then halving h should divide the maximum error by 2^{n+1}: by 4 for the tangent line, 8 for the parabola, and 16 for the cubic. Let us verify this prediction on e^x at $x_0 = 0$.

```
# Max error of P_n for e^x on |x| <= h. Lagrange predicts error ~ h^(n+1),
# so halving h should divide it by 2^(n+1)
def taylor_error(n, h, m=1000):
    x = h * np.arange(-m, m + 1) / m
    P = sum(x**k / math.factorial(k) for k in range(n + 1))
    return np.max(np.abs(np.exp(x) - P))

hs = [0.4, 0.2, 0.1, 0.05]
errs = [[taylor_error(n, h) for h in hs] for n in (1, 2, 3)]
for n, e in zip((1, 2, 3), errs):
    ratios = ' '.join(f'{a / b:.1f}' for a, b in zip(e, e[1:]))
    print(f'n = {n}: ratios between successive windows {ratios}, '
          f'predicted {2**(n + 1)}')
d2l.plot(hs, errs, 'window h', 'max error', xscale='log', yscale='log',
         legend=['n = 1', 'n = 2', 'n = 3'])
```

```
n = 1: ratios between successive windows 4.3 4.1 4.1, predicted 4
n = 2: ratios between successive windows 8.4 8.2 8.1, predicted 8
n = 3: ratios between successive windows 16.7 16.3 16.2, predicted 16
```



The measured ratios indeed sit near 4, 8, and 16, and on the log–log plot the three error curves appear as straight lines with slopes 2, 3, and 4, matching $n+1$. The ratios slightly exceed the theoretical values on the wider windows because the factor e^{ξ} in the remainder is larger there.

The Lagrange remainder also answers when the infinite Taylor series reproduces the function itself. For e^x, every derivative equals e^x, so Equation 34.5 around $x_0 = 0$ gives

$$\left|e^x - P_n(x)\right| = \frac{e^{\xi}\,|x|^{n+1}}{(n+1)!} \le \frac{e^{|x|}\,|x|^{n+1}}{(n+1)!}.$$

For any fixed x, the right-hand side vanishes as $n \to \infty$ because factorials eventually outgrow powers. Hence

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \cdots$$

on all of $\mathbb{R}$. Be careful, though: having infinitely many derivatives does not guarantee that a function equals its Taylor series. For $f(x) = e^{-1/x^2}$ (extended by $f(0) = 0$), every single derivative vanishes at 0, so its Taylor series at the origin is identically zero and agrees with f only at $x = 0$ (Figure 34.7). Luckily, nothing we do later depends on infinite series: we work exclusively with finite Taylor polynomials P_n, whose error Equation 34.5 bounds directly.

34.1.5.4 The Descent Lemma

Does a bound on curvature guarantee that gradient descent lowers any smooth function, just as it did for x^2? The Lagrange remainder with $n = 1$ answers yes. Among all results in this section, this is the one optimization theory uses most.

Proposition (descent lemma). *Let f be twice differentiable on an open interval I containing x and $x - \eta f'(x)$, with $|f''| \le L$ on I. Then*

$$f\big(x - \eta\, f'(x)\big) \;\le\; f(x) - \eta\Big(1 - \frac{L\eta}{2}\Big) f'(x)^2. \tag{34.6}$$

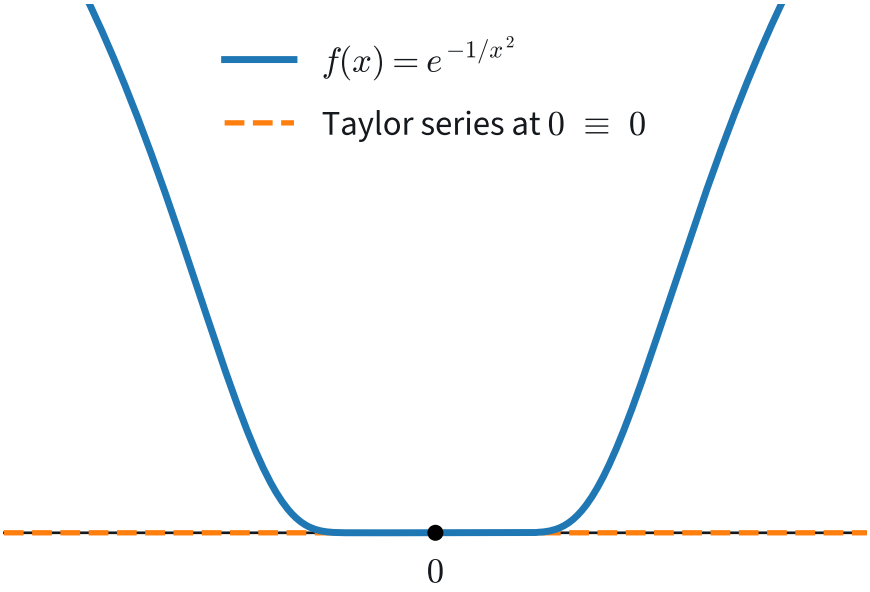


Figure 34.7: The smooth function $f(x) = e^{-1/x^2}$ has every derivative equal to zero at the origin, so its Taylor series there is identically zero. The series converges everywhere yet agrees with f only at $x = 0$: smooth does not imply analytic.

In particular, f strictly decreases whenever $0 < \eta < 2/L$ and $f'(x) \neq 0$, and the guaranteed decrease is maximized at $\eta = 1/L$, where it equals $f'(x)^2/(2L)$.

Proof. For any step u, the Lagrange remainder with $n = 1$ gives $f(x+u) = f(x)+f'(x)u+\frac{1}{2}f''(\xi)u^2 \leq f(x) + f'(x)u + \frac{L}{2}u^2$. Setting $u = -\eta f'(x)$ turns the right-hand side into $f(x) - \eta(1 - L\eta/2)f'(x)^2$. The factor $1 - L\eta/2$ is positive whenever $\eta < 2/L$, and the quadratic $\eta(1 - L\eta/2)$ attains its maximum at $\eta = 1/L$. ■

Geometrically, the lemma replaces the unknown local curvature $f''(\xi)$ with its worst-case bound L. The resulting quadratic $f(x)+f'(x)u+\frac{L}{2}u^2$ acts as an upper-bounding parabola that touches f at x and stays above it throughout I. The step $\eta = 1/L$ simply jumps to the exact minimizer of this ceiling parabola (Figure 34.8). This bound is the workhorse of smooth optimization (Nesterov 2018). In fact, it does not even require a second derivative: it holds whenever f' is *L-Lipschitz*, meaning that slopes at any two points differ by at most L times their distance, $|f'(u) - f'(v)| \leq L|u - v|$. That is the form used in multivariable optimization (Section 35.1), and exercise 1 proves it in one variable.

Let us check the lemma on two familiar functions. For x^2, we have $L = 2$, so $2/L = 1$ recovers the exact stability threshold we found from the closed form. For the logistic loss, $\ell''(w) = \sigma(-w)(1 - \sigma(-w)) \leq \frac{1}{4}$ with equality at $w = 0$, so $L = \frac{1}{4}$ and every step size $\eta < 8$ is guaranteed to decrease ℓ from any starting weight. Starting from $w = 0$ with $\eta = 1$ moves to $w_1 = 0.5$: the bound guarantees a loss at most $0.693147-(1-\frac{1}{8})\cdot\frac{1}{4} = 0.4744$, and the actual value is $\ell(0.5) = 0.4741$. The bound is nearly tight here because the curvature stays close to its maximum L over that short step. With the larger step $\eta = 4 = 1/L$, the bound promises 0.1931 while the actual loss drops all the way to $\ell(2) = 0.1269$: away from

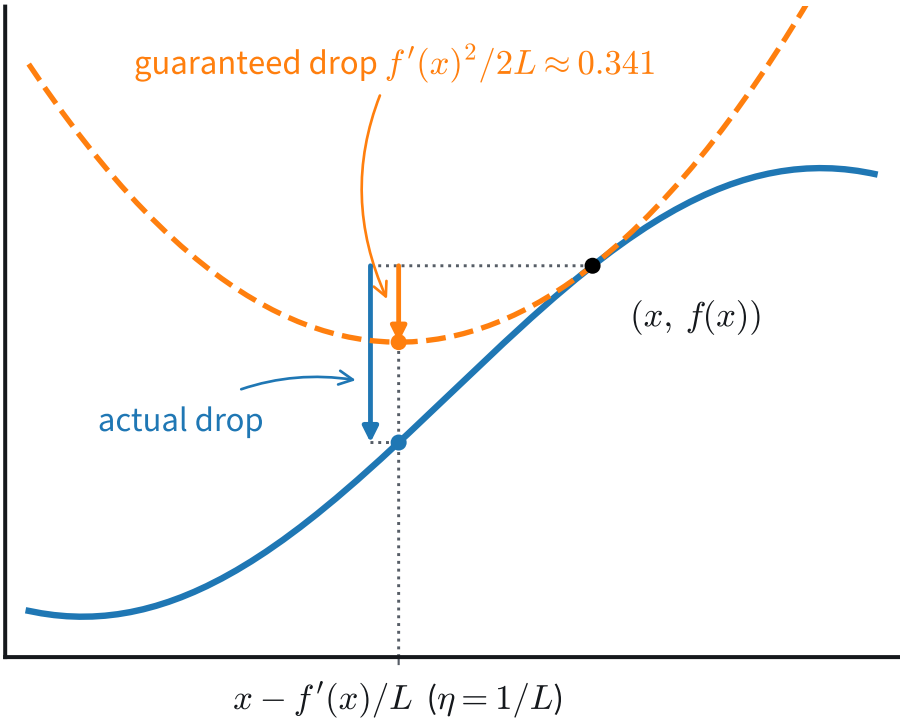


Figure 34.8: For $|f''| \leq L$, $f(x) + f'(x)u + \frac{L}{2}u^2$ is a quadratic upper bound in the step u that equals f at the base point. The gradient step with $\eta = 1/L$ minimizes this bound and guarantees a decrease of at least $\frac{1}{2L}f'(x)^2$. Here $f(x) = \sin(2x)$ with $L = 4$, and the actual decrease exceeds the worst-case guarantee.

the origin, the curvature flattens out, so the curve descends even further than the worst-case ceiling predicts.

34.1.5.5 Newton's Method

The descent lemma builds its quadratic model using the global worst-case curvature L. If instead we use the actual local curvature $f''(x_t)$ at the current iterate, we obtain a step that minimizes any quadratic function in a single leap. The local model at x_t is the second-order Taylor polynomial P_2,

$$q(x) = f(x_t) + f'(x_t)\,(x - x_t) + \tfrac{1}{2} f''(x_t)\,(x - x_t)^2.$$

Whenever $f''(x_t) > 0$, this parabola opens upward and has a unique minimizer. Setting $q'(x) = f'(x_t) + f''(x_t)(x - x_t) = 0$ and solving for x yields the update of **Newton's method**:

$$x_{t+1} = x_t - \frac{f'(x_t)}{f''(x_t)}. \tag{34.7}$$

This is a gradient step with an adaptive local step size $1/f''(x_t)$, and on any upward-opening parabola it lands on the exact minimizer in one step. Figure 34.9 illustrates one step on $f(x) = x^4/4 - x$, whose minimizer is $x^\star = 1$. At $x_t = 2$, we have $f(2) = 2$, $f'(2) = 7$, and $f''(2) = 12$, so the parabola's minimizer lies at $x_{t+1} = 2 - 7/12 = 1.4167$.

Let us see how quickly Newton's method converges on $x^4/4 - x$ from $x_0 = 2$, and also test what happens on a second function, $x \arctan x - \frac{1}{2}\log(1 + x^2)$, starting from $x_0 = 1.4$ and $x_0 = 1.3$. We will analyze that second example right after inspecting the numbers.

```python
# Newton on f(x) = x^4/4 - x: solve f'(x) = x^3 - 1 = 0, root x* = 1
x = 2.0
for t in range(6):
    print(f't = {t}: x = {x:.12f}, error = {abs(x - 1):.1e}')
    x = x - (x**3 - 1) / (3 * x**2)  # x - f'(x) / f''(x)

# Newton on the convex f(x) = x arctan(x) - log(1 + x^2)/2,
# with f' = arctan(x) and f'' = 1/(1 + x^2)
for x0 in [1.4, 1.3]:
    x, xs = x0, []
    for t in range(6):
        x = x - math.atan(x) * (1 + x**2)
        xs.append(x)
    print(f'from x0 = {x0}:', ', '.join(f'{v:.2f}' for v in xs))
```

```
t = 0: x = 2.000000000000, error = 1.0e+00
t = 1: x = 1.416666666667, error = 4.2e-01
t = 2: x = 1.110534409842, error = 1.1e-01
t = 3: x = 1.010636768405, error = 1.1e-02
t = 4: x = 1.000111557304, error = 1.1e-04
t = 5: x = 1.000000012443, error = 1.2e-08
from x0 = 1.4: -1.41, 1.45, -1.55, 1.85, -2.89, 8.71
from x0 = 1.3: -1.16, 0.86, -0.37, 0.03, -0.00, 0.00
```

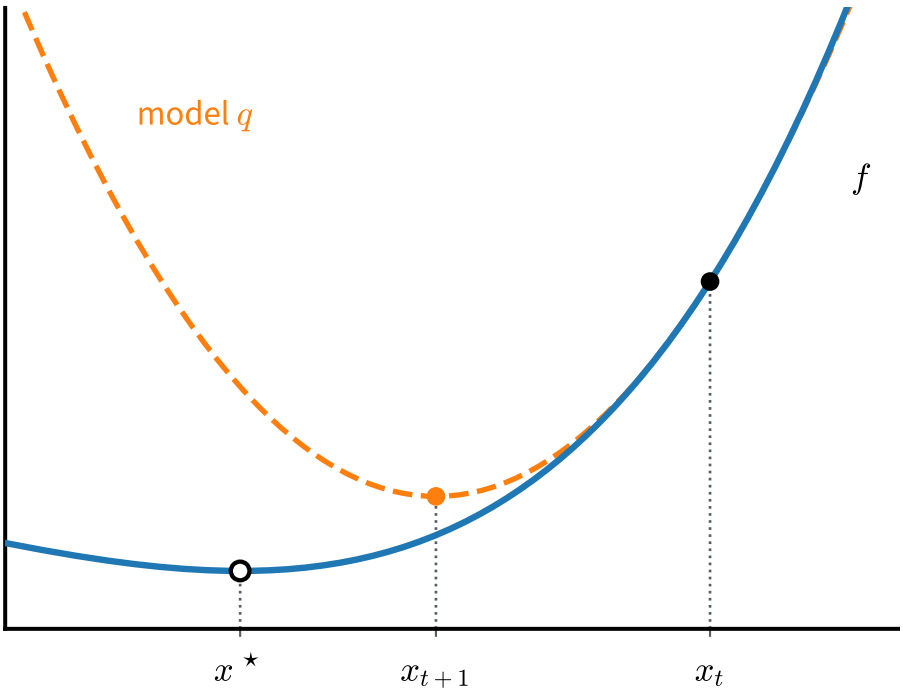


Figure 34.9: One Newton step on $f(x) = x^4/4 - x$ from $x_t = 2$. The parabola q matches the value, slope, and curvature of f at x_t; its minimizer is the next iterate $x_{t+1} = x_t - f'(x_t)/f''(x_t) = 2 - 7/12$, closer to the minimizer $x^\star = 1$.

On $x^4/4 - x$, the errors plummet through 1.0, 0.42, 0.11, 1.1×10^{-2}, 1.1×10^{-4}, and 1.2×10^{-8}: once close to the minimum, each step squares the previous error and roughly doubles the number of correct digits. By contrast, gradient descent on x^2 shrank the error by a constant factor $|1 - 2\eta|$ per step, gaining only a fixed number of digits at each iteration.

The Lagrange remainder explains why the error squares. Let $x^\star$ be a stationary point and let $e_t = x_t - x^\star$ denote the error at step t. Applying Equation 34.5 with $n = 1$ to f' around the base point x_t and evaluating at $x^\star$ gives $0 = f'(x_t) - f''(x_t)\, e_t + \frac{1}{2} f'''(\xi_t)\, e_t^2$. Dividing through by $f''(x_t) \neq 0$ and substituting Newton's update Equation 34.7 leaves

$$e_{t+1} = \frac{f'''(\xi_t)}{2 f''(x_t)}\, e_t^2 \tag{34.8}$$

for some ξ_t between x_t and $x^\star$, provided f is three times differentiable. For $x^4/4 - x$, the prefactor converges to $f'''(1)/(2f''(1)) = 6/(2 \cdot 3) = 1$, and the printed errors indeed yield $e_{t+1}/e_t^2 \approx 0.42$, 0.64, 0.87, and 0.99. The general multivariable theorem, giving quadratic convergence from any starting point close enough to a minimizer with positive curvature, appears in Section 35.2.

That qualification "close enough" is essential, even for a strictly convex function. Consider $f(x) = x \arctan x - \frac{1}{2}\log(1+x^2)$, which has $f'(x) = \arctan x$ and $f''(x) = 1/(1+x^2) > 0$, so it is strictly convex with a unique global minimum at 0. Starting from $x_0 = 1.4$, the second loop above prints $-1.41, 1.45, -1.55, 1.85, -2.89, 8.71$: Newton's method diverges wildly. Starting slightly closer at $x_0 = 1.3$, it converges smoothly (-1.16, 0.86, -0.37, 0.03, ...), with the threshold between convergence and divergence sitting near $x_0 \approx 1.39$.

The local parabola explains what went wrong. Far from the origin, the curvature $1/(1+x^2)$ becomes tiny, so the approximating parabola q is nearly flat and places its minimizer far past $x^\star$. The standard fix is to scale back the step by a factor chosen to ensure f actually decreases, a technique called *damping*. We develop damped Newton steps, and examine the cost of computing them in high dimensions, in Section 35.2.1.

34.1.6 Nondifferentiable Points

Every ReLU network is full of corners: $\mathrm{ReLU}(x) = \max(0, x)$ has no derivative at 0. Yet backpropagation still returns a number there. Which number does it pick, and in what sense is that number a valid slope?

34.1.6.1 One-Sided Derivatives

The difference quotient Equation 34.1 exposes the problem immediately. For $f(x) = |x|$ at $x = 0$,

$$\frac{|0 + \epsilon| - |0|}{\epsilon} = \frac{|\epsilon|}{\epsilon} = \begin{cases} +1 & \epsilon > 0, \\ -1 & \epsilon < 0. \end{cases}$$

The quotient is constant on either side of 0, but the two constants disagree. Approaching from the right, the graph rises with slope $+1$; approaching from the left, it falls with slope -1. These limits are the **one-sided derivatives**, $f'_+(0) = +1$ and $f'_-(0) = -1$, and an ordinary derivative exists only when the two match. The same calculation for ReLU at 0

gives $f'_-(0) = 0$ on the flat left side and $f'_+(0) = 1$ on the right. No tangent slope exists at these corners, yet an automatic differentiation engine must still return a number. Let us check what automatic differentiation actually returns.

```
# What autograd returns at the corner, where no derivative exists
for name, fn in [('d|x|/dx', torch.abs), ("ReLU'", torch.relu)]:
    x = torch.tensor(0.0, requires_grad=True)
    fn(x).backward()
    print(f"{name} at 0: autograd returns {x.grad.item():.1f}")
```

```
d|x|/dx at 0: autograd returns 0.0
ReLU' at 0: autograd returns 0.0
```

For $\mathrm{ReLU}'(0)$, autograd returns 0, the left-hand slope. For $\frac{d}{dx}|x|$ at 0, it returns a fixed value chosen by convention.

Here it returns 0, following the convention $\mathrm{sign}(0) = 0$.

34.1.6.2 Subgradients and Optimality

Although a corner has no single tangent line, it has an entire family of lines that touch the graph at the corner without ever rising above it. For a *convex* function, one whose chords all lie on or above its graph (Section 35.3), these supporting lines step into the role of the tangent. A number g is called a **subgradient** of a convex function f at x if

$$f(y) \geq f(x) + g\,(y - x) \quad \text{for all } y, \tag{34.9}$$

meaning that the line through $(x, f(x))$ with slope g lies everywhere on or below the graph of f. The set of all subgradients at x is the **subdifferential** $\partial f(x)$ (Rockafellar 1970). At any point where f is differentiable, $\partial f(x)$ contains only the single number $f'(x)$. At a corner, it expands into the closed interval between the one-sided derivatives, corresponding to the fan of supporting lines in Figure 34.10:

$$\partial|x|(0) = [-1, 1], \qquad \partial\,\mathrm{ReLU}(0) = [0, 1].$$

Every value returned by autograd above lies inside the corresponding interval.

Our running function shows intuitively where this interval comes from. The sharpened softplus $\mathrm{softplus}_\beta(x) = \frac{1}{\beta}s(\beta x)$ converges to $\mathrm{ReLU}(x)$ as $\beta \to \infty$, and its derivative is $\sigma(\beta x)$ (Figure 34.11). At $x = \pm 0.1$, the slopes are 0.40 and 0.60 when $\beta = 4$, and sharpen to 0.17 and 0.83 when $\beta = 16$. The subdifferential $[0, 1]$ is precisely the set of slopes that these smooth approximations approach as $\beta \to \infty$ and $x \to 0$ together.

Subgradients also yield a clean optimality test. Setting $g = 0$ in Equation 34.9 gives $f(y) \geq f(x)$ for all y, so a point x is a global minimizer of a convex function f if and only if

$$0 \in \partial f(x). \tag{34.10}$$

Both directions of the equivalence are immediate from that one line. For $|x|$, the membership $0 \in [-1, 1]$ certifies the origin as the global minimum even though no derivative exists

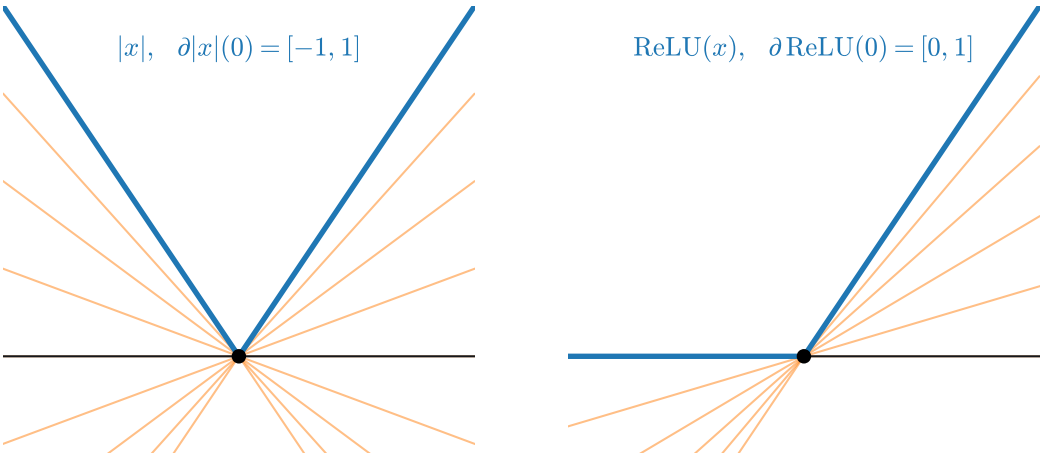


Figure 34.10: Two corners and their subdifferentials: the fan of supporting lines through the origin of $|x|$ (left) and of ReLU (right), with slopes running over $\partial|x|(0)$ and $\partial\,\mathrm{ReLU}(0)$ respectively. Off each corner there is a single tangent and the subdifferential collapses to the ordinary derivative.

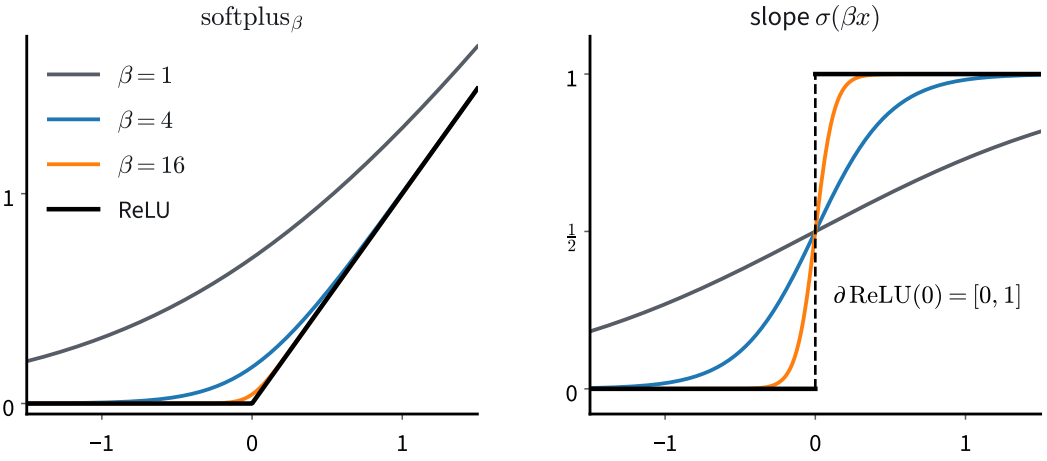


Figure 34.11: Smoothing the corner. Left: $\mathrm{softplus}_\beta(x) = \frac{1}{\beta}\log(1 + e^{\beta x})$ approaches ReLU as β grows. Right: its slope $\sigma(\beta x)$ steepens into the step from 0 to 1; near $x = 0$ the slopes of the smooth neighbors fill the interval $[0, 1]$, the subdifferential of ReLU at the corner.

there. Without convexity, by contrast, $f'(x) = 0$ is only necessary (Fermat's lemma): x^3 satisfies $f'(0) = 0$ yet has no minimum at all.

We extend subgradients to many variables in Section 35.3, but one caution belongs right here: unlike a true gradient step, a subgradient step is not guaranteed to descend. Right at the minimum of $|x|$, the number $g = \frac{1}{2}$ is a valid subgradient, yet stepping to $x = -\eta/2$ actually increases $|x|$. Subgradient methods therefore converge by gradually shrinking their step sizes rather than by guaranteeing progress on every step (Nesterov 2018).

34.1.6.3 Kinks in a Network

Each kink in a neural network originates in a simple convex building block such as $|x|$, ReLU, or the hinge loss. At the kink, autograd picks one fixed element from that block's subdifferential (for example, $\mathrm{ReLU}'(0) = 0$) and propagates it through the surrounding composition using the chain rule. Unfortunately, chaining valid subgradients of individual pieces can produce a number that is not a subgradient of the composite function at all. The simplest counterexample is the identity function written in a roundabout way:

$$\varphi(x) = \mathrm{ReLU}(x) - \mathrm{ReLU}(-x) = x \quad \text{for all } x.$$

Because $\varphi(x) = x$ everywhere, its only true slope at 0 is 1. Yet applying the chain rule mechanically gives $\varphi'(x) = \mathrm{ReLU}'(x) + \mathrm{ReLU}'(-x)$, which equals 1 everywhere except at $x = 0$, where the convention $\mathrm{ReLU}'(0) = 0$ yields $0 + 0 = 0$. Let us verify that automatic differentiation indeed does this.

```
# phi(x) = relu(x) - relu(-x) is the identity, so the true slope at 0 is 1
x = torch.tensor(0.0, requires_grad=True)
phi = torch.relu(x) - torch.relu(-x)
phi.backward()
print(f"autograd: phi'(0) = {x.grad.item():.1f}  (true slope: 1.0)")
```

```
autograd: phi'(0) = 0.0  (true slope: 1.0)
```

As predicted, autograd reports slope 0 for the identity function.

Triggering such an error requires a pre-activation to land on 0 exactly. In exact real arithmetic, when data and initial weights are drawn from continuous distributions, this occurs with probability zero at any fixed step. In finite floating-point arithmetic, however, exact zeros do occur. Bertoin et al. (2021) measured how often on a small fully connected network. We quote the figures of the authors' 2023 erratum (the revised arXiv version of the paper), which repairs a bug in the published experiments and reverses some of their conclusions. Backpropagation with $\mathrm{ReLU}'(0) = 0$ and with $\mathrm{ReLU}'(0) = 1$ produced different gradients on at least one minibatch for every initialization in 16-bit precision, for a little under half of all initializations in 32-bit precision, and for none in 64-bit precision.

Whether those discrepancies affect training is a separate question, and the answer is reassuring: practically not at all. In 32-bit arithmetic, only a few minibatches in a million are affected, and even on those the two gradients differ by a relative amount of only about 10^{-3}. In the corrected experiments, any choice of $\mathrm{ReLU}'(0)$ in $[0, 1]$ yields virtually identical training loss and test accuracy; training degrades only for extreme choices such as $\mathrm{ReLU}'(0) = 10^4$, and batch normalization and Adam tame even those. The convention is

therefore a real source of minor numerical differences between runs, yet a harmless one. Exercise 5 examines this behavior on a small scale.

Everything in this section concerned a single scalar weight. A neural network has millions of weights moving together: which joint direction lowers the loss, and what curvature means in many dimensions, are the subjects of Section 34.2.

34.1.7 Summary

The derivative $f'(x)$ is the slope of the linear approximation $f(x+\epsilon) = f(x)+\epsilon f'(x)+o(\epsilon)$. Proved from this approximation, the sum, product, and chain rules let us differentiate any composition built from a short table of elementary functions, such as $s' = \sigma$ for the softplus. To go from slopes at single points to guarantees over intervals, we rely on the Mean Value Theorem and its higher-order extension, the Lagrange remainder, which bounds the error of the degree-n Taylor polynomial by a multiple of $|x-x_0|^{n+1}$ whenever $f^{(n+1)}$ is bounded.

Applied at order $n = 1$, the Lagrange remainder gives the descent lemma: if $|f''| \le L$ on the step interval and $f'(x) \neq 0$, every gradient step with $0 < \eta < 2/L$ strictly lowers f. Replacing the global bound L with the local curvature $f''(x_t)$ gives Newton's step $-f'/f''$, which squares the error near a minimizer with $f'' > 0$ but can diverge far from it, even on a strictly convex function. Finally, at nondifferentiable corners of a convex function, the subdifferential $\partial f(x)$ replaces the derivative and $0 \in \partial f(x)$ characterizes global minima; autograd selects one element per kink, and although chaining those choices can fail to produce a true subgradient in floating point, the effect on training is negligible.

34.1.8 Exercises

1. **Smoothness of the logistic loss.** The descent lemma Equation 34.6 requires a bound L on the curvature.
   1. Use the Mean Value Theorem Equation 34.3 to show that if $|f''| \le L$ on an interval, then f' is L-Lipschitz there. Conversely, show from the definition Equation 34.1 that a twice differentiable f with an L-Lipschitz f' satisfies $|f''| \le L$.
   2. Logistic regression with one weight w, inputs $a_i \in \mathbb{R}$, and labels $y_i \in \{-1, 1\}$ minimizes

$$\ell(w) = \frac{1}{n}\sum_{i=1}^{n} \log(1 + e^{-y_i a_i w}). \tag{34.11}$$

      Express ℓ'' in terms of σ. Find the smallest L for which ℓ' is L-Lipschitz on all of $\mathbb{R}$, and the step sizes for which Equation 34.6 guarantees a decrease from every starting point.
   3. Prove the descent lemma when f is only differentiable and f' is L-Lipschitz on the interval of the step. For $u = -\eta f'(x)$ let $\psi(t) = f(x+tu)-f(x)-tf'(x)u-\frac{L}{2}t^2u^2$. Show that $\psi(0) = 0$ and $\psi'(t) = u\,(f'(x+tu) - f'(x)) - Ltu^2 \le 0$ for $t \in [0, 1]$, and conclude $\psi(1) \le 0$ from the Mean Value Theorem. This is the form of the lemma that optimization uses for functions of many variables.
2. **Entropy extremes.** The binary entropy of a coin that shows heads with probability p is

$$H(p) = -p \log p - (1-p) \log(1-p), \qquad 0 < p < 1. \tag{34.12}$$

1. Find the minimum of $f(x) = x \log x$ on $x \geq 0$, where $f(0) = 0$ is the limiting value. Why does the minimum not lie at the boundary $x = 0$?
2. Show that $H'(p) = -\log \frac{p}{1-p}$, the negative logit of p. Locate the stationary point of H and classify it with the second-derivative test.
3. Find the limit of $H'(p)$ as $p \to 0$. Then parameterize the coin by its logit z, so that $p = \sigma(z)$, and compute $\frac{d}{dz} H(\sigma(z))$. In which of the two parameterizations is the slope of H Lipschitz on the whole domain, so that Equation 34.6 certifies one step size of gradient descent on $-H$ for every starting point?

3. **Newton at degenerate minima.** For $p > 1$, the function $f_p(x) = |x|^p / p$ is convex with its unique minimum at $x = 0$.

   1. Show that f_p is differentiable everywhere, including at 0, with $f_p'(x) = \operatorname{sign}(x)\,|x|^{p-1}$, and compute $f_p''(x)$ for $x \neq 0$. For which p does $f_p''(0)$ exist, and what is its value?
   2. Show that Newton's method Equation 34.7 maps every $x_t \neq 0$ to $x_{t+1} = \frac{p-2}{p-1}\, x_t$. For which p does the iteration converge? How fast does it converge for $p = 4$?
   3. Which hypothesis of the local convergence statement in Section 34.1.5.5 fails for $p = 4/3$, and which fails for $p = 4$?
   4. For $p = 4/3$, the damped update $x_{t+1} = x_t - \alpha f_p'(x_t)/f_p''(x_t)$ shortens the Newton step by a factor $\alpha > 0$. For which α does it converge from every x_0, and at what rate?

   Adapted from MIT 18.01SC, Session 33 problems, "Cube Root of x".

4. code **Minimizers at kinks.** A support vector machine with one weight x, weight decay $\lambda > 0$, and a single training example whose label times input equals 1 minimizes

$$F(x) = \max(0, 1 - x) + \frac{\lambda}{2} x^2. \tag{34.13}$$

   1. Compute the subdifferential of the hinge loss $\max(0, 1-x)$ at every x from Equation 34.9.
   2. Take as given that adding a differentiable convex function q to a convex function h adds q' to every subgradient, $\partial(q+h)(x) = q'(x) + \partial h(x)$. Use the optimality condition Equation 34.10 to find the minimizer of F for every $\lambda > 0$. For which λ does it sit at the kink?
   3. Take $\lambda = 1/2$ and run the subgradient method $x_{t+1} = x_t - \eta_t g_t$ from $x_0 = 0$ for 10^4 steps, where g_t is the value that autograd returns for $F'(x_t)$ when the hinge is written as $\mathrm{ReLU}(1 - x)$. Compare the constant steps $\eta_t = 0.1$ and $\eta_t = 0.01$ with the decaying steps $\eta_t = 0.5/\sqrt{t+1}$. How does the final distance to the minimizer depend on the schedule?
   4. The kink $\{x = 1\}$ is a single point, which a draw from a density hits with probability zero. What do (b) and (c) imply for the argument in Section 34.1.6.3 that training rarely meets a kink?

5. code **Chained subgradients.** Autograd returns $\mathrm{ReLU}'(0) = 0$ and chains this value through every expression, as in Section 34.1.6.3.

1. Let f_1 and f_2 be convex, $g_1 \in \partial f_1(x)$, $g_2 \in \partial f_2(x)$, and $c_1, c_2 \geq 0$. Show from Equation 34.9 that $c_1 g_1 + c_2 g_2 \in \partial(c_1 f_1 + c_2 f_2)(x)$. Why does this argument not cover $\mathrm{ReLU}(x) - \mathrm{ReLU}(-x)$?
2. Show that for every real c,

$$k_c(x) = \mathrm{ReLU}(x) + \mathrm{ReLU}(-x) + c\,(x - \mathrm{ReLU}(x) + \mathrm{ReLU}(-x)) = |x|. \tag{34.14}$$

   Predict the value that autograd returns for $k_c'(0)$, and check the prediction for $c = 0$, 1, and 5. For which c is the returned value a subgradient of $|x|$ at 0?
3. Run gradient descent with the derivative from autograd on $\varphi(x) = \mathrm{ReLU}(x) - \mathrm{ReLU}(-x)$ for 10 steps from $x_0 = 0.5$, once with $\eta = 0.125$ and once with $\eta = 0.1$. What happens in each run? Why do the two step sizes behave differently? What does the difference imply for the probability-zero argument of Section 34.1.6.3 in floating-point arithmetic?

6. **Quartic descent.** ● The minimum of $f(x) = x^4/4$ is flat, since $f''(0) = 0$. Gradient descent Equation 34.4 on f reads $x_{t+1} = x_t - \eta x_t^3$.
   1. Using problem 1a, find the smallest L for which f' is L-Lipschitz on $[-R, R]$. Which starting points x_0 does the descent lemma Equation 34.6 cover for a given η? Why does no fixed η cover all of them?
   2. Show that the iterates converge to 0 if $\eta x_0^2 < 2$ and diverge if $\eta x_0^2 > 2$. What happens at $\eta x_0^2 = 2$? Compare this threshold with the starting points covered in (a).
   3. Show that $1/x_{t+1}^2 = 1/x_t^2 + 2\eta + O(x_t^2)$, and conclude that $x_t^2 \approx 1/(x_0^{-2} + 2\eta t)$. Check the prediction for $\eta = 0.1$ and $x_0 = 1$ at $t = 10$, 100, and 1000. How does this rate compare with gradient descent on x^2, and with Newton's method on $x^4/4$ in problem 3b?

7. code **Central differences.** ● The forward difference $D_+ f(x) = \frac{f(x+\epsilon) - f(x)}{\epsilon}$ and the central difference $D_0 f(x) = \frac{f(x+\epsilon) - f(x-\epsilon)}{2\epsilon}$ both approximate $f'(x)$.
   1. Use the Lagrange remainder Equation 34.5 to show $D_+ f(x) - f'(x) = \frac{1}{2} f''(\xi)\,\epsilon$ and $D_0 f(x) - f'(x) = \frac{1}{6} f'''(\xi')\,\epsilon^2$ for some ξ, ξ' near x, where f is three times continuously differentiable (the continuity of f''' merges two remainder points into one). Expand $f(x \pm \epsilon)$ to the needed order and subtract.
   2. For $f(x) = x^2 + 1701(x-4)^3$ at $x = 4$, show that the two errors are $\epsilon + 1701\epsilon^2$ and $1701\epsilon^2$. At $\epsilon = 0.1$ they are 17.11 and 17.01, nearly equal. The central difference raises the order of the error, which helps only once ϵ is small. At which ϵ is the central error a hundred times smaller than the forward one?

   In floating point, rounding puts a floor under ϵ. The gradient-check exercise of Section 34.3 balances it against the truncation error of (a).

8. code **Gradient descent past $2/L$.** ●● On $f(x) = x^2$, every step size $\eta > 2/L$ diverges. On other functions, gradient descent past this threshold can settle into an oscillation instead (Chen and Bruna 2023), a one-dimensional version of the edge of stability (Cohen et al. 2021). Let $x^\star$ be a local minimizer of a four times continuously

differentiable f with $f''(x^\star) > 0$, and write $T(x) = x - \eta f'(x)$ for one step of Equation 34.4.

1. Use Equation 34.3 to show that iterates that start close enough to $x^\star$ converge to it if $\eta < 2/f''(x^\star)$. Show that if $\eta > 2/f''(x^\star)$, every iterate in some neighborhood of $x^\star$, other than $x^\star$ itself, is followed by one farther from $x^\star$.
2. A period-2 orbit is a pair $a \neq b$ with $T(a) = b$ and $T(b) = a$. Show that for every period-2 orbit there is a point ξ strictly between a and b with $f''(\xi) = 2/\eta$.
3. Let $\eta = (1+\delta)\,2/f''(x^\star)$ with a small $\delta > 0$. Expand $T(T(x))$ around $x^\star$ to third order, and show that a period-2 orbit near $x^\star$ exists and attracts nearby iterates if

$$3f'''(x^\star)^2 - f''(x^\star)\,f''''(x^\star) > 0. \tag{34.15}$$

   Derive the half-width $|a-b|/2$ of the orbit to leading order in δ.
4. Run gradient descent on the double well $f(x) = \frac{1}{4}(x^2-1)^2$ from $x_0 = 0.5$ for 5000 steps for each step size $\eta \in [0.9, 1.5]$, and plot the last 100 iterates against η. Compare the half-width of the orbit near $\eta = 1$ with (c). Compare the range of η with a stable period-2 orbit with the range $1 < \eta < \sqrt{4.5} - 1$ in which Chen and Bruna (2023) prove convergence to the orbit from almost every $0 < x_0 < 1$.
5. Predict from (c) what gradient descent with $\eta = 2.02$ does on $\log\cosh x$ and on $x^2/2 + x^4/4$, both of which have $x^\star = 0$ and $f''(x^\star) = 1$. Confirm the predictions by running both from $x_0 = 0.01$ for 500 steps.

34.2 Multivariable Calculus

In the previous section, we studied a loss as a function of a single weight while holding every other weight fixed. Real training never works that way. Every step of gradient descent nudges all the weights at once, so we need to know how the loss responds to a joint move, and which joint move drives it down fastest. Having a million separate slopes, one per weight, does not by itself answer either question.

Yet under one modest hypothesis, those slopes answer both. When the loss is differentiable, the vector of slopes, called the *gradient*, predicts the effect of every small joint move at once to first order and points in the direction of steepest increase. To go further and tell a local minimum from a saddle, we need a symmetric table of second derivatives, the *Hessian*, which measures how the surface bends. This section develops what these two objects tell us about a function of many variables. How a program actually computes a gradient for a large network is a separate problem, which Section 34.3 takes up next.

Section 1.4 already stated the differentiation rules and put the gradient to work. Here we supply the foundations, much as Section 34.1 did for one variable: we prove when the linear approximation holds and how second derivatives classify critical points. Two results carry the entire section: the multivariable linear model and the second-derivative test. The short subsection on tangent planes and the closing material on Lagrange multipliers can safely be skipped on a first reading. Throughout, we will test each idea on a running example: the two-variable log-sum-exp function $f(x, y) = \log(e^x + e^y)$ evaluated at $(0, \log 2)$. Its gradient will turn out to be the familiar softmax.

34.2.1 Partial Derivatives and the Gradient

34.2.1.1 Partial Derivatives

If we freeze every weight except w_1, the loss becomes a function of a single variable. Its ordinary derivative is the *partial derivative* $\partial \ell / \partial w_1$ from Section 1.4: geometrically, it is the slope of the one-dimensional slice of the graph of ℓ obtained by varying only w_1. Just as in single-variable calculus, this slope predicts the effect of a small change ϵ_1 in w_1 alone,

$$\ell(w_1 + \epsilon_1, w_2, \dots, w_N) \approx \ell(w_1, w_2, \dots, w_N) + \epsilon_1 \frac{\partial \ell}{\partial w_1}(w_1, w_2, \dots, w_N). \tag{34.16}$$

Let us compute the partial derivatives of our running example $f(x, y) = \log(e^x + e^y)$. Holding y fixed and differentiating with respect to x gives $\partial f / \partial x = e^x / (e^x + e^y)$, and by symmetry $\partial f / \partial y = e^y / (e^x + e^y)$. At the point $(0, \log 2)$, we have $e^x = 1$ and $e^y = 2$, so $f(0, \log 2) = \log 3 \approx 1.0986$, $\partial f / \partial x = \frac{1}{3}$, and $\partial f / \partial y = \frac{2}{3}$. In other words, nudging y alone raises f twice as fast as nudging x alone.

34.2.1.2 Differentiability and the Gradient

Partial derivatives tell us the slope along each coordinate axis, one at a time. They do not automatically tell us what happens when all coordinates move together, for instance when we step from $(0, \log 2)$ by $\epsilon = (0.01, -0.03)$. It is tempting to guess that the two coordinate-wise changes simply add, giving $\frac{1}{3} \cdot 0.01 + \frac{2}{3} \cdot (-0.03)$. That guess is actually a strong claim about how f behaves near the point, and it requires a hypothesis.

The hypothesis is that near $\mathbf{w}$, the function ℓ looks linear up to an error that vanishes faster than the step size. Formally, we say that ℓ is *differentiable* at $\mathbf{w}$ if there exists a linear map Λ such that

$$\ell(\mathbf{w} + \epsilon) = \ell(\mathbf{w}) + \Lambda(\epsilon) + o(\|\epsilon\|), \tag{34.17}$$

where $o(\|\epsilon\|)$ denotes any remainder that, when divided by $\|\epsilon\|$, tends to zero as $\epsilon \to \mathbf{0}$.

Whenever such a linear map Λ exists, the partial derivatives determine it completely. Setting $\epsilon = h\mathbf{e}_i$ in Equation 34.17, dividing both sides by h, and letting $h \to 0$ forces $\Lambda(\mathbf{e}_i) = \partial \ell / \partial w_i$. Because Λ is linear, its action on any joint step ϵ is then fixed:

$$\Lambda(\epsilon) = \Lambda\Big(\sum_{i=1}^{N} \epsilon_i \mathbf{e}_i\Big) = \sum_{i=1}^{N} \epsilon_i \frac{\partial \ell}{\partial w_i} = \epsilon \cdot \nabla_{\mathbf{w}} \ell, \qquad \nabla_{\mathbf{w}} \ell = \left[\frac{\partial \ell}{\partial w_1}, \dots, \frac{\partial \ell}{\partial w_N}\right]^{\top}.$$

We call this column vector $\nabla_{\mathbf{w}} \ell$ of partial derivatives the *gradient* of ℓ. Substituting it back into Equation 34.17 gives the multivariable linear approximation

$$\ell(\mathbf{w} + \epsilon) = \ell(\mathbf{w}) + \epsilon \cdot \nabla_{\mathbf{w}} \ell(\mathbf{w}) + o(\|\epsilon\|). \tag{34.18}$$

Compare this with the one-variable approximation $f(x + \epsilon) \approx f(x) + \epsilon f'(x)$: the shape is identical, with the scalar slope replaced by a vector and ordinary multiplication replaced by a dot product. The gradient is the unique vector whose dot product with a step gives the

first-order change in ℓ. Of all the ways to think about a gradient, this is the one to remember. Throughout this book, we treat the gradient as a column vector.

Returning to our running example at $(0, \log 2)$, we have $\nabla f = [\frac{1}{3}, \frac{2}{3}]^\top$. Plugging $\epsilon = (0.01, -0.03)$ into Equation 34.18 predicts $f(0.01, \log 2 - 0.03) \approx \log 3 + \frac{1}{3}\cdot 0.01 - \frac{2}{3}\cdot 0.03 \approx 1.081946$. Notice that at *every* point (x, y), the two entries of ∇f are positive and sum to one: the gradient of log-sum-exp is the softmax $\mathbf{p} = \operatorname{softmax}(x, y)$, with entries $e^x/(e^x + e^y)$ and $e^y/(e^x + e^y)$.

It is worth pausing to see why differentiability asks for more than the mere existence of partial derivatives. Consider $f(x, y) = 2xy/(x^2 + y^2)$ with $f(0, 0) = 0$. Along both coordinate axes f is identically zero, so both partial derivatives at the origin are 0, leaving the constant function 0 as the only possible linear model. Yet along the diagonal $x = y \neq 0$, we have $f(x, x) = 1$ no matter how close we get to the origin, so no linear approximation can hold there. Here f is not even continuous at the origin, and Exercise 1 shows that even a continuous function can have all partial derivatives and still fail to be differentiable.

Fortunately, we rarely have to check Equation 34.17 directly. A convenient sufficient condition is that all partial derivatives exist in a neighborhood of $\mathbf{w}$ and are continuous at $\mathbf{w}$ (Rudin 1976). Since the partial derivatives of log-sum-exp are continuous everywhere, it is differentiable everywhere.

The definition of differentiability promises only that the approximation error is $o(\|\epsilon\|)$, meaning it shrinks faster than $\|\epsilon\|$. How much faster is it in practice? Before running the next cell, guess by what factor the error of the linear model shrinks when we cut the step $\epsilon = (0.01, -0.03)$ by a factor of ten. The experiment below uses plain NumPy to check.

```
import numpy as np
import torch
```

```
def lse(xy):  # f(x, y) = log(e^x + e^y), computed stably
    return np.logaddexp(xy[0], xy[1])

def grad_lse(xy):  # its gradient, the softmax of (x, y)
    return np.exp(xy - lse(xy))

xy0 = np.array([0.0, np.log(2)])
print('gradient:', grad_lse(xy0))
for scale in [1, 0.1, 0.01]:
    eps = scale * np.array([0.01, -0.03])
    linear = lse(xy0) + eps @ grad_lse(xy0)
    true = lse(xy0 + eps)
    print(f'|eps| {np.linalg.norm(eps):.1e}: linear {linear:.8f}, '
          f'true {true:.8f}, gap {true - linear:.3e}')
```

```
gradient: [0.33333333 0.66666667]
|eps| 3.2e-02: linear 1.08194562, true 1.08212418, gap 1.786e-04
|eps| 3.2e-03: linear 1.09694562, true 1.09694740, gap 1.779e-06
|eps| 3.2e-04: linear 1.09844562, true 1.09844564, gap 1.778e-08
```

At $\|\epsilon\| \approx 0.03$, the linear model misses the true value by 1.79×10^{-4} (only in the fourth decimal place), and each tenfold reduction in the step shrinks the gap by a factor of a hun-

dred. That means the error is second order in the step, which is considerably tighter than differentiability alone guarantees. Soon, the Hessian will let us predict this gap accurately (Section 34.2.3.2).

34.2.1.3 Directional Derivatives

So far we have used the gradient to predict small steps. It also gives the exact instantaneous slope along any direction, not just along the coordinate axes. Take any unit vector $\mathbf{u}$, substitute $\epsilon = h\mathbf{u}$ into Equation 34.18, divide by h, and let $h \to 0$. The $o(\|h\mathbf{u}\|)$ remainder vanishes, leaving

$$\lim_{h \to 0} \frac{\ell(\mathbf{w} + h\mathbf{u}) - \ell(\mathbf{w})}{h} = \mathbf{u} \cdot \nabla_{\mathbf{w}} \ell(\mathbf{w}). \tag{34.19}$$

The limit on the left is the *directional derivative* of ℓ along $\mathbf{u}$; the right side says it is simply the projection of the gradient onto $\mathbf{u}$. A single vector therefore encodes the slope in every possible direction at once.

Three directions for our running example at $(0, \log 2)$ illustrate the full range of behavior. Along the diagonal $\mathbf{u} = (1,1)/\sqrt{2}$, the slope is $(\frac{1}{3} + \frac{2}{3})/\sqrt{2} = 1/\sqrt{2} \approx 0.707$. In fact, this slope is the same at *every* point, because $f(x+t, y+t) = \log(e^t(e^x + e^y)) = f(x,y) + t$: moving a distance $\sqrt{2}\,t$ along the diagonal always raises f by t. Pointing directly against the gradient, along $\mathbf{u} = -(1,2)/\sqrt{5}$, gives the slope $-(\frac{1}{3} + \frac{4}{3})/\sqrt{5} = -\sqrt{5}/3 \approx -0.745$. Finally, along $\mathbf{u} = (2,-1)/\sqrt{5}$, the slope is $(\frac{2}{3} - \frac{2}{3})/\sqrt{5} = 0$, so to first order f does not change at all in that direction.

34.2.2 The Geometry of Gradients

34.2.2.1 Steepest Descent

Gradient descent (Section 2.1) nudges the parameters a short distance in the direction along which the loss drops fastest. By Equation 34.19, we are looking for the unit vector $\mathbf{v}$ whose projection $\mathbf{v} \cdot \nabla\ell$ is as negative as possible. Section 1.4 identified this direction from the cosine of the angle between $\mathbf{v}$ and the gradient; the Cauchy–Schwarz inequality gives an even cleaner proof and confirms that the direction is unique.

Proposition (steepest ascent and descent). *At any point where $\nabla\ell \neq \mathbf{0}$, among all unit vectors $\mathbf{v}$, the directional derivative $\mathbf{v} \cdot \nabla\ell$ is maximized by $\mathbf{v} = \nabla\ell/\|\nabla\ell\|$ and minimized by $\mathbf{v} = -\nabla\ell/\|\nabla\ell\|$. Thus $+\nabla\ell$ is the direction of steepest ascent and $-\nabla\ell$ is the direction of steepest descent.*

Proof. The Cauchy–Schwarz inequality (Section 33.1) states that $|\mathbf{v} \cdot \mathbf{g}| \le \|\mathbf{v}\|\|\mathbf{g}\|$, with equality if and only if $\mathbf{v}$ and $\mathbf{g}$ are collinear. For a unit vector $\mathbf{v}$ and $\mathbf{g} = \nabla\ell$, this reads

$$-\|\nabla\ell\| \;\le\; \mathbf{v} \cdot \nabla\ell \;\le\; \|\nabla\ell\|.$$

Each bound is attained uniquely by a unit vector collinear with $\nabla\ell$: the upper bound by $\mathbf{v} = \nabla\ell/\|\nabla\ell\|$ and the lower bound by $\mathbf{v} = -\nabla\ell/\|\nabla\ell\|$. ∎

In our running example at $(0, \log 2)$, the steepest descent direction is $-(1,2)/\sqrt{5}$, with slope $-\|\nabla f\| = -\sqrt{5}/3 \approx -0.745$, the most negative of the three slopes we computed a moment ago. Taking a step along $-\nabla_{\mathbf{w}}\ell$ gives the familiar gradient-descent update

$$\mathbf{w} \leftarrow \mathbf{w} - \eta\, \nabla_{\mathbf{w}} \ell(\mathbf{w}).$$

Notice that the update does not normalize the gradient. Its step length is $\eta\|\nabla\ell\|$, which naturally shrinks as the gradient gets smaller near a minimum.

It is easy to overlook that "unit" in the proposition means unit *Euclidean* length, and that this is a modeling choice rather than a law of nature. If we instead constrain each coordinate of the step to have magnitude at most one, the steepest descent step becomes $-\operatorname{sign}(\nabla\ell)$ applied coordinate by coordinate. At $(0, \log 2)$, that step is $(-1,-1)$, with slope $-(\frac{1}{3} + \frac{2}{3}) = -1$. Adaptive optimizers like Adam, which divide each coordinate of the update by a running estimate of its scale (Chapter 9), behave much more like this sign step than like Euclidean steepest descent. Exercise 4 explores steepest descent under other norms.

34.2.2.2 Gradients and Level Sets

We can also read the gradient directly off a contour map, without talking about steps at all. A *level set* of ℓ is the collection of all points where ℓ takes a fixed value c. Because moving along a level set keeps ℓ constant, the directional derivative along any tangent direction should vanish.

Proposition (gradients are normal to level sets). *Let ℓ be differentiable at $\mathbf{x}$. Then $\nabla\ell(\mathbf{x})$ is orthogonal to every direction tangent at $\mathbf{x}$ to the level set $\{\mathbf{x}' : \ell(\mathbf{x}') = c\}$ passing through $\mathbf{x}$.*

Proof. A vector $\mathbf{v}$ is *tangent* to the level set at $\mathbf{x}$ if $\mathbf{v} = \gamma'(0)$ for some differentiable curve $\gamma(t)$ lying inside the level set with $\gamma(0) = \mathbf{x}$. Along such a curve, $\ell(\gamma(t)) = c$ for all t, so the derivative of $t \mapsto \ell(\gamma(t))$ at $t = 0$ must be zero. To first order, $\gamma(t) = \mathbf{x} + t\mathbf{v} + o(t)$, so Equation 34.18 tells us that this derivative equals $\mathbf{v} \cdot \nabla\ell(\mathbf{x})$. Hence $\mathbf{v} \cdot \nabla\ell(\mathbf{x}) = 0$. ∎

Let us check this on our running example. The level set of f passing through $(0, \log 2)$ is the curve $e^x + e^y = 3$. Differentiating along it gives $e^x\,dx + e^y\,dy = 0$, so the tangent direction at $(0, \log 2)$ is $(e^y, -e^x) = (2, -1)$. Taking the dot product with the gradient confirms $(2,-1) \cdot (\frac{1}{3}, \frac{2}{3}) = 0$, the direction of zero slope we found earlier.

On a contour map, the gradient therefore crosses every contour at a right angle and points toward higher values. Moreover, if neighboring contours differ by Δc and sit a distance d apart, the slope across them is roughly $\Delta c / d$, so the gradient is longest wherever the contours are packed most tightly (Figure 34.12).

34.2.2.3 Tangent Planes

If we drop the error term from the linear approximation Equation 34.18 and write the predicted height as z, we obtain the *tangent plane* to the graph of f at $\mathbf{x}_0$,

$$z = f(\mathbf{x}_0) + \nabla f(\mathbf{x}_0) \cdot (\mathbf{x} - \mathbf{x}_0). \tag{34.20}$$

For our running example at $(0, \log 2)$, this plane is $z = \log 3 + \frac{1}{3}x + \frac{2}{3}(y - \log 2)$. Rewriting Equation 34.20 as $\nabla f(\mathbf{x}_0) \cdot (\mathbf{x} - \mathbf{x}_0) - (z - f(\mathbf{x}_0)) = 0$ shows immediately that the vector

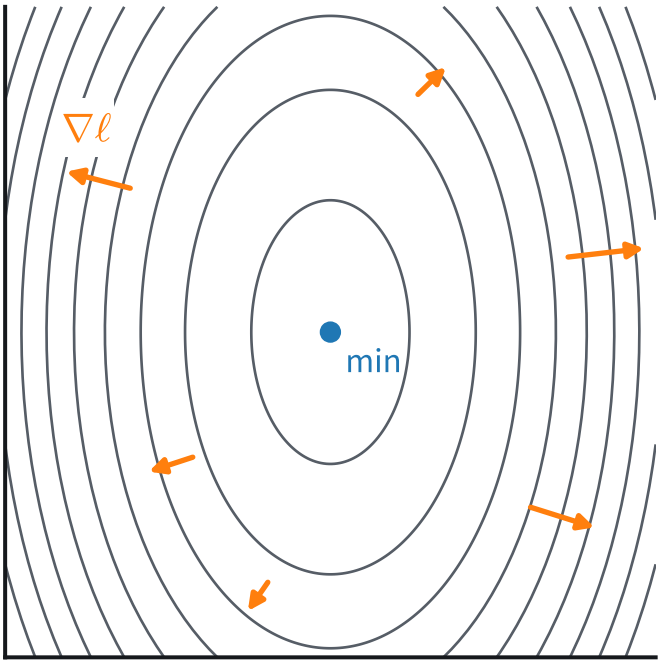


Figure 34.12: Contours of a loss ℓ (gray) with $\nabla \ell$ at several points. Each gradient is normal to its contour, points toward larger values, and is longest where the contours are packed most tightly.

$[\nabla f(\mathbf{x}_0), -1]^\top$ is normal to the tangent plane in $(\mathbf{x}, z)$-space. Dropping its height coordinate leaves $\nabla f(\mathbf{x}_0)$, which is the normal to the level curve in the base plane (Figure 34.13).

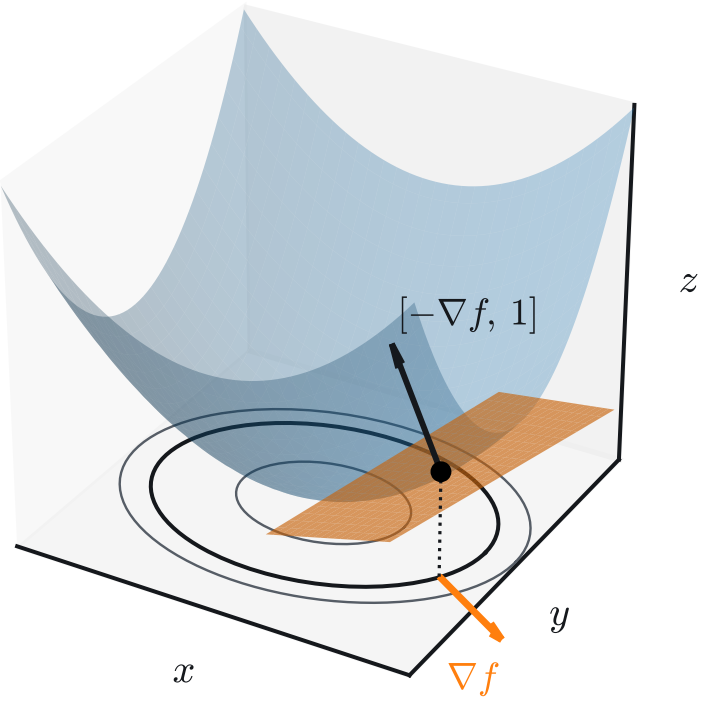


Figure 34.13: A surface $z = f(\mathbf{x})$ (blue) with its tangent plane at a point (orange). The vector $[-\nabla f, 1]$, the negative of $[\nabla f, -1]$, is normal to the plane (black arrow). Dropping the height coordinate leaves a multiple of ∇f, which on the base plane (orange arrow) is normal to the level curve through the point (black).

34.2.2.4 Critical Points

Deep learning losses are far too complicated to minimize in closed form, yet the linear model still gives a necessary condition that every minimum must obey, and gradient descent comes to rest right where this condition holds. Throughout this discussion, suppose f is defined on all of $\mathbb{R}^n$ and differentiable at $\mathbf{x}_0$.

If $\nabla f(\mathbf{x}_0) \neq \mathbf{0}$, we can step against it by setting $\epsilon = -\eta \nabla f(\mathbf{x}_0)$ with $\eta > 0$. By Equation 34.18, the resulting change in f is $-\eta \|\nabla f(\mathbf{x}_0)\|^2 + o(\eta)$, which is strictly negative for every sufficiently small η. Thus $\mathbf{x}_0$ cannot be a local minimum; repeating the argument with $+\eta$ shows it cannot be a local maximum either. Therefore, **at any local minimum or maximum, $\nabla f(\mathbf{x}_0) = \mathbf{0}$.** Points where the gradient vanishes are called *critical points*. This first-order condition is necessary, not sufficient, and the next example shows what else we need to be sure we have found a global extremum.

Consider $\psi(x, y) = x e^{-x^2 - y^2}$, a surface with a positive hill for $x > 0$ and a matching negative pit for $x < 0$. Its partial derivatives are

$$\frac{\partial \psi}{\partial x} = (1 - 2x^2)\, e^{-x^2 - y^2}, \qquad \frac{\partial \psi}{\partial y} = -2xy\, e^{-x^2 - y^2}.$$

Setting the second partial to zero forces either $x = 0$ or $y = 0$. Since $x = 0$ makes the first partial equal to $e^{-y^2} \neq 0$, we must have $y = 0$, and then $1 - 2x^2 = 0$ gives $x = \pm 1/\sqrt{2}$. The two critical points have values $\psi(\mp 1/\sqrt{2}, 0) = \mp e^{-1/2}/\sqrt{2} \approx \mp 0.4289$.

To conclude that $(-1/\sqrt{2}, 0)$ is the *global* minimum, however, we also need to know that a global minimum exists at all. Outside the disc of radius $R = 2$, the exponential decay forces $|\psi| \leq R e^{-R^2} < 0.04$, whereas inside the disc $\psi(-1/\sqrt{2}, 0) \approx -0.43$. Because ψ is continuous, it attains its minimum on the closed disc; since that minimum is below -0.04, it cannot lie on the boundary and must be a global minimum of ψ on all of $\mathbb{R}^2$, hence a critical point. Comparing our two candidates identifies $(-1/\sqrt{2}, 0)$ as the global minimum, and the same argument identifies $(+1/\sqrt{2}, 0)$ as the global maximum. Skip the existence check and the logic breaks down: x^3 has a single critical point yet no minimum anywhere.

Notice also what first-order information cannot do: both the minimum and the maximum have zero gradient, so the gradient alone cannot tell them apart. For that, we turn to second derivatives (Section 34.2.3.3).

34.2.3 The Hessian

34.2.3.1 Symmetry of the Hessian

The linear model captures only first-order behavior. Whether a critical point is a minimum, a maximum, or neither depends on how the surface curves away from its tangent plane, and curvature is measured by second derivatives. A function of n variables has n^2 second partial derivatives,

$$\frac{\partial^2 f}{\partial x_i \partial x_j} = \frac{\partial}{\partial x_i}\left(\frac{\partial f}{\partial x_j}\right),$$

which we collect into the *Hessian* matrix

$$\mathbf{H}_f(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1 \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_n} \end{bmatrix}. \tag{34.21}$$

Entry (i, j) measures how fast the j-th slope changes as we move along coordinate x_i. The Hessian is also written $\nabla^2 f$, and when the function and point are clear from context we simply write $\mathbf{H}$.

For log-sum-exp, let us write $(x_1, x_2) = (x, y)$ and differentiate the softmax entries $p_i = e^{x_i}/(e^{x_1} + e^{x_2})$ once more. Applying the quotient rule gives $\partial p_i / \partial x_j = p_i \delta_{ij} - p_i p_j$, where δ_{ij} is 1 when $i = j$ and 0 otherwise. In matrix form,

$$\mathbf{H}_f = \operatorname{diag}(\mathbf{p}) - \mathbf{p}\mathbf{p}^{\top}, \qquad \mathbf{H}_f(0,\log 2) = \begin{bmatrix} \frac{1}{3}-\frac{1}{9} & -\frac{2}{9} \\ -\frac{2}{9} & \frac{2}{3}-\frac{4}{9} \end{bmatrix} = \frac{2}{9}\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}. \tag{34.22}$$

This 2×2 matrix has eigenvalue 0 along $(1,1)$ and eigenvalue $\frac{4}{9}$ along $(1,-1)$. The zero eigenvalue matches what we already saw: along the diagonal $(1,1)$, $f(x+t,y+t) = f(x,y)+t$ is completely linear and has no curvature at all. We will meet this positive semidefinite, singular Hessian again in the second-derivative test.

The n^2 entries of a Hessian are not independent. Whenever f is twice continuously differentiable, the order of differentiation does not matter and the Hessian is symmetric, a classical result known as the Clairaut–Schwarz theorem (Rudin 1976). That symmetry is crucial: it guarantees real eigenvalues and orthogonal eigenvectors via the spectral theorem (Section 33.3), which makes the second-derivative test possible.

Proposition (symmetry of the Hessian; Clairaut–Schwarz). *Suppose the first partial derivatives of f and the mixed partials $\frac{\partial^2 f}{\partial x_i \partial x_j}$ and $\frac{\partial^2 f}{\partial x_j \partial x_i}$ exist near $\mathbf{x}$, and suppose the mixed partials are continuous at $\mathbf{x}$. Then*

$$\frac{\partial^2 f}{\partial x_i \partial x_j}(\mathbf{x}) = \frac{\partial^2 f}{\partial x_j \partial x_i}(\mathbf{x}).$$

When this holds for all i, j, the Hessian is symmetric: $\mathbf{H}_f = \mathbf{H}_f^{\top}$.

Proof. The idea is to realize both mixed partials as limits of the *same* quantity, the symmetric second difference

$$\Delta_h = \frac{f(\mathbf{x}+h\mathbf{e}_i+h\mathbf{e}_j) - f(\mathbf{x}+h\mathbf{e}_i) - f(\mathbf{x}+h\mathbf{e}_j) + f(\mathbf{x})}{h^2}, \tag{34.23}$$

which is visibly unchanged if we swap i and j. Our task is to express Δ_h as a mixed partial evaluated at a nearby point. Define the one-variable function $\varphi(t) = f(\mathbf{x}+t\mathbf{e}_i+h\mathbf{e}_j) - f(\mathbf{x}+t\mathbf{e}_i)$, so that $h^2\Delta_h = \varphi(h)-\varphi(0)$. Applying the mean value theorem (Section 34.1.3) twice, first to φ and then along x_j to the difference of first partials inside φ', gives numbers τ and σ between 0 and h such that

$$\begin{aligned} \varphi(h)-\varphi(0) &= h\,\varphi'(\tau) = h\Big[\frac{\partial f}{\partial x_i}(\mathbf{x}+\tau\mathbf{e}_i+h\mathbf{e}_j) - \frac{\partial f}{\partial x_i}(\mathbf{x}+\tau\mathbf{e}_i)\Big] \\ &= h^2\,\frac{\partial^2 f}{\partial x_j \partial x_i}(\mathbf{x}+\tau\mathbf{e}_i+\sigma\mathbf{e}_j). \end{aligned}$$

Thus $\Delta_h = \frac{\partial^2 f}{\partial x_j \partial x_i}(\xi_h)$ at a point ξ_h inside the square with corners $\mathbf{x}$ and $\mathbf{x}+h\mathbf{e}_i+h\mathbf{e}_j$, at distance at most $\sqrt{2}\,|h|$ from $\mathbf{x}$. Swapping the roles of i and j leaves Δ_h unchanged and yields $\Delta_h = \frac{\partial^2 f}{\partial x_i \partial x_j}(\xi_h')$ at another point of the same square. Letting $h \to 0$, continuity of the mixed partials at $\mathbf{x}$ sends the two expressions to the two mixed partials at $\mathbf{x}$. Because they agree for every h, their limits must agree as well. $\blacksquare$

The continuity hypothesis is strictly needed: there exist functions whose mixed partials exist everywhere yet fail to be continuous at the origin, where they disagree.

34.2.3.2 The Second-Order Taylor Approximation

Just as the gradient supplies the best linear approximation to f, the Hessian supplies the best *quadratic* approximation. To see where the coefficients come from, it helps to read them off an exact quadratic first. Take $f(x_1, x_2) = a + b_1 x_1 + b_2 x_2 + c_{11} x_1^2 + c_{12} x_1 x_2 + c_{22} x_2^2$. Evaluating its value, gradient, and Hessian Equation 34.21 at the origin gives

$$f(0,0) = a, \qquad \nabla f(0,0) = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}, \qquad \mathbf{H}_f(0,0) = \begin{bmatrix} 2c_{11} & c_{12} \\ c_{12} & 2c_{22} \end{bmatrix},$$

and assembling these pieces recovers the polynomial exactly: $f(\mathbf{x}) = f(\mathbf{0}) + \nabla f(\mathbf{0}) \cdot \mathbf{x} + \frac{1}{2} \mathbf{x}^\top \mathbf{H}_f(\mathbf{0})\, \mathbf{x}$. The factor of $\frac{1}{2}$ simply cancels the factor of 2 produced when we differentiate a square.

For a general function f, this quadratic is no longer exact, but restricting f to a line segment shows how accurate it is. Suppose f is twice continuously differentiable near the segment connecting $\mathbf{x}_0$ to $\mathbf{x}$. Write the displacement as $\delta = \mathbf{x} - \mathbf{x}_0$, and consider the single-variable slice $\varphi(t) = f(\mathbf{x}_0 + t\delta)$. By the linear model Equation 34.18, its first derivative is $\varphi'(t) = \sum_i \frac{\partial f}{\partial x_i}(\mathbf{x}_0 + t\delta)\, \delta_i$. Because each partial derivative in this sum is itself differentiable, applying the same step once more gives the second derivative: $\varphi''(t) = \sum_{i,j} \delta_j \frac{\partial^2 f}{\partial x_j \partial x_i}(\mathbf{x}_0 + t\delta)\, \delta_i = \delta^\top \mathbf{H}_f(\mathbf{x}_0 + t\delta)\, \delta$.

Now apply the one-variable Taylor theorem with Lagrange remainder (Section 34.1.5.2) of degree one to φ between 0 and 1: there is some $\tau \in (0,1)$ such that $\varphi(1) = \varphi(0) + \varphi'(0) + \frac{1}{2}\varphi''(\tau)$. Translating back to f gives

$$f(\mathbf{x}) = f(\mathbf{x}_0) + \nabla f(\mathbf{x}_0) \cdot \delta + \tfrac{1}{2} \delta^\top \mathbf{H}_f(\xi)\, \delta, \qquad \xi = \mathbf{x}_0 + \tau\delta.$$

This identity is exact, at the cost of evaluating the Hessian at an unknown intermediate point ξ on the segment rather than at $\mathbf{x}_0$. Because the second partials are continuous, replacing $\mathbf{H}_f(\xi)$ by $\mathbf{H}_f(\mathbf{x}_0)$ changes the quadratic term by at most $\frac{1}{2}\|\mathbf{H}_f(\xi) - \mathbf{H}_f(\mathbf{x}_0)\|\, \|\delta\|^2 = o(\|\delta\|^2)$. That yields the *second-order Taylor approximation*:

$$\begin{aligned} f(\mathbf{x}) = f(\mathbf{x}_0) &+ \nabla f(\mathbf{x}_0) \cdot (\mathbf{x} - \mathbf{x}_0) \\ &+ \frac{1}{2} (\mathbf{x} - \mathbf{x}_0)^\top \mathbf{H}_f(\mathbf{x}_0) (\mathbf{x} - \mathbf{x}_0) + o(\|\mathbf{x} - \mathbf{x}_0\|^2). \end{aligned} \tag{34.24}$$

This quadratic model misses f only by an error that vanishes faster than the squared step size, and it is the unique quadratic with that property. Whenever f has three continuous derivatives, the error is of third order in the step.

Along any single direction $\mathbf{v}$, a step $t\mathbf{v}$ produces a quadratic term of $\frac{t^2}{2} \mathbf{v}^\top \mathbf{H} \mathbf{v}$. The number $\mathbf{v}^\top \mathbf{H} \mathbf{v} = \frac{d^2}{dt^2} f(\mathbf{x}_0 + t\mathbf{v})\big|_{t=0}$ is the *second directional derivative*: it measures the curvature of the slice of f along $\mathbf{v}$.

We can now explain the gap left by the linear model in our first code cell. For the step $\epsilon = (0.01, -0.03)$ from $(0, \log 2)$, the Hessian Equation 34.22 predicts a second-order correction of $\frac{1}{2}\epsilon^\top \mathbf{H}\epsilon = \frac{1}{9}(\epsilon_1 - \epsilon_2)^2 = \frac{1}{9}(0.04)^2 \approx 1.78 \times 10^{-4}$, which closely matches the measured gap of 1.786×10^{-4}. Let us verify both the eigenvalues of $\mathbf{H}$ and the predicted gaps in code.

```
p = grad_lse(xy0)                # the softmax at (0, log 2)
H = np.diag(p) - np.outer(p, p)  # Hessian of log-sum-exp
print('9 H =', (9 * H).round(6).tolist())
print('eigenvalues:', np.linalg.eigvalsh(H).round(6) + 0.0)  # no '-0.'
for scale in [1, 0.1]:
    eps = scale * np.array([0.01, -0.03])
    print(f'|eps| {np.linalg.norm(eps):.1e}: '
          f'predicted gap {0.5 * eps @ H @ eps:.4e}')
```

```
9 H = [[2.0, -2.0], [-2.0, 2.0]]
eigenvalues: [0.       0.444444]
|eps| 3.2e-02: predicted gap 1.7778e-04
|eps| 3.2e-03: predicted gap 1.7778e-06
```

Both the matrix and its eigenvalues, 0 and $\frac{4}{9}$, match Equation 34.22. The predicted gaps of 1.7778×10^{-4} and 1.7778×10^{-6} agree with the measured 1.786×10^{-4} and 1.779×10^{-6} to two or three significant digits. The tiny leftover of about 10^{-6} in the first gap comes from the third-order term, and when the step shrinks by a factor of ten, that leftover drops by roughly a thousandfold, disappearing below the printed digits.

In the log-sum-exp example, one direction was completely flat. Let us look at a function that bends in both directions: $\psi(x, y) = xe^{-x^2-y^2}$ at the base point $\mathbf{x}_0 = (-1, 0)$. From the partial derivatives we computed earlier, $\psi(-1, 0) = -e^{-1}$ and $\nabla\psi(-1, 0) = (-e^{-1}, 0)$, and differentiating once more gives $\mathbf{H}_\psi(-1, 0) = 2e^{-1}\mathbf{I}$. Substituting these into Equation 34.24 yields the quadratic model $q(x, y) = e^{-1}(-1 - (x + 1) + (x + 1)^2 + y^2)$. As Figure 34.14 shows, the surface and its quadratic approximation are nearly indistinguishable near $(-1, 0)$ before pulling apart farther away.

If the error of this quadratic model is indeed third order, doubling the step size should multiply the gap by about eight. The next cell evaluates ψ and q at steps of d in each coordinate from $(-1, 0)$ to check.

```
def psi(x, y):
    return x * np.exp(-x**2 - y**2)

def quad(x, y):  # second-order Taylor model of psi at (-1, 0)
    return np.exp(-1) * (-1 - (x + 1) + (x + 1)**2 + y**2)

for d in [0.0, 0.05, 0.1, 0.3]:  # step d in each coordinate from (-1, 0)
    x, y = -1 + d, d
    print(f'step {d:.2f}: psi = {psi(x, y):.6f}, quadratic = {quad(x, y):.6f}, '
          f'gap = {abs(psi(x, y) - quad(x, y)):.6f}')
```

```
step 0.00: psi = -0.367879, quadratic = -0.367879, gap = 0.000000
step 0.05: psi = -0.384315, quadratic = -0.384434, gap = 0.000119
```

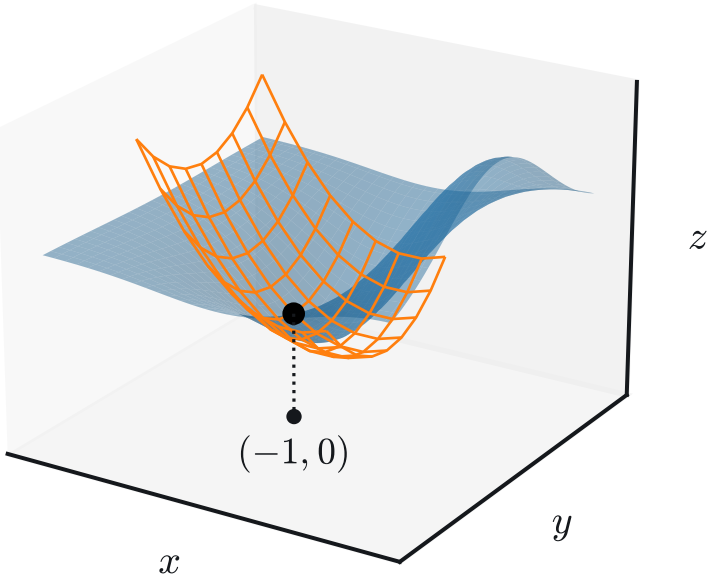


Figure 34.14: The surface $z = xe^{-x^2-y^2}$ (blue) and its second-order Taylor quadratic at the base point $(-1, 0)$ (orange). The two agree in value, slope, and curvature there, so they are nearly indistinguishable near the base point and separate farther out.

```
step 0.10: psi = -0.396388, quadratic = -0.397310, gap = 0.000921
step 0.30: psi = -0.391929, quadratic = -0.412025, gap = 0.020096
```

As expected, the gap vanishes at the base point, and doubling the step from 0.05 to 0.1 multiplies it by just under eight (from 0.000119 to 0.000921), matching a third-order remainder. Fitting this local quadratic and jumping directly to *its* minimum at each step is Newton's method, which Section 34.1.5.5 introduced in one variable and Section 9.1.5.1 develops as a multivariable optimizer.

34.2.3.3 The Second-Derivative Test

At a critical point $\mathbf{x}_0$, the linear term in Equation 34.24 disappears. Near $\mathbf{x}_0$, the function therefore behaves like its quadratic term up to an error of $o(\|\mathbf{x}-\mathbf{x}_0\|^2)$, so whether f rises or falls as we move away in a direction $\mathbf{v}$ is governed by the sign of $\mathbf{v}^\top \mathbf{H}\mathbf{v}$, provided that curvature is nonzero.

The eigenvalues of $\mathbf{H}$ organize these signs cleanly. Because $\mathbf{H}$ is symmetric, the spectral theorem (Section 33.3) guarantees an orthonormal basis of eigenvectors $\mathbf{q}_i$ with real eigenvalues λ_i. Expanded in this basis, the quadratic form becomes a simple weighted sum of squares:

$$\mathbf{v}^\top \mathbf{H}\mathbf{v} = \sum_{i=1}^{n} \lambda_i \, (\mathbf{q}_i^\top \mathbf{v})^2. \tag{34.25}$$

From this formula we can read the local geometry at a glance. If every $\lambda_i > 0$, the curvature is strictly positive in every direction and is at least $\lambda_{\min}\|\mathbf{v}\|^2$. If the eigenvalues have mixed signs, the surface bends upward along some eigenvectors and downward along others.

Proposition (second-order conditions). *Let f be twice continuously differentiable near a critical point $\mathbf{x}_0$, and write $\mathbf{H} = \mathbf{H}_f(\mathbf{x}_0)$.*

1. *If $\mathbf{x}_0$ is a local minimum, then $\mathbf{H} \succeq 0$ (all eigenvalues are ≥ 0).*
2. *If $\mathbf{H} \succ 0$ (all eigenvalues are > 0), then $\mathbf{x}_0$ is a strict local minimum.*
3. *If $\mathbf{H}$ has eigenvalues of both signs, then $\mathbf{x}_0$ is neither a local minimum nor a local maximum, regardless of any zero eigenvalues: it is a saddle point.*

The corresponding statements for maxima follow by applying the proposition to $-f$. When $\mathbf{H}$ is singular and semidefinite, neither case 2 nor case 3 applies, and the second-derivative test is inconclusive.

Proof. (1) Suppose instead that $\mathbf{v}^\top \mathbf{H}\mathbf{v} < 0$ for some direction $\mathbf{v}$. The one-variable slice $\varphi(t) = f(\mathbf{x}_0 + t\mathbf{v})$ has $\varphi'(0) = \nabla f(\mathbf{x}_0)\cdot\mathbf{v} = 0$ and $\varphi''(0) = \mathbf{v}^\top\mathbf{H}\mathbf{v} < 0$. By the single-variable second-derivative test, φ has a strict local maximum at $t = 0$, so f takes values strictly below $f(\mathbf{x}_0)$ arbitrarily close to $\mathbf{x}_0$, contradicting the assumption that $\mathbf{x}_0$ is a local minimum.

(2) This is the step that requires care, because we must control the remainder uniformly in all directions at once. Using the exact Taylor identity with $\nabla f(\mathbf{x}_0) = \mathbf{0}$, we have $f(\mathbf{x}_0+\delta) - f(\mathbf{x}_0) = \tfrac{1}{2}\delta^\top \mathbf{H}_f(\xi)\delta$ for some ξ on the segment from $\mathbf{x}_0$ to $\mathbf{x}_0+\delta$. Split the Hessian as $\mathbf{H}_f(\xi) = \mathbf{H} + (\mathbf{H}_f(\xi) - \mathbf{H})$. By Equation 34.25, the leading term satisfies $\delta^\top\mathbf{H}\delta \geq \lambda_{\min}\|\delta\|^2$. By continuity of the second partials, $\|\mathbf{H}_f(\xi) - \mathbf{H}\| \leq \tfrac{1}{2}\lambda_{\min}$ for all

sufficiently small δ. Combining the two bounds gives $f(\mathbf{x}_0+\delta)-f(\mathbf{x}_0)\geq\frac{1}{2}(\lambda_{\min}-\frac{1}{2}\lambda_{\min})\|\delta\|^2=\frac{1}{4}\lambda_{\min}\|\delta\|^2>0$ for every sufficiently small $\delta\neq\mathbf{0}$.

(3) Along an eigenvector with $\lambda>0$, the slice φ satisfies $\varphi'(0)=0$ and $\varphi''(0)=\lambda>0$, so f takes strictly larger values than $f(\mathbf{x}_0)$ arbitrarily close to $\mathbf{x}_0$. Along an eigenvector with $\lambda<0$, it takes strictly smaller values. ■

Of the three statements, the second is the one to remember, and the only one whose proof takes real work. Notice also that the inconclusive case is not just a theoretical footnote: along a zero eigenvector the quadratic model is completely flat, leaving higher-order terms in charge. For example, x^2+y^4 and x^2-y^4 share the same Hessian $\operatorname{diag}(2,0)$ at the origin, yet the first has a strict minimum there while the second has a saddle (Exercise 3 pushes this phenomenon even further). Figure 34.15 illustrates all four cases on the quadratic models from our running examples.

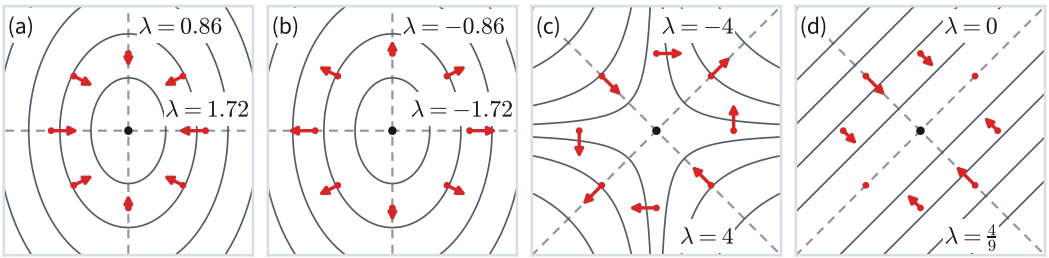


Figure 34.15: Contours of the quadratic model $\frac{1}{2}\mathbf{v}^\top\mathbf{H}\mathbf{v}$ (gray), its eigenvector directions with their eigenvalues (dashed), and descent arrows $-\mathbf{H}\mathbf{v}$ (red). (a) $\mathbf{H}\succ 0$: a minimum, every descent arrow points in. (b) $\mathbf{H}\prec 0$: a maximum. (c) Indefinite: a saddle; descent flows in along one eigenvector and out along the other. (d) Singular and semidefinite: the quadratic model is flat along the zero eigenvector, and second order is inconclusive.

In Figure 34.15, panel (a) shows the Hessian of ψ at $(-1/\sqrt{2},0)$, which equals $e^{-1/2}\operatorname{diag}(2\sqrt{2},\sqrt{2})$, while panel (b) shows its negative at $(+1/\sqrt{2},0)$. Panel (c) shows the origin of x^4+y^4-4xy, whose Hessian is $\left(\begin{smallmatrix}0&-4\\-4&0\end{smallmatrix}\right)$. Panel (d) reuses the log-sum-exp Hessian Equation 34.22 purely as an example matrix: log-sum-exp itself has no critical point, because its gradient, the softmax, never vanishes.

What happens when we implement the second-derivative test numerically and run it on a flat direction? A Hessian built from finite differences almost never produces an exact zero eigenvalue, so we must choose a tolerance below which an eigenvalue is treated as zero. In the next cell, the centered mixed difference `fxy` is the second difference Equation 34.23 with step $2h$ anchored at $(x-h,y-h)$.

```
def hessian(f, x, y, h=1e-4):  # 2x2 Hessian from second differences
    fxx = (f(x + h, y) - 2 * f(x, y) + f(x - h, y)) / h**2
    fyy = (f(x, y + h) - 2 * f(x, y) + f(x, y - h)) / h**2
    fxy = (f(x + h, y + h) - f(x + h, y - h)
           - f(x - h, y + h) + f(x - h, y - h)) / (4 * h**2)
    return np.array([[fxx, fxy], [fxy, fyy]])

def classify(lo, hi, tol=1e-6):  # from the smallest, largest eigenvalue
```

```
    if lo > tol:
        return 'minimum'
    if hi < -tol:
        return 'maximum'
    if lo < -tol < tol < hi:
        return 'saddle'
    return 'inconclusive'

quartic = lambda x, y: x**4 + y**4 - 4 * x * y
trough = lambda x, y: x**2 + y**4
for fn, x0 in [(psi, -1 / np.sqrt(2)), (psi, 1 / np.sqrt(2)),
               (quartic, 0.0), (trough, 0.0)]:
    lo, hi = np.linalg.eigvalsh(hessian(fn, x0, 0.0))
    print(f'({x0:+.4f}, 0): eigenvalues {lo:+.3f}, {hi:+.3f} '
          f'-> {classify(lo, hi)}')
```

```
(-0.7071, 0): eigenvalues +0.858, +1.716 -> minimum
(+0.7071, 0): eigenvalues -1.716, -0.858 -> maximum
(+0.0000, 0): eigenvalues -4.000, +4.000 -> saddle
(+0.0000, 0): eigenvalues +0.000, +2.000 -> inconclusive
```

At $(-1/\sqrt{2}, 0)$, the eigenvalues come out to 0.858 and 1.716 (matching $\sqrt{2}e^{-1/2}$ and $2\sqrt{2}e^{-1/2}$), confirming the global minimum we found earlier. At $(+1/\sqrt{2}, 0)$, both flip sign, confirming a maximum. At the origin of $x^4 + y^4 - 4xy$, the eigenvalues are ± 4, giving a saddle whose directions of rise and fall, $(1, -1)$ and $(1, 1)$, are rotated away from the coordinate axes. The fourth point is the instructive one. At the origin of $x^2 + y^4$, finite differences return $2h^2 = 2 \times 10^{-8}$ along the flat direction; because this sits below our tolerance, the program rightly reports "inconclusive" even though the origin is a strict minimum. Second-order information simply cannot see the quartic rise, and only the tolerance prevents the numerical test from claiming otherwise.

34.2.3.4 Saddles in High Dimension

In n dimensions, a local minimum requires all n eigenvalues of the Hessian to be nonnegative, and a nondegenerate minimum requires all n to be strictly positive. If the eigenvalue signs at a random critical point behaved like independent coin flips, all n would land positive with probability 2^{-n}, already vanishingly small when n is large.

Surprisingly, this coin-flip heuristic actually *overestimates* the chance of finding a minimum. Consider random symmetric matrices from the Gaussian orthogonal ensemble, whose diagonal entries have twice the variance of the off-diagonal entries. Even at $n = 2$, the exact probability that both eigenvalues are positive is $(1 - 1/\sqrt{2})/2 \approx 0.146$ rather than 0.25 (Exercise 4). For large n, the probability decays far faster than exponentially, like $\exp\left(-\frac{\ln 3}{4} n^2\right)$ (Dean and Majumdar 2006), because the eigenvalues of a random symmetric matrix repel one another and almost never crowd onto one side of zero.

In random high-dimensional landscapes, then, critical points at typical (high) function values are overwhelmingly saddles. Only at low function values does the Hessian shift toward positive eigenvalues so that minima begin to dominate (Bray and Dean 2007). Neural networks exhibit the same behavior: when Newton's method locates critical points on the loss surfaces of small networks, the high-loss critical points are saddles (Dauphin et al. 2014). Practically, when an optimizer stalls at high loss, a poor local minimum is the least likely

explanation. The optimizer may be lingering near a saddle or, more often still, crossing a poorly conditioned region in which the gradient is not small at all.

34.2.4 Constrained Optimization and Lagrange Multipliers

Until now, our variables were free to move in any direction. Many machine learning problems instead restrict the parameters to a surface: weights normalized to unit length, probabilities constrained to sum to one, or the closest point on a curve to the origin. In all of these, we want to minimize $f(\mathbf{x})$ subject to a constraint $g(\mathbf{x}) = c$. The unconstrained condition $\nabla f = \mathbf{0}$ no longer holds, because at the best feasible point ∇f need not vanish; it only has to produce no first-order decrease along moves that stay on the constraint surface. Remarkably, what replaces $\nabla f = \mathbf{0}$ follows directly from the geometry of level sets, without any second derivatives at all.

Suppose f and g are differentiable. At a feasible point, the admissible infinitesimal moves are the directions $\mathbf{v}$ tangent to the constraint surface $\{g = c\}$. At a constrained local minimizer $\mathbf{x}^\star$, no admissible move can decrease f to first order, so $\mathbf{v} \cdot \nabla f(\mathbf{x}^\star) \geq 0$ for every tangent direction $\mathbf{v}$. Because $-\mathbf{v}$ is tangent as well, we must have equality: $\nabla f(\mathbf{x}^\star)$ is orthogonal to every tangent direction. By our level-set proposition, $\nabla g(\mathbf{x}^\star)$ is also orthogonal to every tangent direction, which forces the two gradients to align.

Proposition (Lagrange condition). *Let $\mathbf{x}^\star$ be a local minimizer of f on $\{\mathbf{x} : g(\mathbf{x}) = c\}$. Assume that f is differentiable at $\mathbf{x}^\star$, that g is continuously differentiable near $\mathbf{x}^\star$, and that $\nabla g(\mathbf{x}^\star) \neq \mathbf{0}$. Then there exists a scalar μ, called the Lagrange multiplier, such that*

$$\nabla f(\mathbf{x}^\star) = \mu \, \nabla g(\mathbf{x}^\star). \tag{34.26}$$

Proof. The proof relies on one geometric fact about the constraint set. Under the last two hypotheses, $\{g = c\}$ is a smooth surface near $\mathbf{x}^\star$, and every vector $\mathbf{v}$ satisfying $\nabla g(\mathbf{x}^\star) \cdot \mathbf{v} = 0$ is the velocity $\gamma'(0)$ of a differentiable curve γ lying in $\{g = c\}$ with $\gamma(0) = \mathbf{x}^\star$ (Rudin 1976). Along such a curve, the one-variable function $t \mapsto f(\gamma(t))$ has a local minimum at $t = 0$, so its derivative there must vanish. Just as in the level-set proof, that derivative equals $\mathbf{v} \cdot \nabla f(\mathbf{x}^\star)$. Consequently, $\nabla f(\mathbf{x}^\star)$ is orthogonal to the entire hyperplane of vectors $\mathbf{v}$ orthogonal to $\nabla g(\mathbf{x}^\star)$. The only vectors orthogonal to that hyperplane are scalar multiples of $\nabla g(\mathbf{x}^\star)$, so $\nabla f(\mathbf{x}^\star) = \mu \nabla g(\mathbf{x}^\star)$. $\blacksquare$

Geometrically, at a constrained optimum the gradients of f and g are parallel, meaning the level set of f through $\mathbf{x}^\star$ is tangent to the constraint surface. Algebraically, pairing the n equations of Equation 34.26 with the constraint $g(\mathbf{x}) = c$ gives $n + 1$ equations in the $n + 1$ unknowns $\mathbf{x}$ and μ.

As a concrete example, let us find the points on the hyperbola $xy = 1$ closest to the origin: we want to minimize $f(x, y) = x^2 + y^2$ subject to $g(x, y) = xy = 1$. The gradients are $\nabla f = (2x, 2y)$ and $\nabla g = (y, x)$, and $\nabla g \neq \mathbf{0}$ everywhere on the hyperbola. The Lagrange condition gives $2x = \mu y$ and $2y = \mu x$. Substituting the second equation into the first yields $4x = \mu^2 x$, or $x(4 - \mu^2) = 0$. Since $x = 0$ is impossible on $xy = 1$, we must have $\mu = \pm 2$. Choosing $\mu = -2$ forces $y = -x$ and hence $xy = -x^2 = 1$, which has no real solution. That leaves $\mu = 2$ and $x = y$, giving the two candidates $(1, 1)$ and $(-1, -1)$, both with $f = 2$. Because $f = x^2 + 1/x^2$ grows without bound as $x \to 0$ or $|x| \to \infty$ on each branch, a global minimum must exist, so these two points are indeed the closest ones. At $(1, 1)$, the circle $x^2 + y^2 = 2$ kisses the hyperbola tangently, and $\nabla f = (2, 2) = 2 \, \nabla g$ (Figure 34.16).

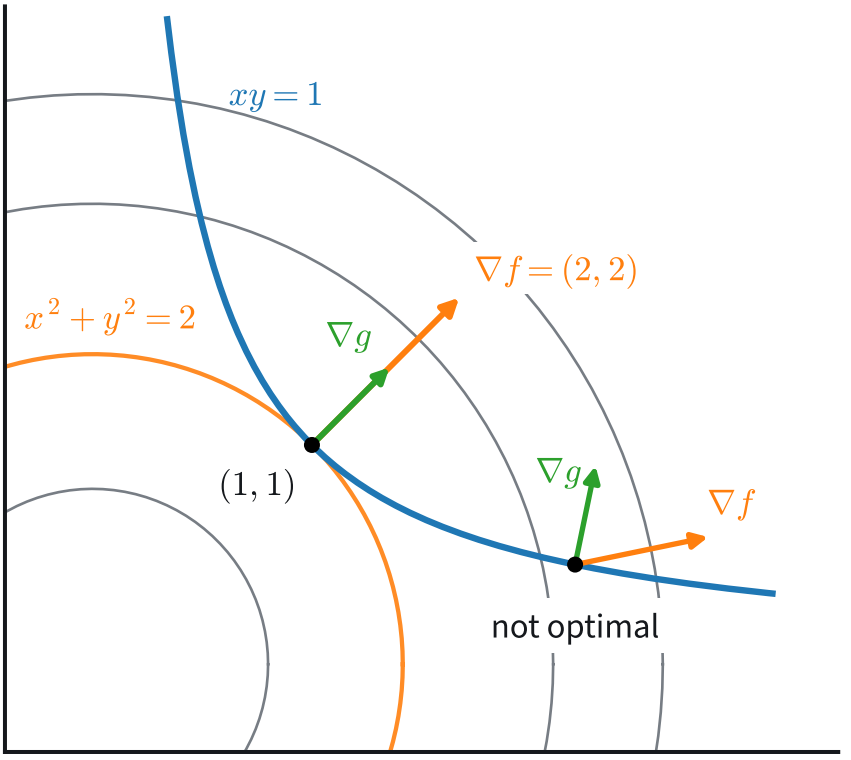


Figure 34.16: The closest point of $xy = 1$ to the origin. At $(1, 1)$ the circle $x^2 + y^2 = 2$ is tangent to the hyperbola and $\nabla f = (2, 2) = 2\nabla g$, so $\mu = 2$; the mirror solution $(-1, -1)$ is off the frame. At a feasible point that is not optimal the two gradients point in different directions, and ∇f keeps a component along the constraint.

The hyperbola makes the geometry easy to draw, but the same tool brings us straight back to our running example. Consider maximizing $\mathbf{z}\cdot\mathbf{p}-\sum_i p_i \log p_i$ over probability vectors $\mathbf{p}$ subject to $\sum_i p_i = 1$. Writing out the Lagrange condition gives $z_i - \log p_i - 1 = \mu$ for each coordinate i, which implies $p_i \propto e^{z_i}$. The optimal distribution is therefore the softmax, and the maximum value attained is the log-sum-exp (Section 35.7).

The regularity hypothesis $\nabla g(\mathbf{x}^\star) \neq \mathbf{0}$ cannot be dropped: without it, a Lagrange multiplier need not exist at all (Exercise 6 gives a clean counterexample). Later, Section 35.6 extends this condition to inequality constraints through the KKT conditions, and Section 35.7 develops the dual problem.

Every result in this section took the gradient as already known. For a two-variable formula on a blackboard, computing it by hand is effortless. For a neural network with a million weights, obtaining the gradient efficiently is the hard part.

34.2.5 Summary

A differentiable function changes, to first order, by the dot product of the step with its gradient; merely having partial derivatives does not guarantee this. Geometrically, the negative gradient points in the direction of steepest descent, and the gradient stands normal to every level set. One order higher, the symmetric Hessian defines the local quadratic approximation: at a critical point, the signs of its eigenvalues distinguish minima, maxima, and saddles, while a singular semidefinite Hessian leaves the test inconclusive. When the variables are restricted to a constraint surface $g(\mathbf{x}) = c$, the geometry of level sets replaces $\nabla f = \mathbf{0}$ with the Lagrange condition $\nabla f = \mu \nabla g$.

34.2.6 Exercises

1. **Directional derivatives without a gradient.** Let $f(x,y) = x^2 y/(x^2+y^2)$ for $(x,y) \neq (0,0)$ and $f(0,0) = 0$.
    1. Show that f is continuous at the origin.
    2. Compute the two partial derivatives of f at the origin and the directional derivative there along every unit vector $\mathbf{u} = [\cos\theta, \sin\theta]^\top$.
    3. Show that no vector $\mathbf{g}$ gives these directional derivatives as $\mathbf{u}\cdot\mathbf{g}$, so that f is not differentiable at the origin. Which hypothesis of the sufficient condition for differentiability in Section 34.2.1.2 fails? What does this function show that the counterexample $2xy/(x^2+y^2)$ there does not?
2. **Gradients of an elongated bowl.** Let $f(x,y) = x^2 + \kappa y^2$ with $\kappa \geq 1$. Its minimizer is the origin, and κ measures how much more strongly f curves along y than along x.
    1. Find a direction tangent to the level curve of f through $(1,1)$, check that it is orthogonal to $\nabla f(1,1)$, and compute the angle between $-\nabla f(1,1)$ and the direction $-[1,1]^\top$ from $(1,1)$ to the minimizer. At which points $(x,y) \neq (0,0)$ does the negative gradient point at the minimizer?
    2. Find the smallest value of the cosine of this angle over all points $(x,y) \neq (0,0)$, and the points where it is attained. (Hint: set $\rho = y/x$ and minimize over ρ.) Evaluate it for $\kappa = 10$. What does the result imply for a gradient-descent step on such a loss as κ grows?
3. **Flat directions.**

1. Give three functions of (x, y) that have a critical point at the origin with Hessian $\mathrm{diag}(2, 0)$ there: one with a strict local minimum, one with a local minimum that is not strict, and one without a local minimum.
2. The function $f(x, y) = (x - y^2)(x - 2y^2)$ also has Hessian $\mathrm{diag}(2, 0)$ at the origin. Show that its restriction to every line through the origin has a strict local minimum there. Sketch the region of the plane where $f < 0$, and show that the origin is nevertheless not a local minimum of f. What does this imply for testing a critical point by probing the loss along a few directions?
3. The *monkey saddle* $f(x, y) = x^3 - 3xy^2$ has a critical point at the origin. Compute its Hessian there, write f in polar coordinates, and find the directions along which f decreases. How does $\|\nabla f\|$ scale with the distance to the origin, compared with the saddle $x^2 - y^2$? What does this imply for gradient descent started near each of the two saddles?

4. code **Random Hessians.** ● A local minimum requires all n eigenvalues of the Hessian to be nonnegative, and a nondegenerate one requires them all positive; the section's coin-flip heuristic puts the latter at 2^{-n}. Test this heuristic on random symmetric matrices $\mathbf{A} = (\mathbf{G} + \mathbf{G}^\top)/\sqrt{2}$, where the $n \times n$ matrix $\mathbf{G}$ has independent standard normal entries.
   1. For $n = 2$, show that $s = (a_{11}+a_{22})/2$, $d = (a_{11}-a_{22})/2$, and a_{12} are independent standard normal variables, and that $\mathbf{A} \succ 0$ exactly when $s > \sqrt{d^2 + a_{12}^2}$. Use the rotational symmetry of the standard normal distribution in three dimensions to compute $P(\mathbf{A} \succ 0)$ exactly, and compare it with 2^{-2}.
   2. Estimate $P(\mathbf{A} \succ 0)$ by Monte Carlo for $n = 1, \dots, 6$, with 10^6 draws for $n \le 5$ and 10^7 draws, generated in batches, for $n = 6$. Plot $\log P$ against n together with $\log 2^{-n}$, and fit a quadratic in n to your estimates. Dean and Majumdar (2006) prove that $\log P \approx -\frac{\ln 3}{4} n^2$ for large n. Compare this coefficient with the quadratic coefficient of your fit. Does the heuristic over- or underestimate the probability of a minimum, and which property of the eigenvalues does it ignore?
   3. For large n the smallest eigenvalue of $\mathbf{A}$ lies close to $-2\sqrt{n}$. For $n = 5, 10, 20$, find the shift β at which half of the matrices $\mathbf{A} + \beta\mathbf{I}$ are positive definite, and compare it with $2\sqrt{n}$. In the random landscapes of Bray and Dean (2007), the Hessian at a critical point behaves like such a shifted matrix, with a shift that grows as the value of the random function at the critical point decreases. What does your result suggest about critical points of low loss?

5. [extended] **Saddles of a linear network.** ●● Let $\mathbf{A}$ be a symmetric $n \times n$ matrix with eigenvalues $\lambda_1 > \lambda_2 > \cdots > \lambda_n > 0$ and orthonormal eigenvectors $\mathbf{q}_1, \dots, \mathbf{q}_n$. A linear network with one hidden unit, input weights $\mathbf{v}$, and output weights $\mathbf{u}$ maps $\mathbf{x}$ to $\mathbf{u}\mathbf{v}^\top\mathbf{x}$. Trained on whitened inputs ($\mathbb{E}[\mathbf{x}\mathbf{x}^\top] = \mathbf{I}$) with targets $\mathbf{A}\mathbf{x}$, half its expected squared error is

$$\ell(\mathbf{u}, \mathbf{v}) = \tfrac{1}{2}\|\mathbf{A} - \mathbf{u}\mathbf{v}^\top\|_F^2.$$

   Baldi and Hornik (1989) showed that the local minima of such linear networks are global. This problem derives the complete set of critical points and compares it with the relation between loss and index that Dauphin et al. (2014) measured for nonlinear networks; the *index* of a critical point is the fraction of negative eigenvalues of its Hessian.

1. Compute $\nabla_{\mathbf{u}}\ell$ and $\nabla_{\mathbf{v}}\ell$. Show that the critical points are the origin and the points $\mathbf{u} = \alpha\mathbf{q}_i$, $\mathbf{v} = (\lambda_i/\alpha)\mathbf{q}_i$ with $\alpha \neq 0$, and compute ℓ at each of them.
2. Compute the eigenvalues of the $2n \times 2n$ Hessian at the balanced point $\mathbf{u} = \mathbf{v} = \sqrt{\lambda_i}\,\mathbf{q}_i$ and at the origin. (In the eigenvector basis, the Hessian splits into 2×2 blocks, one for each pair of coordinates along $\mathbf{q}_j$ in $\mathbf{u}$ and in $\mathbf{v}$.) How many eigenvalues are negative at each critical point? Explain the zero eigenvalue by a symmetry of ℓ, and conclude that every local minimum is global.
3. Take $n = 4$, $(\lambda_1, \dots, \lambda_4) = (4, 3, 2, 1)$, and a random orthonormal basis of eigenvectors. Run Newton's method from many random starting points of different scales, and record the loss and the index of every critical point it reaches. The Hessian is singular at every critical point other than the origin, so compute each Newton step by least squares. Plot the loss against the index, as in Figure 1 of Dauphin et al. (2014) . How often does Newton's method end at each kind of critical point, and where does gradient descent from small random initializations end?
4. Replace $\mathbf{u}\mathbf{v}^\top$ by $\mathbf{U}\mathbf{V}^\top$ with two hidden units, so that $\mathbf{U}$ and $\mathbf{V}$ are 4×2 matrices. Predict the loss values of the critical points from the rank-one analysis, and confirm them with Newton's method. How many zero eigenvalues does the Hessian have at the critical points, and which symmetry explains them? Is the index a monotone function of the loss?

6. **Lagrange multipliers and the Rayleigh quotient.** Let $\mathbf{A}$ be a symmetric $n \times n$ matrix. Maximize $f(\mathbf{x}) = \mathbf{x}^\top\mathbf{A}\mathbf{x}$ subject to $g(\mathbf{x}) = \|\mathbf{x}\|^2 = 1$.
   1. Write the Lagrange condition and show that it reads $\mathbf{A}\mathbf{x} = \mu\mathbf{x}$: every constrained critical point is a unit eigenvector, and its multiplier is the eigenvalue.
   2. Show that $f(\mathbf{x}) = \mu$ at every constrained critical point. Why does a constrained maximum exist? Conclude that the maximum is $\lambda_{\max}(\mathbf{A})$ and the minimum $\lambda_{\min}(\mathbf{A})$.
   3. For $\mathbf{A} = \left(\begin{smallmatrix} 2 & 1 \\ 1 & 2 \end{smallmatrix}\right)$ find the maximizers and minimizers and their values, and check at a maximizer that the level set of f through it is tangent to the unit circle.
   4. Which hypothesis of the Lagrange proposition does $g(\mathbf{x}) = \|\mathbf{x}\|^2$ satisfy at every point of the unit sphere? Now minimize the linear function $\mathbf{c}^\top\mathbf{x}$, $\mathbf{c} \neq \mathbf{0}$, subject to $\|\mathbf{x}\|^2 = 0$. What is the minimizer, and why does no multiplier μ satisfy $\nabla f = \mu\nabla g$ there?

34.3 The Chain Rule on Computational Graphs

Every training step in this book begins with a single call, `backward()` or `grad`, that returns the slope of the loss with respect to every weight in the network. In Section 34.2 we saw what those slopes mean: together they form the gradient, the vector that predicts how the loss changes under any small joint move of the parameters. What we have not yet explained is how a program computes millions of these slopes in roughly the time it takes to evaluate the network a few times. The naive approach, nudging one weight at a time and rerunning the network for each perturbation, requires a separate pass per weight and quickly becomes infeasible.

The chain rule does vastly better, and the reason is easy to see before writing down any algebra. A neural network is a deep composition of simple functions, so the derivatives with respect to its many weights share almost all of their work: every weight's influence has to travel through the later layers to reach the loss. A single nested formula hides this

shared structure, whereas a *computational graph* lays it bare: one node for each intermediate value and one directed edge for each direct dependence. Viewed on a graph, the chain rule becomes a sum over paths, and exploiting the shared work amounts to choosing how to group that sum. Grouping the paths from the output backward yields the derivative with respect to every input in a single pass. That grouping is backpropagation.

To make every step concrete and easy to verify by hand, we work on small graphs of scalar variables with integer values throughout. The entire development rests on one fact from Section 34.2: near any point where a function is differentiable, its change is approximated by the dot product of the step with its gradient, the linear model of Section 34.2.1.2. Extending the idea to layers that map vectors to vectors, and measuring its computational cost on real networks, comes in the two sections that follow.

34.3.1 The Chain Rule as a Sum over Paths

34.3.1.1 A Graph of Scalars

Let us start with a concrete example small enough to trace completely by hand. Four scalar inputs w, x, y, z feed through two layers of intermediate variables to produce a single scalar output:

$$\begin{aligned} f(u,v) &= (u+v)^2, \\ u(a,b) &= (a+b)^2, \qquad v(a,b) = (a-b)^2, \\ a(w,x,y,z) &= (w+x+y+z)^2, \qquad b(w,x,y,z) = (w+x-y-z)^2. \end{aligned} \tag{34.27}$$

These equations define a directed acyclic graph (Figure 34.17), where each node holds an intermediate value and each arrow indicates a direct functional dependence. There is nothing special about the squaring operations; we choose them simply because they keep all intermediate values and derivatives exact integers. Throughout this section, f denotes the scalar output of the graph under study.

34.3.1.2 Two Paths through One Step

Before tackling the whole graph, consider just its final composite step, $f(u(a,b), v(a,b))$, and ask how f responds when we nudge a by a small amount ϵ. Both intermediate nodes feel the perturbation according to their partial derivatives: u shifts to $u+\epsilon\frac{\partial u}{\partial a}$, and v shifts to $v+\epsilon\frac{\partial v}{\partial a}$. Substituting both shifts into the first-order linear approximation (Section 34.2.1.2) of f yields

$$f\big(u+\epsilon\tfrac{\partial u}{\partial a},\, v+\epsilon\tfrac{\partial v}{\partial a}\big) \approx f(u,v) + \epsilon\left[\frac{\partial f}{\partial u}\frac{\partial u}{\partial a} + \frac{\partial f}{\partial v}\frac{\partial v}{\partial a}\right].$$

Reading off the coefficient of ϵ gives the derivative of the composite function with respect to a, which is the multivariable chain rule:

$$\frac{\mathrm{d}f}{\mathrm{d}a} = \frac{\partial f}{\partial u}\frac{\partial u}{\partial a} + \frac{\partial f}{\partial v}\frac{\partial v}{\partial a}.$$

Notice the two different symbols for differentiation; we will keep them strictly separate from here on. The curly ∂ denotes the derivative of a single node's defining function with respect

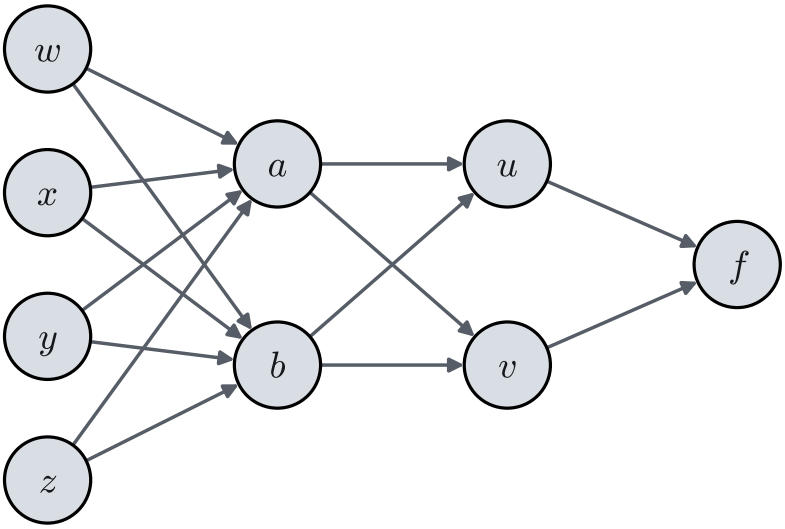


Figure 34.17: The dependency graph of Equation 34.27: each node represents a variable and each directed edge represents a direct functional dependence.

to one of its direct arguments, holding its other arguments fixed. The upright d denotes the *total* derivative $\mathrm{d}f/\mathrm{d}a$, which allows every variable downstream of a to respond. (For an input node, the total derivative coincides with the ordinary partial derivative of f viewed as a function of the inputs alone.) Geometrically, there are two directed *paths* by which a influences f, namely $a \to u \to f$ and $a \to v \to f$. Each path contributes the product of the derivatives along its edges, and the total derivative sums those contributions across all paths.

This derivation mirrors the one-variable proof of the chain rule (Section 34.1.2): we feed the linear change of the inner functions into the linear model of the outer one. Because two variables move simultaneously, however, f must be *differentiable* at (u, v), not merely possess partial derivatives. Recall the counterexample $2xy/(x^2+y^2)$ from Section 34.2.1.2: along the diagonal $x = y = t$, both partial derivatives vanish at the origin, so the chain rule would predict a derivative of 0 at $t = 0$, yet the function jumps discontinuously from 0 to 1 there.

34.3.1.3 Edge Derivatives and Total Derivatives

The same reasoning extends by induction to any directed acyclic graph. Define the *edge derivative* of a node with respect to one of its parents as the partial derivative of that node's function with respect to that parent, with all other parents held fixed. Then the total derivative of an output f with respect to an input x is

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \sum_{\text{directed paths } x \to \cdots \to f} \; \prod_{\text{edges on the path}} (\text{edge derivative}). \tag{34.28}$$

In words, each directed path from x to f multiplies the local rates of change along its edges, and the total derivative adds those path products together.

A graph with a skip connection makes it clear why edge derivatives and total derivatives must never be confused. Consider the five-node system

$$\begin{aligned} p(x,y) &= x+y, \qquad q(x,y) = x-y, \\ r(p) &= p^2, \qquad s(q) = q^2, \\ f(r,p,s) &= r+p+s. \end{aligned} \tag{34.29}$$

Here f depends on r and s, and also directly on p through a skip edge (Figure 34.18). Consequently, the input y reaches f along three distinct paths: $y \to p \to r \to f$, the shortcut $y \to p \to f$, and $y \to q \to s \to f$. The local edge derivatives are $\partial f/\partial r = \partial f/\partial p = \partial f/\partial s = 1$, $\partial r/\partial p = 2p$, $\partial s/\partial q = 2q$, $\partial p/\partial y = 1$, and $\partial q/\partial y = -1$. Summing the three path products gives

$$\begin{aligned} \frac{\mathrm{d}f}{\mathrm{d}y} &= \underbrace{1 \cdot 2p \cdot 1}_{y\to p\to r\to f} + \underbrace{1 \cdot 1}_{y\to p\to f} + \underbrace{1 \cdot 2q \cdot (-1)}_{y\to q\to s\to f} \\ &= 2p + 1 - 2q = 1 + 4y. \end{aligned}$$

We can check this directly by substituting everything into f: $f = (x+y)^2 + (x+y) + (x-y)^2$, whose derivative with respect to y is indeed $1 + 4y$. At $(x, y) = (1, 2)$, we have $p = 3$ and $q = -1$, so the three paths contribute $6 + 1 + 2 = 9 = 1 + 4 \cdot 2$. Notice the sharp distinction at node p: the *edge* derivative $\partial f/\partial p$ is just 1 (holding r fixed), whereas the *total* derivative $\mathrm{d}f/\mathrm{d}p$ accounts for both the skip edge and the indirect route through r, giving $2p + 1 = 7$. Confusing these two quantities is the single most common mistake when applying the multivariable chain rule.

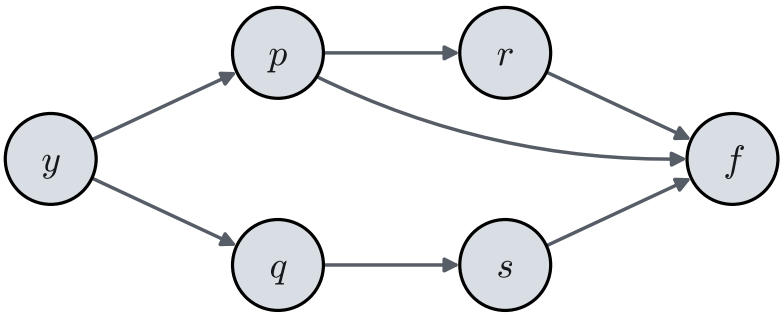


Figure 34.18: The dependency graph of Equation 34.29: the input y reaches f by three paths, namely $y \to p \to r \to f$, the skip edge $y \to p \to f$, and $y \to q \to s \to f$.

This path-sum viewpoint also explains how gradients travel through deep architectures. Mechanisms that open or close paths, such as the gates in an LSTM (Section 12.1) or the

skip connections in a residual network (Section 7.4), shape learning directly by controlling which paths carry gradient signal back to earlier layers.

34.3.2 Backpropagation

As a mathematical identity, the sum over paths Equation 34.28 is completely general: it gives every derivative of every computational graph. As a literal algorithm, however, it is hopeless, because the number of paths can grow exponentially with depth. Even a simple chain of m stages with two parallel nodes per stage already contains 2^m paths from start to finish. Fortunately, those paths overlap heavily, and any sub-product shared by many paths needs to be computed only once. The order in which we factor out those shared sub-products determines the computational cost.

34.3.2.1 Forward Mode

Let us compare two ways of factoring the sum on our four-input function Equation 34.27 at the point $(w, x, y, z) = (1, 1, -2, 1)$. A *forward pass* evaluates the nodes in topological order: $a = 1^2 = 1$, $b = 3^2 = 9$, $u = 10^2 = 100$, $v = (-8)^2 = 64$, and $f = 164^2 = 26896$. From these node values we read off all the local edge derivatives:

$$\begin{aligned}
\frac{\partial f}{\partial u} &= \frac{\partial f}{\partial v} = 2(u+v) = 328, \\
\frac{\partial u}{\partial a} &= \frac{\partial u}{\partial b} = 2(a+b) = 20, \\
\frac{\partial v}{\partial a} &= 2(a-b) = -16, \qquad \frac{\partial v}{\partial b} = -2(a-b) = 16, \\
\frac{\partial a}{\partial w} &= \frac{\partial a}{\partial x} = \frac{\partial a}{\partial y} = \frac{\partial a}{\partial z} = 2(w+x+y+z) = 2, \\
\frac{\partial b}{\partial w} &= \frac{\partial b}{\partial x} = 2(w+x-y-z) = 6, \qquad \frac{\partial b}{\partial y} = \frac{\partial b}{\partial z} = -6.
\end{aligned}$$

Each edge derivative is a local number computed solely from a node and its immediate parents.

To find $\mathrm{d}f/\mathrm{d}w$, notice that all four paths from w to f pass first through a or b and then through u or v. Factoring the paths from the input w forward through the graph gives

$$\begin{aligned}
\frac{\mathrm{d}f}{\mathrm{d}w} &= \frac{\partial f}{\partial u}\Big(\frac{\partial u}{\partial a}\frac{\partial a}{\partial w} + \frac{\partial u}{\partial b}\frac{\partial b}{\partial w}\Big) + \frac{\partial f}{\partial v}\Big(\frac{\partial v}{\partial a}\frac{\partial a}{\partial w} + \frac{\partial v}{\partial b}\frac{\partial b}{\partial w}\Big) \\
&= 328\,(20 \cdot 2 + 20 \cdot 6) + 328\,(-16 \cdot 2 + 16 \cdot 6) \\
&= 328 \cdot 224 = 73472.
\end{aligned}$$

The two parenthesized sums are simply $\mathrm{d}u/\mathrm{d}w = 160$ and $\mathrm{d}v/\mathrm{d}w = 64$, the total derivatives of the intermediate nodes with respect to w. Propagating the sensitivity to one input forward through the graph in this way is called *forward mode*. We can verify this calculation in plain Python:

```
w, x, y, z = 1, 1, -2, 1
# Forward pass: node values
a, b = (w + x + y + z)**2, (w + x - y - z)**2
u, v = (a + b)**2, (a - b)**2
f = (u + v)**2
print('a, b, u, v, f =', a, b, u, v, f)

# Edge derivatives
df_du, df_dv = 2*(u + v), 2*(u + v)
du_da, du_db, dv_da, dv_db = 2*(a + b), 2*(a + b), 2*(a - b), -2*(a - b)
da_dw, db_dw = 2*(w + x + y + z), 2*(w + x - y - z)

# Forward mode: push the dependence on w through the graph
du_dw, dv_dw = du_da*da_dw + du_db*db_dw, dv_da*da_dw + dv_db*db_dw
df_dw = df_du*du_dw + df_dv*dv_dw
print('df/dw =', df_dw)
```

```
a, b, u, v, f = 1 9 100 64 26896
df/dw = 73472
```

Forward mode returns $\mathrm{d}f/\mathrm{d}w = 73472$, but none of its intermediate derivatives can be reused for x, y, or z: both $\mathrm{d}u/\mathrm{d}w$ and $\mathrm{d}v/\mathrm{d}w$ are tied to w alone. Because training a neural network requires the derivative of a single scalar loss with respect to every parameter, computing them one input at a time leaves most of the shared structure unused.

34.3.2.2 Adjoints and the Backward Pass

The remedy is to reverse the perspective: keep the output fixed rather than the input. Forward mode fixed one input and asked how every downstream node responds to it. Now we fix the single output f and ask how it responds to every upstream node. Define the *adjoint* $\bar{n}$ of a node n as the total derivative $\mathrm{d}f/\mathrm{d}n$ of the output with respect to n: the sum of the edge-derivative products over all directed paths from n to f. Every such path must leave n through one of its immediate children c, and the remaining paths from c to f sum to $\bar{c}$. Grouping the path sum Equation 34.28 by this first edge turns it into a simple recursion:

$$\bar{f} = 1, \qquad \bar{n} = \sum_{\text{children } c \text{ of } n} \bar{c}\,\frac{\partial c}{\partial n}. \tag{34.30}$$

Only local edge derivatives and the adjoints of downstream nodes appear on the right-hand side. Starting from $\bar{f} = 1$ and visiting the nodes in reverse topological order, every child's adjoint is already known by the time we need it. On the skip-edge graph Equation 34.29, for instance, the recursion gives $\bar{r} = 1$ and $\bar{p} = \bar{r} \cdot 2p + \bar{f} \cdot 1 = 7$, matching the total derivative we found earlier. Applying the same sweep to our four-input graph gives:

- $\bar{u} = \bar{v} = 328$;
- $\bar{a} = \bar{u} \cdot 20 + \bar{v} \cdot (-16) = 1312$ and $\bar{b} = \bar{u} \cdot 20 + \bar{v} \cdot 16 = 11808$;
- $\bar{w} = \bar{x} = \bar{a} \cdot 2 + \bar{b} \cdot 6 = 73472$ and $\bar{y} = \bar{z} = \bar{a} \cdot 2 - \bar{b} \cdot 6 = -68224$.

A single *backward pass* thus produces the partial derivatives with respect to all four inputs at once (Figure 34.19). Let us run this backward sweep in Python, reusing the edge derivatives from the previous cell.

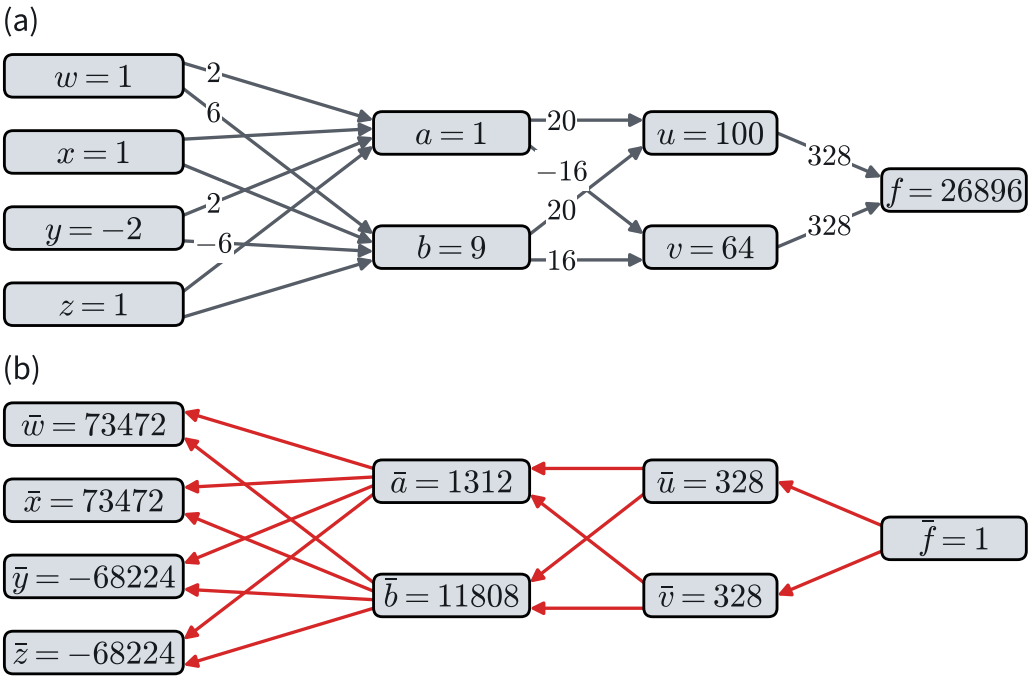


Figure 34.19: (a) The forward pass records every node value and local edge derivative. (b) The backward pass accumulates adjoints starting from $\bar{f} = 1$, giving $\bar{a} = 328 \cdot 20 + 328 \cdot (-16) = 1312$ and $\bar{w} = 1312 \cdot 2 + 11808 \cdot 6 = 73472$.

```
da_dy, db_dy = 2*(w + x + y + z), -2*(w + x - y - z)  # also for x, z
# Backward pass: adjoints from the output toward the inputs
u_bar, v_bar = df_du, df_dv
a_bar = u_bar*du_da + v_bar*dv_da
b_bar = u_bar*du_db + v_bar*dv_db
print(f'adjoints: u {u_bar}, v {v_bar}, a {a_bar}, b {b_bar}')
w_bar = a_bar*da_dw + b_bar*db_dw  # x enters a, b like w
y_bar = a_bar*da_dy + b_bar*db_dy  # z enters a, b like y
print('gradient:', {'w': w_bar, 'x': w_bar, 'y': y_bar, 'z': y_bar})
```

```
adjoints: u 328, v 328, a 1312, b 11808
gradient: {'w': 73472, 'x': 73472, 'y': -68224, 'z': -68224}
```

The derivatives with respect to w and x match, as do those with respect to y and z, because f depends on its inputs only through the combinations $w + x$ and $y + z$.

A quick operation count makes the advantage concrete. Forward mode needs six multiplications per input (two each for $\mathrm{d}u/\mathrm{d}w$, $\mathrm{d}v/\mathrm{d}w$, and $\mathrm{d}f/\mathrm{d}w$), or 24 in total for all four inputs. The backward pass needs four multiplications to compute $\bar{a}$ and $\bar{b}$ and then two per input, or 12 in total, because it traverses the shared upper part of the graph only once. A factor of two on a toy graph is modest, but the gap widens rapidly with the number of inputs: every additional input costs forward mode an entire pass through the graph, whereas the backward pass needs only two more multiplications.

34.3.2.3 Backpropagation in a Framework

Computing derivatives by propagating adjoints from the output back to the inputs is *backpropagation* (Rumelhart et al. 1986), which Section 4.3 applies to multilayer perceptrons. Every modern automatic differentiation engine executes the same two passes: it records intermediate values (or the operations needed to form edge derivatives) during the forward pass, and then accumulates adjoints during the backward pass. Let us check our hand computation against the framework's automatic differentiation on the same four inputs.

```
wxyz = torch.tensor([1., 1., -2., 1.], requires_grad=True)
w, x, y, z = wxyz
a, b = (w + x + y + z)**2, (w + x - y - z)**2
u, v = (a + b)**2, (a - b)**2
f = (u + v)**2
f.backward()  # one backward pass
print(wxyz.grad)
```

```
tensor([ 73472.,  73472., -68224., -68224.])
```

The framework returns $(73472, 73472, -68224, -68224)$, matching the adjoints from our hand calculation. The agreement is exact rather than approximate because every intermediate number is an integer below 2^{24}, which single precision represents without rounding error.

So far every node in our graphs has been a scalar, and we could count multiplications on the back of an envelope. In a real neural network, each layer maps a vector to a vector, so the edge derivatives become matrices and the order in which those matrices are multiplied matters far more than the order of scalar multiplications. We take that step in Section 34.4.

34.3.3 Summary

On a computational graph, the total derivative of an output with respect to an input equals the sum, over all directed paths between them, of the products of the local edge derivatives. The adjoint of a node, the total derivative of the output with respect to that node, collects every path leaving it. Factoring those paths by their first outgoing edge yields a recursion that, run backward from the output, computes every adjoint in a single pass. This reverse accumulation is backpropagation. Propagating one input forward instead computes only that input's derivative, which is why differentiating a scalar loss with respect to many parameters is always done backward.

34.3.4 Exercises

1. code **Backward pass for two tanh units.** A network with one input x, two hidden units, and one output computes, for a target t,

$$h_i = \tanh(w_i x + b_i) \quad (i = 1, 2), \qquad f = v_1 h_1 + v_2 h_2, \qquad \ell = \tfrac{1}{2}(f - t)^2.$$

   1. Draw the dependency graph of ℓ. In plain Python, write a forward pass that stores every intermediate value and a backward pass that returns the partial derivatives of ℓ with respect to all six parameters $w_1, b_1, w_2, b_2, v_1, v_2$. Check them against automatic differentiation at $x = 1.5$, $t = 0.5$, and $(w_1, b_1, w_2, b_2, v_1, v_2) = (0.5, 0, -1, 0.2, 1, -0.5)$. Which intermediate derivative does every one of the six partials reuse?
   2. Set $w_1 = 5$ and $x = 2$, keeping the other values. Compute $\partial \ell / \partial w_1$ with the plain-Python backward pass and explain its size from the product of edge derivatives along its path. Compare it with the value that automatic differentiation returns in single precision, and explain the difference. What does this imply for training a unit whose pre-activation is large?
   3. Set $w_1 = w_2$, $b_1 = b_2$, and $v_1 = v_2$. Show from the backward pass that both units receive identical gradients, so that gradient descent keeps them identical at every step. What does this imply for the initialization of hidden units?

2. **Paths through a residual stack.** ● In a one-dimensional model of the residual networks of Section 7.4, scalar blocks compute $x_{l+1} = x_l + g_l(x_l)$ for $l = 0, \dots, m-1$ with differentiable functions g_l.

   1. Use the chain rule to write $\partial x_m / \partial x_0$ as a product of m factors.
   2. Expand the product and match each term to a directed path from x_0 to x_m in the dependency graph. How many paths are there, and how many of them pass through exactly k of the functions g_l? Compare the number of paths with the number of multiplications that a backward pass needs to compute $\partial x_m / \partial x_0$.
   3. For a trained network of $m = 54$ residual blocks, Veit et al. (2016) found that almost all of the gradient flows along the paths through 5 to 17 blocks. What fraction of all paths are these? Now assume that $g_l'(x_l) = c > 0$ for every block. Which value of c makes the paths through about 11 blocks contribute the most to $\partial x_m / \partial x_0$, and what share of the derivative do the paths through 5 to 17 blocks then carry?

3. code **Gradient checks in single precision.** ● A gradient check compares a backward pass with the central difference $(f(w + h, x, y, z) - f(w - h, x, y, z))/(2h)$. Let

f be the function of Equation 34.27, whose exact partial derivative at $(w, x, y, z) = (1, 1, -2, 1)$ is $\partial f/\partial w = 73472$.

1. Implement f in NumPy and compute the central difference at this point for $h = 10^{-1}, 10^{-2}, \dots, 10^{-12}$, once in `float64` and once in `float32`. Plot the relative error against h on log-log axes for both precisions, and read off the best h and the smallest relative error of each.
2. The error of the central difference is a truncation part of order h^2 plus a rounding part of order $\varepsilon_{\text{mach}}|f|/h$, where the machine epsilon $\varepsilon_{\text{mach}}$ is about 2.2×10^{-16} for `float64` and 1.2×10^{-7} for `float32`. By which factors should the best h and the smallest relative error change from `float64` to `float32`? Compare the predicted factors with the plot.
3. Which tolerance on the relative error can a `float32` gradient check use without rejecting a correct backward pass? Why are gradient checks run in double precision?
4. Why does the `float32` estimate become exactly zero for the smallest values of h?

34.4 Matrix Calculus

In the last two sections we treated a loss as a function of scalar variables. In Section 34.2 we assembled its gradient as a column of partial derivatives, and in Section 34.3 we computed that column by a backward pass through a graph with one number per node. A neural network, however, is not written in scalars: each layer maps a vector to a vector, and its weights form a matrix. What is the derivative of a loss with respect to a vector or a matrix, and how do we compute it for a whole layer at once?

A network small enough to work by hand shows what is needed. Consider a network with two inputs, two ReLU hidden units, and one output,

$$g(\mathbf{x}) = \mathbf{w}_2^\top \varphi(\mathbf{W}_1\mathbf{x} + \mathbf{b}), \qquad \mathbf{W}_1 = \begin{pmatrix} 1 & 2 \\ 3 & -1 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 0 \\ -4 \end{pmatrix}, \quad \mathbf{w}_2 = \begin{pmatrix} 2 \\ -1 \end{pmatrix},$$

where φ applies $\max(0, \cdot)$ to each entry. At $\mathbf{x} = (1, 1)$ the pre-activations are $\mathbf{W}_1\mathbf{x} + \mathbf{b} = (3, -2)$, so the second hidden unit is off, the hidden vector is $(3, 0)$, and the output is $g = 6$. A training step raises three questions about this output. First, if the input moves by a small δ, the output changes by $2\delta_1 + 4\delta_2$, a row of two numbers that is the transpose of the column gradient. Second, if the weight matrix moves by $\Delta\mathbf{W}_1$, the output changes by $2\,\Delta W_{11} + 2\,\Delta W_{12}$ and not at all through the second row, so the derivative with respect to a matrix is itself a matrix, $\left(\begin{smallmatrix} 2 & 2 \\ 0 & 0 \end{smallmatrix}\right)$. Third, the hidden layer $\mathbf{x} \mapsto \varphi(\mathbf{W}_1\mathbf{x} + \mathbf{b})$ maps a vector to a vector. Neither a scalar slope nor a gradient vector describes its derivative, yet the output depends on $\mathbf{W}_1$ only through that layer.

Derivatives like these are surprisingly easy to get wrong by hand. If you drop a transpose on a square weight matrix, the dimensions still match, no error is raised, and training simply goes astray. Reference tables of matrix derivatives run to hundreds of entries, yet looking up formulas does little to protect against that kind of slip.

This section answers all three questions with a single object and a single principle. The derivative of a vector-valued map is a matrix, the *Jacobian*, defined as the map's best linear approximation. Composing maps multiplies their Jacobians, so the derivative of a network becomes a product of matrices, one per layer. Every gradient a layer needs (linear

and quadratic forms, least squares, matrix products, and softmax) then follows from one method: write down the first-order change of the function and read off the gradient as its coefficient. That same method produces the backward pass of each layer used in Section 4.3, as well as $\mathbf{p} - \mathbf{y}$, the classifier gradient of Section 3.1.2.2.

Three results are worth remembering: the product of Jacobians Equation 34.36, the rule that a gradient always has the shape of its variable, and the backward rule Equation 34.39 for a matrix product. You can skip the proof of the chain rule on a first reading, but not the differential method (Section 34.4.3.1), since everything that follows relies on it.

We use the following imports throughout this section.

```
import numpy as np
import torch
```

34.4.1 The Jacobian

34.4.1.1 The Best Linear Approximation

Near $\mathbf{x} = (1, 1)$ the hidden layer of our network is affine, so its linear approximation is exact there and leaves no error to study. Let us therefore begin with a curved map that we will reuse as a running example here and in the next section, returning to the network once the definition is in place:

$$\mathbf{f}(x, y) = \begin{pmatrix} x + \sin y \\ y + x^2/2 \end{pmatrix}, \qquad \mathbf{x}_0 = (0.5,\ 0.5).$$

At $\mathbf{x}_0$, the map evaluates to $\mathbf{f}(\mathbf{x}_0) = (0.9794,\ 0.6250)$. Each of the two outputs has two partial derivatives: $\partial f_1/\partial x = 1$, $\partial f_1/\partial y = \cos y = 0.8776$, $\partial f_2/\partial x = x = 0.5$, and $\partial f_2/\partial y = 1$. We arrange these four numbers in a matrix with one row per output and one column per input,

$$\mathbf{J}(\mathbf{x}_0) = \begin{pmatrix} 1 & \cos 0.5 \\ 0.5 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0.8776 \\ 0.5 & 1 \end{pmatrix}.$$

Column j records how both outputs respond when input j moves. This matrix is the derivative of $\mathbf{f}$ at $\mathbf{x}_0$, in the sense that $\mathbf{f}(\mathbf{x}_0 + \delta) \approx \mathbf{f}(\mathbf{x}_0) + \mathbf{J}\delta$ for small δ, and no other matrix approximates $\mathbf{f}$ as closely.

In one variable, the derivative is the slope of the best straight-line fit, $f(x + \delta) \approx f(x) + f'(x)\,\delta$. The vector-valued generalization rests on the same local approximation. We say that $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ is *differentiable* at $\mathbf{x}$ if there is a matrix $\mathbf{J} \in \mathbb{R}^{m \times n}$ such that

$$\mathbf{f}(\mathbf{x} + \delta) = \mathbf{f}(\mathbf{x}) + \mathbf{J}\,\delta + o(\|\delta\|), \tag{34.31}$$

meaning that the error of the linear model $\mathbf{f}(\mathbf{x}) + \mathbf{J}\delta$ divided by $\|\delta\|$ tends to zero as $\delta \to \mathbf{0}$. Whenever such a matrix exists, it is unique. We call it the *Jacobian* $\mathbf{J}_\mathbf{f}(\mathbf{x})$, and its entries are simply the partial derivatives:

$$[\mathbf{J}_\mathbf{f}]_{ij} = \frac{\partial f_i}{\partial x_j}. \tag{34.32}$$

To see why, substitute the perturbation $\delta = t\,\mathbf{e}_j$ (a small step along coordinate axis j) into Equation 34.31 and look at component i. Dividing by t and letting $t \to 0$ leaves $\partial f_i/\partial x_j$ in the (i, j) slot, regardless of which matrix $\mathbf{J}$ we started with. The partial derivatives therefore supply the entries of $\mathbf{J}$, while the linear approximation is what defines it.

The existence of partial derivatives alone does not guarantee differentiability. Consider the function $f(x, y) = 2xy/(x^2 + y^2)$ with $f(0, 0) = 0$ from Section 34.2.1.2: both partials are 0 at the origin, yet f equals 1 everywhere along the diagonal $x = y$. The only candidate Jacobian is the zero row vector, whose approximation error along the diagonal remains 1 no matter how small δ is, so f has no Jacobian at the origin. Continuity of the partial derivatives is sufficient to rule out such behavior: if every partial $\partial f_i/\partial x_j$ exists in a neighborhood of $\mathbf{x}$ and is continuous at $\mathbf{x}$, then $\mathbf{f}$ is differentiable at $\mathbf{x}$ (Rudin 1976).

Figure 34.20 illustrates the definition geometrically: near $\mathbf{x}_0$, the map acts like its Jacobian. A circle of radius r around $\mathbf{x}_0$ maps nearly onto the ellipse $\mathbf{f}(\mathbf{x}_0) + \mathbf{J}\delta$ ($\|\delta\| = r$), whose semi-axes are r times the singular values of $\mathbf{J}$ (as shown in Section 33.5, any matrix maps the unit circle to an ellipse with these semi-axes). Here the singular values are 1.706 and 0.329, so the circle of radius 0.35 becomes an ellipse with semi-axes 0.60 and 0.12. The largest gap between the true image and the ellipse is 0.061 at $r = 0.35$ and 0.015 at $r = 0.175$: halving the radius cuts the gap fourfold.

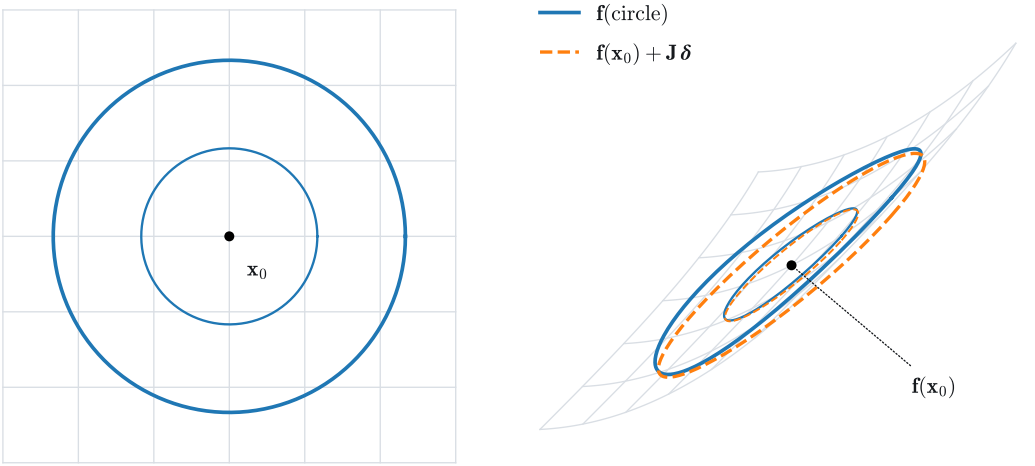


Figure 34.20: A differentiable map is locally its Jacobian. Two circles of radius 0.35 and 0.175 around $\mathbf{x}_0 = (0.5, 0.5)$ (left) and their images under $\mathbf{f}(x, y) = (x + \sin y,\ y + x^2/2)$ (right, blue) against the ellipses $\mathbf{f}(\mathbf{x}_0) + \mathbf{J}(\mathbf{x}_0)\delta$ (orange, dashed). The semi-axes of each ellipse are the radius times the singular values 1.71 and 0.33 of $\mathbf{J}$. Halving the radius cuts the largest gap between image and ellipse from 0.061 to 0.015, the $o(\|\delta\|)$ remainder.

The definition guarantees that the error of the linear model divided by $\|\delta\|$ tends to zero, but it does not say how fast. Let us shrink a perturbation tenfold at a time and inspect that ratio.

```
def f(v):                                   # the running map, R^2 -> R^2
    x, y = v
    return np.array([x + np.sin(y), y + x**2 / 2])

def J(v):                                   # its Jacobian, entry by entry
    x, y = v
    return np.array([[1.0, np.cos(y)], [x, 1.0]])

x0 = np.array([0.5, 0.5])
for s in [1, 0.1, 0.01, 0.001]:
    delta = s * np.array([0.2, 0.1])
    err = np.linalg.norm(f(x0 + delta) - f(x0) - J(x0) @ delta)
    print(f'|delta| = {np.linalg.norm(delta):.1e}   '
          f'error / |delta| = {err / np.linalg.norm(delta):.1e}')
```

```
|delta| = 2.2e-01   error / |delta| = 9.0e-02
|delta| = 2.2e-02   error / |delta| = 9.0e-03
|delta| = 2.2e-03   error / |delta| = 9.0e-04
|delta| = 2.2e-04   error / |delta| = 9.0e-05
```

The ratio drops from 0.09 to 0.009, 0.0009, and 0.00009. It tends to zero, confirming the $o(\|\delta\|)$ remainder in the definition. Notice that it also falls tenfold for every tenfold reduction in $\|\delta\|$, which means the error itself falls a hundredfold and is $O(\|\delta\|^2)$. The definition promises only the first property; the faster quadratic decay holds because $\mathbf{f}$ has continuous second derivatives.

34.4.1.2 Gradients, Hessians, and the Shape of a Derivative

Both the gradient and the Hessian from Section 34.2 are special cases of the Jacobian. Viewing them this way settles a shape question that arises in every hand derivation: whether a derivative is a row or a column, and how a derivative with respect to a matrix should be arranged.

For a scalar-valued function $f : \mathbb{R}^n \to \mathbb{R}$ ($m = 1$), the Jacobian is a single row, $\partial f/\partial \mathbf{x} = [\partial f/\partial x_1, \dots, \partial f/\partial x_n]$, and Equation 34.31 becomes the first-order expansion $f(\mathbf{x} + \delta) \approx f(\mathbf{x}) + (\partial f/\partial \mathbf{x})\,\delta$. In Section 34.2 we wrote the gradient as a column vector. The two are simply transposes of each other,

$$\nabla_{\mathbf{x}} f = \left(\frac{\partial f}{\partial \mathbf{x}}\right)^{\top}, \tag{34.33}$$

and we use both in this book, each for its own purpose. Jacobians of vector-valued maps are $m \times n$, with one row per output, so that the chain rule below composes cleanly without transposes. Gradients of scalars are column vectors, $\nabla_{\mathbf{x}} f$, matching the shape of $\mathbf{x}$ so an optimizer can subtract them directly. For our opening network, $\partial g/\partial \mathbf{x} = (2,\ 4)$ is a 1×2 Jacobian, while $\nabla_{\mathbf{x}} g = (2, 4)^{\top}$ is the column gradient.

A scalar function of a matrix $\mathbf{W} \in \mathbb{R}^{p \times q}$ needs one more convention. We define its gradient $\nabla_{\mathbf{W}} f$ as the $p \times q$ matrix with entries $\partial f/\partial W_{ij}$, matching the shape of $\mathbf{W}$. The first-order expansion then reads

$$f(\mathbf{W}+\mathbf{E}) = f(\mathbf{W}) + \langle \nabla_{\mathbf{W}} f,\ \mathbf{E}\rangle + o(\|\mathbf{E}\|), \qquad \langle \mathbf{A}, \mathbf{B}\rangle = \operatorname{tr}(\mathbf{A}^\top \mathbf{B}) = \sum_{ij} A_{ij} B_{ij}, \quad (34.34)$$

where the inner product $\langle \cdot, \cdot \rangle$ multiplies two matrices of the same shape entry by entry and sums the products, just like the ordinary dot product of the matrices flattened into vectors. In words, the first-order change in f is the sum over all entries of "sensitivity times nudge". In our opening example, $\nabla_{\mathbf{W}_1} g$ is the 2×2 matrix $\left(\begin{smallmatrix} 2 & 2 \\ 0 & 0 \end{smallmatrix}\right)$, and $\langle \nabla_{\mathbf{W}_1} g, \Delta\mathbf{W}_1 \rangle = 2\,\Delta W_{11} + 2\,\Delta W_{12}$ recovers the change we found earlier. This gives a single sanity check for every formula in the section: a gradient always has the shape of its variable. If a derived gradient does not, a transpose is missing.

Differentiating once more brings in the Hessian. The gradient field $\nabla f : \mathbb{R}^n \to \mathbb{R}^n$ is itself a vector-valued map, and its Jacobian is the matrix of second partials, the Hessian $\mathbf{H}_f$, whose entries are $[\mathbf{H}_f]_{ij} = \partial^2 f / \partial x_i \partial x_j$. Strictly speaking, $[\mathbf{J}_{\nabla f}]_{ij} = \partial^2 f / \partial x_j \partial x_i$ takes the two derivatives in the opposite order, but the two agree, and $\mathbf{H}_f$ is symmetric, whenever the mixed partials are continuous (the Clairaut–Schwarz theorem of Section 34.2).

34.4.2 The Chain Rule for Jacobians

A neural network is a composition of layers. On a graph of scalars, the chain rule summed the product of derivatives over every path from an input to an output (Section 34.3.1). When the layers are vector-valued maps, that sum over paths becomes a matrix product.

34.4.2.1 Composition as a Matrix Product

Proposition (chain rule). *If $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^p$ is differentiable at $\mathbf{x}$ and $\mathbf{g} : \mathbb{R}^p \to \mathbb{R}^m$ is differentiable at $\mathbf{f}(\mathbf{x})$, then $\mathbf{g} \circ \mathbf{f}$ is differentiable at $\mathbf{x}$ and*

$$\mathbf{J}_{\mathbf{g}\circ\mathbf{f}}(\mathbf{x}) = \mathbf{J}_{\mathbf{g}}\big(\mathbf{f}(\mathbf{x})\big)\, \mathbf{J}_{\mathbf{f}}(\mathbf{x}). \quad (34.35)$$

Proof. Write $\mathbf{y} = \mathbf{f}(\mathbf{x})$, $\mathbf{A} = \mathbf{J}_{\mathbf{f}}(\mathbf{x})$, and $\mathbf{B} = \mathbf{J}_{\mathbf{g}}(\mathbf{y})$. The idea is simply to substitute the linear model for $\mathbf{f}$ into the linear model for $\mathbf{g}$. By Equation 34.31, $\mathbf{f}(\mathbf{x}+\delta) = \mathbf{y} + \mathbf{A}\delta + o(\|\delta\|)$. Calling that displacement $\eta = \mathbf{A}\delta + o(\|\delta\|)$ and passing it to $\mathbf{g}$ gives

$$\mathbf{g}\big(\mathbf{f}(\mathbf{x}+\delta)\big) = \mathbf{g}(\mathbf{y}) + \mathbf{B}\,\eta + o(\|\eta\|) = \mathbf{g}(\mathbf{y}) + \mathbf{B}\mathbf{A}\,\delta + o(\|\delta\|).$$

The second equality relies on two facts. First, multiplying an $o(\|\delta\|)$ remainder by the fixed matrix $\mathbf{B}$ leaves it $o(\|\delta\|)$. Second, because $\|\eta\| \le \|\mathbf{A}\|\,\|\delta\| + o(\|\delta\|)$ is bounded by a constant multiple of $\|\delta\|$, the remainder $o(\|\eta\|)$ from $\mathbf{g}$ is also $o(\|\delta\|)$; this is the one step that is not purely algebraic. The coefficient of δ is therefore a matrix satisfying Equation 34.31, and by uniqueness it is the Jacobian of the composite: $\mathbf{J}_{\mathbf{g}\circ\mathbf{f}} = \mathbf{B}\mathbf{A}$. ∎

Looking at a single entry of the product recovers the scalar sum over paths: $[\mathbf{B}\mathbf{A}]_{ik} = \sum_j (\partial g_i / \partial y_j)(\partial f_j / \partial x_k)$ sums, over all intermediate variables y_j, the product of derivatives along each path from x_k to g_i.

34.4.2.2 The Jacobian of a Network

Returning to our opening network, we can view it as a composition of three maps: the affine transformation $\mathbf{a} = \mathbf{W}_1\mathbf{x} + \mathbf{b}$, the ReLU activation $\mathbf{h} = \varphi(\mathbf{a})$, and the linear output $g = \mathbf{w}_2^\top \mathbf{h}$. The first map is already affine, so its Jacobian is $\mathbf{W}_1$; the bias shifts the value without changing the slope. The ReLU acts on each entry independently, with slope 1 where $a_i > 0$ and 0 where $a_i < 0$, so its Jacobian is the diagonal matrix $\operatorname{diag}(\mathbf{m})$ formed from the *mask* $\mathbf{m}$ of these slopes. Finally, the output map has Jacobian $\mathbf{w}_2^\top$. At $\mathbf{x} = (1, 1)$, where $\mathbf{a} = (3, -2)$ and $\mathbf{m} = (1, 0)$, the chain rule multiplies the three factors:

$$\mathbf{J}_g(\mathbf{x}) = \mathbf{w}_2^\top \operatorname{diag}(\mathbf{m})\, \mathbf{W}_1 = \begin{pmatrix} 2 & -1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} 1 & 2 \\ 3 & -1 \end{pmatrix} = \begin{pmatrix} 2 & 0 \end{pmatrix} \begin{pmatrix} 1 & 2 \\ 3 & -1 \end{pmatrix} = \begin{pmatrix} 2 & 4 \end{pmatrix}.$$

The mask zeroes out the second entry of $\mathbf{w}_2^\top$, so the second row of $\mathbf{W}_1$ never reaches the output: the weights of an inactive unit contribute nothing to the derivative (Figure 34.21). The resulting row $(2, 4)$ matches the change $2\delta_1 + 4\delta_2$ from the opening. At a different input, the mask changes and so does the Jacobian. For instance, at $\mathbf{x} = (2, -3)$ the pre-activations are $(-4, 5)$, the mask flips to $(0, 1)$, and $\mathbf{J}_g = (0,\ -1)\, \mathbf{W}_1 = (-3,\ 1)$. Because ReLU is piecewise linear, a finite difference along each axis reproduces these entries exactly for any step small enough to keep the mask fixed (exercise 1).

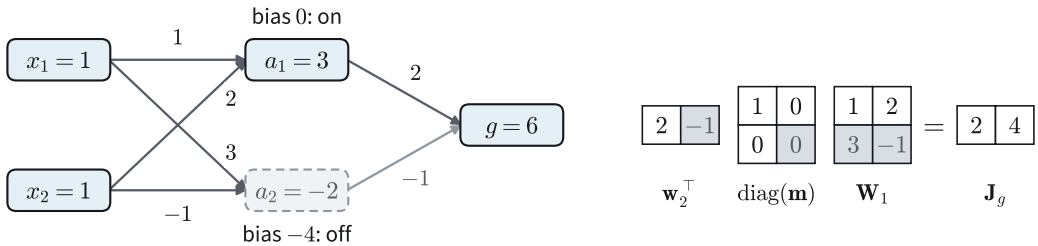


Figure 34.21: The Jacobian of the 2-2-1 ReLU network at $\mathbf{x} = (1, 1)$ is the product of three matrices. The second unit's pre-activation is -2, so its mask entry is 0 and the second row of $\mathbf{W}_1$ never reaches the output: $\mathbf{J}_g = \mathbf{w}_2^\top \operatorname{diag}(1, 0)\, \mathbf{W}_1 = (2,\ 4)$.

Every multilayer perceptron is built from the same two kinds of factors. A dense layer $\mathbf{x} \mapsto \mathbf{W}\mathbf{x}$ has Jacobian $\mathbf{W}$. An elementwise activation $\varphi(\mathbf{x}) = (\varphi(x_1), \dots, \varphi(x_n))$ does not couple distinct coordinates, so its Jacobian is diagonal: $\mathbf{J}_\varphi(\mathbf{x}) = \operatorname{diag}(\varphi'(x_1), \dots, \varphi'(x_n))$. This assumes that φ is differentiable at every x_i; ReLU has no derivative at 0, where frameworks supply a fixed value by convention (Section 34.1.6). A network of depth K is a composition $\mathbf{f} = \mathbf{f}_K \circ \cdots \circ \mathbf{f}_1$, and iterating Equation 34.35 turns its derivative into a single product,

$$\mathbf{J} = \mathbf{J}_K\, \mathbf{J}_{K-1} \cdots \mathbf{J}_1, \tag{34.36}$$

with each factor evaluated at the input of its own layer. For alternating dense layers and activations, this becomes $\mathbf{J} = \operatorname{diag}(\varphi_K')\, \mathbf{W}_K \cdots \operatorname{diag}(\varphi_1')\, \mathbf{W}_1$: weight matrices interleaved with diagonal matrices. Because multiplying a vector by a diagonal matrix simply scales each entry, each activation factor costs $O(n)$ operations rather than $O(n^2)$.

Let us now define the 2-2-1 network in code, which we will reuse later in the section, and compare the product $\mathbf{w}_2^\top \operatorname{diag}(\mathbf{m})\mathbf{W}_1$ against the gradient returned by automatic differentiation.

```
W1, b = torch.tensor([[1.0, 2.0], [3.0, -1.0]]), torch.tensor([0.0, -4.0])
w2, x = torch.tensor([2.0, -1.0]), torch.tensor([1.0, 1.0])

def g(x, W1, b, w2):                        # the 2-2-1 ReLU network
    return w2 @ torch.relu(W1 @ x + b)

m = (W1 @ x + b > 0).float()                # the ReLU mask
print('by hand    :', (w2 @ torch.diag(m) @ W1).numpy())
print('by autograd:', torch.func.grad(g)(x, W1, b, w2).numpy())
```

```
by hand    : [2. 4.]
by autograd: [2. 4.]
```

Both lines print $(2, 4)$, as expected.

The product Equation 34.36 also tells us how large or small a derivative can grow through a deep network. When every factor is the same matrix $\mathbf{W}$ (as in a recurrent network applying one weight matrix at every step, ignoring the activations), the product simplifies to $\mathbf{W}^K$. For a diagonalizable matrix $\mathbf{W} = \mathbf{V}\Lambda\mathbf{V}^{-1}$ we have $\mathbf{W}^K = \mathbf{V}\Lambda^K\mathbf{V}^{-1}$, so for large K its norm grows or shrinks like $|\lambda_{\max}|^K$, where $\lambda_{\max}$ is the eigenvalue of largest magnitude: $0.9^{50} \approx 0.005$, whereas $1.1^{50} \approx 117$. This exponential scaling is the vanishing and exploding gradient problem of deep and recurrent networks (Section 4.4, Section 8.5). Keep in mind that $|\lambda_{\max}|^K$ describes only the long-run rate. When the eigenvectors in $\mathbf{V}$ are far from orthogonal, the product can grow for many steps before eventually decaying (Section 33.4.2.3), and a product of distinct factors is governed by singular values rather than eigenvalues (Section 33.4.2.5).

Notice, however, an important limitation of Equation 34.36 for training: it gives the derivative with respect to the *input*. The weight matrices $\mathbf{W}_k$ appear in it as fixed factors rather than variables, so it does not yet tell us how the output changes when a weight changes. Computing weight gradients requires one more rule per layer, which Section 34.4.3.2 derives and applies to this same network.

34.4.2.3 The Order of Multiplication

Because matrix multiplication is associative, we may bracket the product Equation 34.36 in any order and obtain the exact same matrix. Yet different bracketings differ enormously in cost, by a factor that grows with the width of the network. In training we want the gradient of a scalar loss ℓ, so the chain starts on the left with a row vector: the $1 \times h$ Jacobian $\mathbf{u}^\top = \partial\ell/\partial\mathbf{h}_K$ of the loss with respect to the last layer's output $\mathbf{h}_K$. Multiplying from the left keeps every intermediate result a row vector, so every step is a matrix–vector product. Multiplying from the right instead forms expensive matrix–matrix products all the way across until the row vector finally arrives at the end. The cheap order is the one that keeps every step a matrix–vector product.

A quick operation count makes the gap concrete. Consider $\mathbf{u}^\top\mathbf{J}_3\mathbf{J}_2\mathbf{J}_1$ with shapes $1\times h$, $h\times h$, $h \times h$, and $h \times n$, taking $h = n = 1000$ and recalling that multiplying a $p \times q$ matrix by a $q \times r$ matrix costs pqr multiplications. From the left, we perform three matrix–vector

products for a total of $h^2 + h^2 + hn = 3 \times 10^6$ multiplications. From the right, forming $\mathbf{J}_2\mathbf{J}_1$ and then $\mathbf{J}_3(\mathbf{J}_2\mathbf{J}_1)$ costs $h^2 n = 10^9$ multiplications each, and the final product with $\mathbf{u}^\top$ adds $hn = 10^6$, totaling 2.001×10^9. Three million against two billion is a factor of about 670, or $2h/3$ (Figure 34.22).

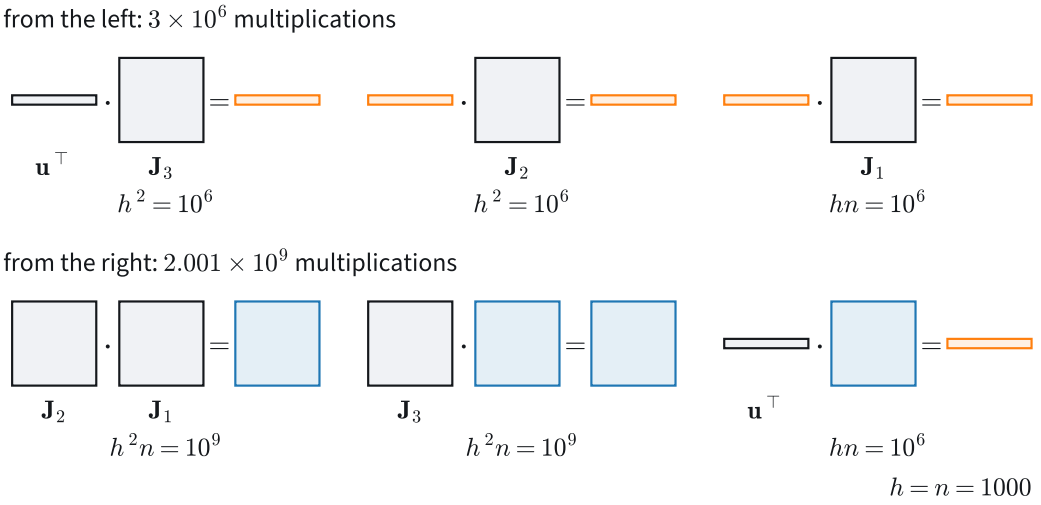


Figure 34.22: Two orders for $\mathbf{u}^\top\mathbf{J}_3\mathbf{J}_2\mathbf{J}_1$ with $h = n = 1000$. From the left every intermediate is a $1 \times h$ row and every step is a matrix–vector product: 3×10^6 multiplications. From the right the first two steps are matrix–matrix products of 10^9 multiplications each. The shapes, not the depth, decide the cost.

These two orders correspond to two directions through the network. Multiplying from the left propagates the sensitivity of the output backward through the layers, at the cost of one matrix–vector product per layer. Multiplying from the right is the cheap order when a column vector $\mathbf{v}$ stands at the right end of the chain: it propagates a single input direction forward and yields $\mathbf{J}\mathbf{v}$. Algorithms that automate these two orders are called reverse-mode and forward-mode automatic differentiation (Section 34.5).

The count above assumed layers of equal width. When the factors have varying shapes, neither end-to-end order need be optimal. For a chain, the cheapest parenthesization can be found by dynamic programming (exercise 2); for a general computation graph, the problem is NP-complete (Naumann 2008).

34.4.3 Derivatives with Respect to Vectors and Matrices

Training a layer requires the gradient of a scalar loss with respect to vectors and matrices alike: its input, its weights, and its bias. In this section we derive every gradient we need with a single method, one that automatically returns each result in the shape of its variable.

34.4.3.1 The Differential Method

Write $d\mathbf{x}$ for a small perturbation of the variable and df for the part of the resulting change in f that is linear in $d\mathbf{x}$—simply the term $\mathbf{J}\delta$ of Equation 34.31 with δ renamed $d\mathbf{x}$. By Equation 34.31 and Equation 34.34, this linear part is an inner product with the gradient,

$$df = \langle \nabla f,\, d\mathbf{x} \rangle, \tag{34.37}$$

for both vector variables ($\langle \mathbf{a}, \mathbf{b} \rangle = \mathbf{a}^\top \mathbf{b}$) and matrix variables. We can read Equation 34.37 in two ways. Given the gradient, it tells us how f responds to a nudge. Given f, it gives a direct recipe for the gradient: compute df using the ordinary rules of differentiation, rearrange it into the form $\langle \mathbf{M},\, d\mathbf{x} \rangle$, and read off $\nabla f = \mathbf{M}$. This reading is unambiguous because a linear function of $d\mathbf{x}$ can be written as $\langle \mathbf{M}, d\mathbf{x} \rangle$ for only one $\mathbf{M}$: setting $d\mathbf{x}$ to the matrix with a single 1 in position (i, j) and zeros elsewhere isolates M_{ij}.

Four familiar rules suffice to compute df. First, the differential is linear. Second, it obeys the product rule $d(\mathbf{AB}) = d\mathbf{A}\,\mathbf{B} + \mathbf{A}\,d\mathbf{B}$, because expanding $(\mathbf{A} + d\mathbf{A})(\mathbf{B} + d\mathbf{B})$ differs from $\mathbf{AB} + d\mathbf{A}\,\mathbf{B} + \mathbf{A}\,d\mathbf{B}$ only by the second-order term $d\mathbf{A}\,d\mathbf{B}$. Third, it commutes with transposition, $d(\mathbf{A}^\top) = (d\mathbf{A})^\top$. Fourth, a scalar function applied entry by entry obeys the scalar chain rule in every entry, $d\varphi(\mathbf{X}) = \varphi'(\mathbf{X}) \odot d\mathbf{X}$, where $\odot$ denotes elementwise multiplication.

To rearrange df into the form $\langle \mathbf{M}, d\mathbf{x} \rangle$, we need just one more tool. The *move rule* shifts a factor standing next to $d\mathbf{x}$ across to the other side of the inner product:

$$\langle \mathbf{A},\, \mathbf{BC} \rangle = \langle \mathbf{B}^\top \mathbf{A},\, \mathbf{C} \rangle = \langle \mathbf{AC}^\top,\, \mathbf{B} \rangle,$$

which follows directly from the cyclic property of the trace, $\operatorname{tr}(\mathbf{XY}) = \operatorname{tr}(\mathbf{YX})$: $\operatorname{tr}(\mathbf{A}^\top \mathbf{BC}) = \operatorname{tr}\big((\mathbf{B}^\top \mathbf{A})^\top \mathbf{C}\big) = \operatorname{tr}(\mathbf{CA}^\top \mathbf{B}) = \operatorname{tr}\big((\mathbf{AC}^\top)^\top \mathbf{B}\big)$. In words, a factor to the left of the variable moves across as its transpose on the left, while a factor to the right moves across as its transpose on the right. An elementwise factor moves across unchanged, $\langle \mathbf{A}, \mathbf{B} \odot \mathbf{C} \rangle = \langle \mathbf{A} \odot \mathbf{B}, \mathbf{C} \rangle$, since both sides equal $\sum_{ij} A_{ij} B_{ij} C_{ij}$.

The two simplest cases take one line each. For a linear form, $d(\mathbf{a}^\top \mathbf{x}) = \mathbf{a}^\top d\mathbf{x} = \langle \mathbf{a}, d\mathbf{x} \rangle$, so $\nabla_{\mathbf{x}}\, \mathbf{a}^\top \mathbf{x} = \mathbf{a}$. For a quadratic form, the product rule hits both copies of $\mathbf{x}$:

$$\begin{aligned} d(\mathbf{x}^\top \mathbf{A} \mathbf{x}) &= d\mathbf{x}^\top \mathbf{A} \mathbf{x} + \mathbf{x}^\top \mathbf{A}\, d\mathbf{x} = \mathbf{x}^\top \mathbf{A}^\top d\mathbf{x} + \mathbf{x}^\top \mathbf{A}\, d\mathbf{x} \\ &= \big\langle (\mathbf{A} + \mathbf{A}^\top)\mathbf{x},\, d\mathbf{x} \big\rangle. \end{aligned}$$

In the second equality we transposed a 1×1 matrix: $d\mathbf{x}^\top \mathbf{A} \mathbf{x}$ is a scalar, so it equals its own transpose $\mathbf{x}^\top \mathbf{A}^\top d\mathbf{x}$. Reading off the coefficient yields $\nabla_{\mathbf{x}}\, \mathbf{x}^\top \mathbf{A} \mathbf{x} = (\mathbf{A} + \mathbf{A}^\top)\mathbf{x}$, which simplifies to $2\mathbf{A}\mathbf{x}$ whenever $\mathbf{A}$ is symmetric.

Writing out indices offers a second route. Entry by entry, the same derivative is $\partial(\sum_{ij} x_i A_{ij} x_j)/\partial x_k = \sum_j A_{kj} x_j + \sum_i x_i A_{ik}$, recovering the k-th entry of $(\mathbf{A} + \mathbf{A}^\top)\mathbf{x}$. Index derivations are longer, but they leave no doubt about transposes; Section 4.3 uses them to derive the matrix-product rule.

A fast sanity check on any matrix identity is to set every matrix to 1×1, in which case the formula must reduce to the familiar scalar rule. Our quadratic-form gradient passes, since $\frac{d}{dx}(ax^2) = 2ax = (a + a)x$. This 1×1 *check* reliably catches a wrong coefficient or a missing term, but it is blind to transposes and factor order because 1×1 matrices commute and equal their own transposes. The shape rule catches most transpose errors, though unfortunately not when the factor is square—the exact pitfall we warned about at the start of the section. For square factors, only a numerical check will catch the mistake.

Take $\mathbf{A} = \left(\begin{smallmatrix} 0 & 1 \\ 0 & 0 \end{smallmatrix}\right)$ and $\mathbf{x} = (1, 2)$. Then $\mathbf{x}^\top \mathbf{A} \mathbf{x} = x_1 x_2$, whose true gradient $(x_2, x_1) = (2, 1)$ equals $(\mathbf{A} + \mathbf{A}^\top)\mathbf{x}$. The naive guess $2\mathbf{A}\mathbf{x} = (4, 0)$, suggested by the scalar rule $2ax$, passes the 1×1 check, has the right shape, and is still wrong.

The differential method handles matrix variables with no extra effort. For the least-squares loss $\ell = \|\mathbf{W}\mathbf{x} - \mathbf{y}\|^2$ with residual $\mathbf{r} = \mathbf{W}\mathbf{x} - \mathbf{y}$, varying $\mathbf{W}$ gives $d\mathbf{r} = d\mathbf{W}\,\mathbf{x}$, and the move rule shifts $\mathbf{x}$ to the other side: $d\ell = 2\mathbf{r}^\top d\mathbf{r} = \langle 2\mathbf{r},\ d\mathbf{W}\,\mathbf{x}\rangle = \langle 2\mathbf{r}\mathbf{x}^\top,\ d\mathbf{W}\rangle$. Therefore,

$$\nabla_{\mathbf{W}} \|\mathbf{W}\mathbf{x} - \mathbf{y}\|^2 = 2\,(\mathbf{W}\mathbf{x} - \mathbf{y})\,\mathbf{x}^\top, \tag{34.38}$$

a rank-one outer product of the residual with the input, automatically in the shape of $\mathbf{W}$. When $\mathbf{x}$ varies instead, $d\mathbf{r} = \mathbf{W}\,d\mathbf{x}$ and $\nabla_{\mathbf{x}} \ell = 2\mathbf{W}^\top \mathbf{r}$. In both cases, the gradient is twice the residual multiplied by the transpose of whichever factor multiplies the variable.

34.4.3.2 Backward Rules of a Layer

Inside a network, a layer $\mathbf{Y} = \mathbf{F}(\mathbf{X})$ is one step in a long computation ending in a scalar loss ℓ. Forming its full Jacobian is out of the question: a 1000×1000 weight matrix applied to a minibatch of 1000 examples produces 10^6 outputs from 10^6 weights, so the Jacobian of outputs with respect to weights would have 10^{12} entries. Fortunately, backpropagation needs far less. All a layer must do is turn the gradient of the loss with respect to its output into the gradients with respect to its inputs and weights. Following Section 34.3.2, we write $\bar{\mathbf{Y}} = \nabla_{\mathbf{Y}} \ell$ for the adjoint of $\mathbf{Y}$—now a matrix of the same shape as $\mathbf{Y}$ holding the total derivative of the loss with respect to each entry of $\mathbf{Y}$.

Proposition (backward rule of a layer). *Let $\mathbf{Y} = \mathbf{F}(\mathbf{X})$ be differentiable at $\mathbf{X}$ and let ℓ be a differentiable scalar function of $\mathbf{Y}$. Then the adjoint $\bar{\mathbf{X}}$ of the composite $\ell(\mathbf{F}(\mathbf{X}))$ is the unique matrix satisfying*

$$\langle \bar{\mathbf{X}},\ d\mathbf{X}\rangle = \langle \bar{\mathbf{Y}},\ d\mathbf{Y}\rangle \quad \textit{for every } d\mathbf{X},$$

where $d\mathbf{Y}$ is the first-order change of $\mathbf{F}$ caused by $d\mathbf{X}$.

Proof. By the chain rule Equation 34.35 (viewing matrices as flattened vectors), the linear part of the composite is the linear part of ℓ applied to the linear part of $\mathbf{F}$: $d\ell = \langle \bar{\mathbf{Y}}, d\mathbf{Y}\rangle$. Because $d\mathbf{Y}$ is linear in $d\mathbf{X}$, so is $d\ell$, which means it can be written as $\langle \mathbf{M}, d\mathbf{X}\rangle$ for a unique matrix $\mathbf{M}$. By Equation 34.37, $\mathbf{M} = \bar{\mathbf{X}}$. ■

In practice, applying this proposition comes down to applying the move rule. For a vector layer with $d\mathbf{y} = \mathbf{J}\,d\mathbf{x}$, it gives $\langle \bar{\mathbf{y}}, \mathbf{J}\,d\mathbf{x}\rangle = \langle \mathbf{J}^\top \bar{\mathbf{y}}, d\mathbf{x}\rangle$, so $\bar{\mathbf{x}} = \mathbf{J}^\top \bar{\mathbf{y}}$: the backward rule of a layer is simply the transpose of its linearization. The transpose of a linear map is also called its adjoint, which is where the name comes from. Written as $\bar{\mathbf{x}}^\top = \bar{\mathbf{y}}^\top \mathbf{J}$, this is just one step of the left-to-right order in Equation 34.36. A row vector times a Jacobian is called a *vector–Jacobian product*. The proposition lets us compute it for matrix variables without ever forming the Jacobian.

The single most frequent derivative in deep learning is that of a matrix product. Depending on whether we process a minibatch or a single vector, the weights stand on the right or the left of the data, so let us derive both layouts together. A minibatch $\mathbf{X} \in \mathbb{R}^{n \times d}$ with one example per row (the convention throughout the main text) multiplies the weight matrix $\mathbf{W} \in \mathbb{R}^{d \times q}$ to produce $\mathbf{Y} = \mathbf{X}\mathbf{W}$. The product rule gives $d\mathbf{Y} = d\mathbf{X}\,\mathbf{W} + \mathbf{X}\,d\mathbf{W}$, and the

move rule splits the inner product into one term per variable: $\langle \bar{\mathbf{Y}}, d\mathbf{X}\,\mathbf{W}\rangle + \langle \bar{\mathbf{Y}}, \mathbf{X}\,d\mathbf{W}\rangle = \langle \bar{\mathbf{Y}}\mathbf{W}^\top, d\mathbf{X}\rangle + \langle \mathbf{X}^\top\bar{\mathbf{Y}}, d\mathbf{W}\rangle$. Reading off both coefficients gives

$$\bar{\mathbf{X}} = \bar{\mathbf{Y}}\,\mathbf{W}^\top, \qquad \bar{\mathbf{W}} = \mathbf{X}^\top\bar{\mathbf{Y}}. \tag{34.39}$$

For a single example written as a column vector, $\mathbf{y} = \mathbf{W}\mathbf{x}$, the exact same calculation yields

$$\bar{\mathbf{W}} = \bar{\mathbf{y}}\,\mathbf{x}^\top, \qquad \bar{\mathbf{x}} = \mathbf{W}^\top\bar{\mathbf{y}},$$

matching the pattern of Equation 34.38: the weight gradient is the outer product of the output adjoint with the input. The two layouts are simply transposes of each other. If $\mathbf{x}_i^\top$ and $\bar{\mathbf{y}}_i^\top$ are the rows of $\mathbf{X}$ and $\bar{\mathbf{Y}}$, the minibatch formula $\mathbf{X}^\top\bar{\mathbf{Y}} = \sum_i \mathbf{x}_i\bar{\mathbf{y}}_i^\top$ sums one such outer product per example.

The backward pass of a linear layer therefore consists of two matrix products, each the same size as the forward product: the forward pass costs ndq multiplications and the backward pass costs $2ndq$. Checking shapes confirms both transposes: $\bar{\mathbf{Y}}\mathbf{W}^\top$ is $n \times d$ like $\mathbf{X}$, and $\mathbf{X}^\top\bar{\mathbf{Y}}$ is $d \times q$ like $\mathbf{W}$. Backward rules for more elaborate matrix operations (inverses, determinants, and decompositions) follow from the very same method (Mike B. Giles 2008).

Two more layers complete a multilayer perceptron. An elementwise activation $\mathbf{Y} = \varphi(\mathbf{X})$ has $d\mathbf{Y} = \varphi'(\mathbf{X}) \odot d\mathbf{X}$, so moving $\varphi'(\mathbf{X})$ across the inner product gives $\bar{\mathbf{X}} = \varphi'(\mathbf{X}) \odot \bar{\mathbf{Y}}$—the diagonal Jacobian from earlier, applied directly as a mask. A bias addition $\mathbf{y} = \mathbf{z} + \mathbf{b}$ has $d\mathbf{y} = d\mathbf{b}$, so $\bar{\mathbf{b}} = \bar{\mathbf{y}}$. For a minibatch, the bias is broadcast across rows via $\mathbf{Z} = \mathbf{X}\mathbf{W} + \mathbf{1}\mathbf{b}^\top$, which is just another matrix product and gives $\bar{\mathbf{b}} = \bar{\mathbf{Z}}^\top\mathbf{1}$, summing the adjoint over all examples. These three local rules are the building blocks from which Section 4.3 assembles the full backward pass of a multilayer perceptron.

They also let us derive the weight-gradient matrix from the opening. Running the network of Figure 34.21 backward from $\bar{g} = 1$, one layer at a time, at $\mathbf{x} = (1, 1)$:

1. The output $g = \mathbf{w}_2^\top\mathbf{h}$ gives $\bar{\mathbf{h}} = \mathbf{w}_2 = (2, -1)$ and $\bar{\mathbf{w}}_2 = \mathbf{h} = (3, 0)$.
2. The ReLU gives $\bar{\mathbf{a}} = \mathbf{m} \odot \bar{\mathbf{h}} = (2, 0)$, and the bias gives $\bar{\mathbf{b}} = \bar{\mathbf{a}} = (2, 0)$.
3. The affine map $\mathbf{a} = \mathbf{W}_1\mathbf{x} + \mathbf{b}$ gives $\bar{\mathbf{W}}_1 = \bar{\mathbf{a}}\,\mathbf{x}^\top = \left(\begin{smallmatrix} 2 & 2 \\ 0 & 0 \end{smallmatrix}\right)$ and $\bar{\mathbf{x}} = \mathbf{W}_1^\top\bar{\mathbf{a}} = (2, 4)$.

Notice that $\bar{\mathbf{x}}$ is the transpose of the Jacobian row $(2, 4)$ we found earlier, and $\bar{\mathbf{W}}_1$ is the matrix we obtained in the opening by nudging the weights. Differentiating `g` with respect to $\mathbf{W}_1$, $\mathbf{b}$, and $\mathbf{w}_2$ in code confirms all three arrays.

```
gW1, gb, gw2 = torch.func.grad(g, argnums=(1, 2, 3))(x, W1, b, w2)
h = torch.relu(W1 @ x + b)
show = lambda t: (t.numpy() + 0.0).tolist()     # + 0.0 turns -0.0 into 0.0
print('dg/dW1 = (w2*m) x^T:', show(torch.outer(w2 * m, x)),
      '  autograd:', show(gW1))
print('dg/db  = w2*m      :', show(w2 * m), '  autograd:', show(gb))
print('dg/dw2 = h         :', show(h), '  autograd:', show(gw2))
```

```
dg/dW1 = (w2*m) x^T: [[2.0, 2.0], [0.0, 0.0]]   autograd: [[2.0, 2.0], [0.0, 0.0]]
dg/db  = w2*m      : [2.0, 0.0]   autograd: [2.0, 0.0]
dg/dw2 = h         : [3.0, 0.0]   autograd: [3.0, 0.0]
```

It is worth noting where the zeros land. The inactive hidden unit leaves three zeros behind: a zero row in $\bar{\mathbf{W}}_1$, a zero in $\bar{\mathbf{b}}$, and a zero in $\bar{\mathbf{w}}_2$ (because its output $h_2 = 0$ is what $w_{2,2}$ multiplies). A training step on this input leaves all four of that unit's parameters untouched.

We still owe the numerical test that catches a misplaced transpose on a square factor. A test that cannot fail proves nothing, so we design this one to expose the mistake: every factor below is a random square (and hence nonsymmetric) matrix, and we print each correct rule beside a plausible wrong guess of the same shape. We test both the quadratic form and the matrix product. For the product, we differentiate the scalar $\langle \bar{\mathbf{Y}}, \mathbf{XW} \rangle$ for a random $\bar{\mathbf{Y}}$; by our backward-rule proposition, its gradients with respect to $\mathbf{X}$ and $\mathbf{W}$ are precisely $\bar{\mathbf{X}}$ and $\bar{\mathbf{W}}$.

```
A, x = torch.tensor(np.random.randn(3, 3)), torch.tensor(np.random.randn(3))
X, W, Ybar = (torch.tensor(np.random.randn(3, 3)) for _ in range(3))  # square
ok = lambda u, v: torch.allclose(u, v, atol=1e-5)
grad = torch.func.grad
gx = grad(lambda x: x @ A @ x)(x)
gX, gW = grad(lambda X, W: (Ybar * (X @ W)).sum(), argnums=(0, 1))(X, W)
print('grad x^T A x: (A + A^T) x', ok(gx, (A + A.T) @ x),
      '| wrong guess 2 A x       ', ok(gx, 2 * A @ x))
print('Xbar        : Ybar W^T   ', ok(gX, Ybar @ W.T),
      '| wrong guess Ybar W      ', ok(gX, Ybar @ W))
print('Wbar        : X^T Ybar   ', ok(gW, X.T @ Ybar),
      '| wrong guess (X^T Ybar)^T', ok(gW, (X.T @ Ybar).T))
```

```
grad x^T A x: (A + A^T) x True | wrong guess 2 A x        False
Xbar        : Ybar W^T    True | wrong guess Ybar W       False
Wbar        : X^T Ybar    True | wrong guess (X^T Ybar)^T False
```

The derived rules print `True` and the wrong guesses print `False`. Here the design of the test matters even more than its outcome. With rectangular factors, say $\mathbf{X}$ of shape 3×4 and $\mathbf{W}$ of shape 4×2, the two wrong guesses for the product would not even execute: a misplaced transpose would trigger a shape error before any number is compared, which is the shape rule doing its job. Likewise, with a symmetric $\mathbf{A}$, the wrong guess $2\mathbf{Ax}$ would have matched the true gradient and slipped past the test. The treacherous case is a square, nonsymmetric factor, so that is what a numerical check must draw.

The matrix-product rule handles losses in which both factors of a product are unknown just as easily, such as the matrix-factorization loss $\|\mathbf{X} - \mathbf{UV}\|_F^2$ in exercise 3. In every case, the backward rule multiplies the incoming adjoint by the transposes of whatever multiplied the variable on the forward pass. Notice also which forward values each rule requires: $\mathbf{X}$ and $\mathbf{W}$ for the matrix product, $\varphi'(\mathbf{X})$ for the activation, and nothing at all for the bias. A backward pass must retain those intermediate values from the forward pass, a memory cost that Section 34.5 measures directly.

34.4.3.3 Softmax and Cross-Entropy

The derivative a classifier evaluates most often is that of softmax composed with cross-entropy loss. Softmax converts a vector of scores $\mathbf{z}$, the *logits*, into a probability vector: setting $S = \sum_k e^{z_k}$, we have $p_i = e^{z_i}/S$. The same quotient rule that gave the log-sum-exp Hessian in Section 34.2 for two logits yields the softmax Jacobian for any number of classes. Writing $\delta_{ij} = 1$ if $i = j$ and 0 otherwise,

$$\frac{\partial p_i}{\partial z_j} = \frac{\delta_{ij} e^{z_i} S - e^{z_i} e^{z_j}}{S^2} = p_i(\delta_{ij} - p_j), \qquad \mathbf{J}_{\text{softmax}} = \operatorname{diag}(\mathbf{p}) - \mathbf{p}\mathbf{p}^\top. \tag{34.40}$$

In words, increasing logit i raises p_i at rate $p_i(1 - p_i)$ and lowers every other probability p_j at rate $p_i p_j$. As a quick check, $\mathbf{J}\mathbf{1} = \mathbf{p} - \mathbf{p}(\mathbf{p}^\top \mathbf{1}) = \mathbf{0}$ because the probabilities sum to one. This must hold, since adding the same constant to every logit leaves $\mathbf{p}$ unchanged; consequently, the softmax Jacobian is always singular.

Notice that $\mathbf{J}_{\text{softmax}}$ is also symmetric, a property an arbitrary Jacobian need not have. The reason is that it is actually a Hessian: softmax is the gradient of the log-sum-exp function $\operatorname{LSE}(\mathbf{z}) = \log \sum_k e^{z_k}$, our chapter's running example, because $\partial \operatorname{LSE} / \partial z_i = e^{z_i}/S = p_i$, and the Jacobian of a gradient is its Hessian. What Equation 34.40 adds to that two-logit formula is a clean statistical reading of the matrix. It is the covariance matrix of a one-hot vector drawn with probabilities $\mathbf{p}$, so for every $\mathbf{z}$ and every vector $\mathbf{v}$ the quadratic form $\mathbf{v}^\top \mathbf{J} \mathbf{v} = \sum_i p_i v_i^2 - \left(\sum_i p_i v_i\right)^2$ is the variance of the entries of $\mathbf{v}$ under the distribution $\mathbf{p}$. Because a variance is never negative, this Hessian is positive semidefinite everywhere, proving that log-sum-exp is convex. A constant vector $\mathbf{v}$ has zero variance, recovering the null vector $\mathbf{1}$ once again.

Now compose softmax with the cross-entropy loss $\ell = -\sum_i y_i \log p_i$ against a target distribution $\mathbf{y}$. The chain rule multiplies the row $\partial \ell / \partial \mathbf{p} = (-y_i/p_i)_i$ by Equation 34.40, and the algebra collapses dramatically:

$$\frac{\partial \ell}{\partial z_j} = -\sum_i \frac{y_i}{p_i} p_i(\delta_{ij} - p_j) = -y_j + p_j \sum_i y_i = p_j - y_j, \qquad \text{that is,} \qquad \nabla_{\mathbf{z}} \ell = \mathbf{p} - \mathbf{y}. \tag{34.41}$$

The gradient with respect to the logits is simply predicted probability minus target, the formula written as $\hat{\mathbf{y}} - \mathbf{y}$ in Section 3.1.2.2. Because the final step used only $\sum_i y_i = 1$, the result holds for any probability vector $\mathbf{y}$, not just one-hot labels—so label smoothing and distillation targets obey the exact same formula. For a target that does not sum to one, the gradient is instead $\left(\sum_i y_i\right)\mathbf{p} - \mathbf{y}$. At $\mathbf{z} = (1, -0.5, 2)$, for example, we have $\mathbf{p} = (0.254,\ 0.057,\ 0.690)$; with the second class as the target, the gradient is $(0.254,\ -0.943,\ 0.690)$.

In practice, libraries fuse softmax and cross-entropy into a single primitive for numerical stability (Section 3.3.5.2, Section 35.8.2). Below, we compute the softmax Jacobian at $\mathbf{z}$ by automatic differentiation and print its eigenvalues, which our variance argument predicts must be nonnegative with one zero. We then differentiate the fused loss for both the one-hot target above and the two-hot target $\mathbf{y} = (1, 1, 0)$, whose entries do not sum to one.

```
z = torch.tensor([1.0, -0.5, 2.0])
p = torch.softmax(z, 0)
J = torch.func.jacrev(lambda z: torch.softmax(z, 0))(z)   # the Jacobian, by autograd
print('eigenvalues of J:', torch.linalg.eigvalsh(J).numpy().round(3) + 0.0)
loss = lambda z, y: -(y * torch.log_softmax(z, 0)).sum()
for y in [torch.tensor([0.0, 1.0, 0.0]), torch.tensor([1.0, 1.0, 0.0])]:
    print(f'y = {y.numpy()}: d loss / d z = {torch.func.grad(loss)(z, y).numpy().round(4)}, '
          f'p - y = {(p - y).numpy().round(4)}')
```

```
eigenvalues of J: [0.    0.079 0.378]
y = [0. 1. 0.]: d loss / d z = [ 0.2537 -0.9434  0.6897], p - y = [ 0.2537 -0.9434  0.6897]
y = [1. 1. 0.]: d loss / d z = [-0.4926 -0.8868  1.3793], p - y = [-0.7463 -0.9434  0.6897]
```

The eigenvalues are 0, 0.079, and 0.378: one zero, corresponding to the direction $\mathbf{1}$, and two positive values. For the one-hot target, the gradient of the fused loss is $\mathbf{p} - \mathbf{y} = (0.2537, -0.9434, 0.6897)$, matching our hand calculation. For the two-hot target, however, it returns $(-0.4926, -0.8868, 1.3793)$, which is $2\mathbf{p} - \mathbf{y}$ rather than the $\mathbf{p} - \mathbf{y}$ printed beside it. "Prediction minus target" is a property of targets that sum to one, not of the loss alone.

Here we multiplied $\mathbf{J}_K \cdots \mathbf{J}_1$ by hand for a simple three-layer chain. In practice, a program must handle arbitrary computation graphs that branch and merge, retain the forward values that each backward rule needs, and compute all gradients at a cost that stays a small constant multiple of evaluating the network, no matter how many parameters it has. That program is automatic differentiation, the subject of Section 34.5.

34.4.4 Summary

The Jacobian of a map $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ is the $m \times n$ matrix of its best linear approximation, with entries $[\mathbf{J}_\mathbf{f}]_{ij} = \partial f_i / \partial x_j$; both the gradient (a single row, written as a column) and the Hessian (the Jacobian of the gradient) are special cases of it. A network's derivative with respect to its input is the product of its layers' Jacobians, $\mathbf{J}_K \cdots \mathbf{J}_1$, and the order of multiplication governs the cost: three million versus two billion multiplications for three layers of width 1000. The differential rule $df = \langle \nabla f, d\mathbf{x} \rangle$ derives every gradient a layer needs automatically in the shape of its variable, including the matrix-product backward rules $\bar{\mathbf{X}} = \bar{\mathbf{Y}}\mathbf{W}^\top$ and $\bar{\mathbf{W}} = \mathbf{X}^\top \bar{\mathbf{Y}}$, as well as the collapse of softmax with cross-entropy to $\mathbf{p} - \mathbf{y}$. While the shape rule and the 1×1 check catch most slips in a hand derivation, a misplaced transpose on a square factor escapes both and requires a numerical test on a nonsymmetric matrix.

34.4.5 Exercises

1. **Jacobian of a ReLU network.** Let $\mathbf{g}(\mathbf{x}) = \mathbf{W}_2 \, \varphi(\mathbf{W}_1 \mathbf{x} + \mathbf{b})$ with $\mathbf{W}_1 \in \mathbb{R}^{h \times n}$, $\mathbf{b} \in \mathbb{R}^h$, $\mathbf{W}_2 \in \mathbb{R}^{m \times h}$, and the elementwise ReLU φ. Assume that no entry of the pre-activation $\mathbf{W}_1 \mathbf{x} + \mathbf{b}$ is zero. By Equation 34.35, $\mathbf{J}_\mathbf{g}(\mathbf{x}) = \mathbf{W}_2 \mathbf{M}(\mathbf{x}) \mathbf{W}_1$, where the diagonal matrix $\mathbf{M}(\mathbf{x})$ has the entry 1 for each active hidden unit and 0 for each inactive one.
   1. Show that $\mathbf{g}(\mathbf{x} + \delta) = \mathbf{g}(\mathbf{x}) + \mathbf{J}_\mathbf{g}(\mathbf{x}) \, \delta$ holds exactly for every sufficiently small δ, so that the remainder in Equation 34.31 vanishes identically. For the 2-2-1 network of this section at $\mathbf{x} = (1, 1)$, find the largest r such that this holds for all δ with $\|\delta\|_\infty < r$. For $n = m = 2$, how does the image of a small circle under $\mathbf{g}$ differ from the one in Figure 34.20?
   2. Let k hidden units be active at $\mathbf{x}$. Show that $\operatorname{rank} \mathbf{J}_\mathbf{g}(\mathbf{x}) \le k$. For $n = m = 2$ and $k = 1$, what is the image of a small circle around $\mathbf{x}$?
   3. Describe the set of inputs at which some pre-activation is zero. What happens to $\mathbf{J}_\mathbf{g}$ when $\mathbf{x}$ crosses this set?
2. code **Cheapest chain order.** ● By Equation 34.36 the Jacobian of a chain of K maps is $\mathbf{J}_K \cdots \mathbf{J}_1$ with $\mathbf{J}_k \in \mathbb{R}^{d_k \times d_{k-1}}$. Multiplying a $p \times q$ matrix by a $q \times r$ matrix costs pqr multiplications.
   1. Give the costs of the forward order, which multiplies from the right, and of the reverse order, which multiplies from the left, as functions of $d_0, \ldots, d_K$. Evaluate

both for $(d_0, \dots, d_4) = (1000, 10, 1000, 10, 1000)$, a map whose layers alternate between wide and narrow, and compare with the section's count for $(1 \times h)(h \times h)(h \times h)(h \times n)$.

2. Write a dynamic program over the sub-chains $\mathbf{J}_j \cdots \mathbf{J}_i$ that finds the cheapest parenthesization, and run it on the dimensions of (a). Which order does it choose, and how much cheaper is it than the better of the two orders of (a)?
3. For a scalar output, $d_K = 1$, compare the optimum with the reverse order on a few thousand random dimension vectors, with K between 2 and 6 and each of $d_0, \dots, d_{K-1}$ drawn uniformly from $\{1, \dots, 10\}$. Characterize the vectors for which some other order is cheaper.

3. code **Backward rules of a matrix product.**
   1. For $\mathbf{Y} = \mathbf{X}\mathbf{W}$, derive Equation 34.39 by the differential method, then derive it again by indices to confirm the transposes.
   2. Verify both rules with the vector–Jacobian product `torch.func.vjp` on $\mathbf{X} \in \mathbb{R}^{3\times 4}$, $\mathbf{W} \in \mathbb{R}^{4\times 2}$ and a random adjoint $\bar{\mathbf{Y}}$. Count the multiplications of the forward product and of the two backward products.
   3. For the factorization loss $\|\mathbf{X} - \mathbf{U}\mathbf{V}\|_F^2$ with $\mathbf{R} = \mathbf{X} - \mathbf{U}\mathbf{V}$, check $\nabla_{\mathbf{U}} = -2\mathbf{R}\mathbf{V}^\top$ and $\nabla_{\mathbf{V}} = -2\mathbf{U}^\top\mathbf{R}$ numerically. Then write the alternating least-squares step that sets one gradient to zero with the other factor fixed, and show that each step is one linear solve.
4. **Softmax vector–Jacobian product.** A backward pass never forms the softmax Jacobian Equation 34.40. It maps the adjoint $\mathbf{u} = \bar{\mathbf{p}}$ of the probabilities to the adjoint $\bar{\mathbf{z}} = \mathbf{J}_{\text{softmax}}^\top \mathbf{u}$ of the logits.
   1. Show that $\mathbf{J}_{\text{softmax}}^\top \mathbf{u} = \mathbf{p} \odot \big(\mathbf{u} - (\mathbf{p}^\top \mathbf{u})\,\mathbf{1}\big)$, where $\odot$ is the elementwise product. For n classes, count the operations of this formula and those of forming $\mathbf{J}_{\text{softmax}}$ and multiplying it by $\mathbf{u}$.
   2. Self-attention applies softmax to each row of an $n \times n$ score matrix $\mathbf{S}$ and produces the weights $\mathbf{P}$. How many entries does the Jacobian of $\mathbf{P}$ with respect to $\mathbf{S}$ have, and how many of them can be nonzero? For $n = 4096$ tokens in float32, compare the memory of the full Jacobian, of its nonzero entries, and of the adjoint of $\mathbf{S}$ that the formula of (a), applied row by row, returns.
5. code **The LayerNorm Jacobian.** Layer normalization (Ba et al. 2016) maps $\mathbf{x} \in \mathbb{R}^n$ to $\hat{\mathbf{x}} = (\mathbf{x} - \mu\mathbf{1})/\sigma$ with $\mu = \frac{1}{n}\sum_i x_i$ and $\sigma^2 = \frac{1}{n}\sum_i (x_i - \mu)^2 > 0$ (no learned scale and no ϵ).
   1. Show by the differential method that $\mathbf{J} = \frac{1}{\sigma}\big(\mathbf{I} - \frac{1}{n}\mathbf{1}\mathbf{1}^\top - \frac{1}{n}\hat{\mathbf{x}}\hat{\mathbf{x}}^\top\big)$. *Hint:* $d\mu = \frac{1}{n}\mathbf{1}^\top d\mathbf{x}$ and $d\sigma = \frac{1}{n}\hat{\mathbf{x}}^\top d\mathbf{x}$.
   2. Show that $\mathbf{1}$ and $\hat{\mathbf{x}}$ are orthogonal and that $\|\hat{\mathbf{x}}\|^2 = n$. Conclude that $\sigma\mathbf{J}$ is the orthogonal projection onto the complement of $\operatorname{span}\{\mathbf{1}, \hat{\mathbf{x}}\}$ and that $\operatorname{rank}\mathbf{J} = n - 2$ for $n \geq 2$. Which two changes of the input can LayerNorm not see, and why?
   3. Check (a) and the two null vectors against the Jacobian computed by automatic differentiation for a random $\mathbf{x} \in \mathbb{R}^5$, and print the eigenvalues of $\mathbf{J}$.
6. **Log-determinant gradient.** ● A normalizing flow (Rezende and Mohamed 2015) passes a random vector $\mathbf{X}$ through invertible layers. For one layer $\mathbf{Y} = \mathbf{g}(\mathbf{X})$, the log-density of $\mathbf{Y}$ at $\mathbf{g}(\mathbf{x})$ is $\log p_{\mathbf{X}}(\mathbf{x}) - \log|\det \mathbf{J}_{\mathbf{g}}(\mathbf{x})|$, by the change-of-variables formula for densities. For a linear layer $\mathbf{x} \mapsto \mathbf{A}\mathbf{x}$ the correction is $-\log|\det \mathbf{A}|$, so maximum-likelihood training differentiates $\log|\det \mathbf{A}|$ with respect to $\mathbf{A}$.
   1. Expand $\det(\mathbf{I} + t\mathbf{M})$ to first order in t and use $\det(\mathbf{A} + d\mathbf{A}) = \det\mathbf{A}\,\det(\mathbf{I} + \mathbf{A}^{-1}d\mathbf{A})$ to derive Jacobi's formula $d\det\mathbf{A} = \det\mathbf{A}\,\operatorname{tr}(\mathbf{A}^{-1}d\mathbf{A})$ for invertible $\mathbf{A}$. Conclude with Equation 34.37 that $\nabla_{\mathbf{A}} \log|\det\mathbf{A}| = \mathbf{A}^{-\top}$. Why does the sign of $\det\mathbf{A}$ not

matter?
2. Check the identity on a random invertible 3×3 matrix by central differences of `np.linalg.slogdet`.
3. How does the norm of the gradient behave as $\mathbf{A}$ approaches a singular matrix? What does this imply for a flow layer whose weight matrix is nearly singular?

7. **Gradient of the spectral norm.** ● Spectral normalization (Miyato et al. 2018) divides each weight matrix $\mathbf{W}$ of a network by its largest singular value $\sigma_1(\mathbf{W})$, so training differentiates σ_1.
1. Let $\mathbf{S}$ be a real symmetric matrix with a simple eigenvalue λ and unit eigenvector $\mathbf{q}$, and perturb it to $\mathbf{S} + t\mathbf{E}$ with $\mathbf{E}$ symmetric. Take as given that $\lambda(t)$ and $\mathbf{q}(t)$, defined by $(\mathbf{S} + t\mathbf{E})\mathbf{q} = \lambda\mathbf{q}$ and $\mathbf{q}^\top\mathbf{q} = 1$, are differentiable near $t = 0$ (the implicit function theorem gives this because λ is simple). Differentiate both equations at $t = 0$ to show the Hellmann–Feynman theorem $\lambda'(0) = \mathbf{q}^\top\mathbf{E}\,\mathbf{q}$, and give $\nabla_{\mathbf{S}}\lambda$ for the inner product $\operatorname{tr}(\mathbf{A}^\top\mathbf{B})$.
2. Apply (a) to $\sigma_1(\mathbf{W})^2 = \lambda_{\max}(\mathbf{W}^\top\mathbf{W})$ to show that $\nabla_{\mathbf{W}}\sigma_1 = \mathbf{u}_1\mathbf{v}_1^\top$ when $\sigma_1 > 0$ is simple, where $\mathbf{u}_1$ and $\mathbf{v}_1$ are the top left and right singular vectors. Why is $\sigma_1 > 0$ needed? Consider $\mathbf{W} = \mathbf{w} \in \mathbb{R}^{n\times 1}$ near $\mathbf{w} = \mathbf{0}$. Check the formula by central differences on a random 5×3 matrix in NumPy.
3. Compute $\sigma_1(\mathbf{I} + t\,\operatorname{diag}(1, -1))$ for small t of both signs. Is σ_1 differentiable at the 2×2 identity, and which hypothesis of (a) fails there?

Adapted from Alan Edelman and Steven G. Johnson, Matrix Calculus for Machine Learning and Beyond, problem set 2, MIT 18.S096, IAP 2023, Problem 3.

34.5 Automatic Differentiation

Throughout the Basics part of this book, every gradient came from a single call to the framework's automatic differentiation engine. The previous section showed what that call computes mathematically: the derivative of a network is a product of Jacobians, one per layer, and the order in which we multiply those matrices determines the cost. A real program, however, is not a neat list of matrices; it is a sequence of additions, multiplications, and function calls. How does a framework extract those factors from ordinary code, and how does it multiply them in the cheap order?

How much the cheap order saves becomes unmistakable on a modern network. Consider a model with 10^8 parameters that outputs a single scalar loss ℓ. Every training step needs the derivative of ℓ with respect to all 10^8 parameters. The direct finite-difference approach nudges one parameter at a time, reruns the network, and divides the change in ℓ by the nudge. That requires 10^8 runs per step. At 10 ms per run, computing a single gradient would take 10^6 s, nearly twelve days, and the choice of step size would still introduce truncation and rounding errors. Backpropagation returns all 10^8 derivatives, exact up to floating-point rounding, for roughly the cost of three forward runs, about 30 ms. This section explains where that factor of 10^8 goes.

The core idea is surprisingly simple: a product of Jacobians can be evaluated from either end. Starting from the input end, each pass propagates one input direction forward, so 10^8 inputs demand 10^8 passes. Starting from the loss end, each pass propagates one output direction backward, and a scalar loss has only a single output.

In Section 1.5, we saw how to invoke both modes in a deep learning framework. Here we open the hood. We will build forward mode and reverse mode from scratch in a few dozen lines of Python, prove that they return true derivatives, trace a backward pass by hand, and

measure the cost of each mode. We will then compose the two modes to compute second-order curvature along a direction and to differentiate the solution of an equation without differentiating through the solver itself. Two takeaways stand above the rest: evaluating a program on *dual numbers* yields derivatives automatically (the section's one proposition), and every sweep costs only a small multiple of one program evaluation, with forward mode needing one sweep per input direction and reverse mode one per output direction. Readers on a first pass may save Section 34.5.3 for a second reading.

Three small running examples will anchor the discussion: the chapter's log-sum-exp $\ell(\mathbf{x}) = \log \sum_j e^{x_j}$, whose gradient is the softmax; the plane map $\mathbf{f}(x, y) = (x + \sin y,\ y + x^2/2)$ from the previous section; and the polynomial expression $y = (uv + u)^2$, which is small enough to trace completely by hand.

We use the following imports throughout this section.

```
import numpy as np
import torch
```

34.5.1 Forward Mode and Dual Numbers

34.5.1.1 Dual Numbers

We want both the value of a program and its derivative from a single run, with no step size to tune and no symbolic formula for f' written down in advance. The trick is a remarkably small extension of the real numbers. We adjoin a new symbol ε that is nonzero, yet whose square vanishes:

$$\varepsilon^2 = 0, \qquad \varepsilon \neq 0. \tag{34.42}$$

With this single rule in place, we compute with expressions of the form $a + b\varepsilon$, called *dual numbers*, using the ordinary rules of algebra. Every differentiation rule in Section 34.1 works by discarding second-order terms, and setting $\varepsilon^2 = 0$ builds that discard directly into the arithmetic.

Let us see what happens when we apply this rule to $3 + \varepsilon$. Squaring and cubing it yields

$$(3 + \varepsilon)^2 = 9 + 6\varepsilon + \varepsilon^2 = 9 + 6\varepsilon, \qquad (3 + \varepsilon)^3 = (9 + 6\varepsilon)(3 + \varepsilon) = 27 + 27\varepsilon.$$

Look at the coefficients of ε: 6 and 27 are the derivatives of x^2 and x^3 at $x = 3$. We did not invoke any differentiation rule; the derivatives fell straight out of multiplication. The same holds for any product. Multiplying two dual numbers and dropping the ε^2 term gives

$$(a + b\varepsilon)(c + d\varepsilon) = ac + (ad + bc)\,\varepsilon. \tag{34.43}$$

If b and d are the derivatives of two quantities whose values are a and c, then $ad + bc$ is the derivative of their product. Multiplication of dual numbers simply is the product rule.

A skeptical reader may object that no number can be nonzero and still have a zero square. No real number can, of course, but a matrix easily can. Take $\varepsilon = \left(\begin{smallmatrix} 0 & 1 \\ 0 & 0 \end{smallmatrix}\right)$. Then $\varepsilon^2 = \mathbf{0}$, the dual number $a + b\varepsilon$ becomes the 2×2 matrix $\left(\begin{smallmatrix} a & b \\ 0 & a \end{smallmatrix}\right)$, and ordinary matrix multiplication

$\left(\begin{smallmatrix} a & b \\ 0 & a \end{smallmatrix}\right)\left(\begin{smallmatrix} c & d \\ 0 & c \end{smallmatrix}\right) = \left(\begin{smallmatrix} ac & ad+bc \\ 0 & ac \end{smallmatrix}\right)$ reproduces Equation 34.43. Dual numbers are just as concrete as 2×2 matrices, much as complex numbers are as concrete as rotations and scalings of the plane.

Real programs also call elementary functions such as sin, exp, and log. For a function φ given by its Taylor series around a, substituting $a + b\varepsilon$ into the series kills every term from $(b\varepsilon)^2$ onward, leaving only the linear approximation:

$$\varphi(a + b\varepsilon) = \varphi(a) + \varphi'(a)\, b\, \varepsilon.$$

Thus $e^{a+b\varepsilon} = e^a + e^a b\, \varepsilon$ and $\sin(a+b\varepsilon) = \sin a + (\cos a)\, b\, \varepsilon$ are forced by the algebra itself. For a primitive that is merely differentiable at a rather than analytic, we adopt the formula above as the definition of its action on dual numbers. In either case, the ε-part obeys the chain rule: the incoming derivative b of the argument is multiplied by the local derivative $\varphi'(a)$.

Proposition (dual numbers compute derivatives). *Let f be an expression built from constants, the identity, sums, products, and primitives φ (such as* sin*,* exp*,* log*). Suppose each primitive is differentiable at the argument it receives when f is evaluated at x (for* log*, a positive argument). Then evaluating the same expression on the dual number $x + \varepsilon$ gives*

$$f(x + \varepsilon) = f(x) + f'(x)\, \varepsilon. \tag{34.44}$$

Proof. We proceed by structural induction on the expression. A constant c is represented as $c + 0\varepsilon$ and the identity as $x + 1\varepsilon$, so their ε-parts are their derivatives, 0 and 1. Now suppose two subexpressions satisfy $g(x + \varepsilon) = g(x) + g'(x)\varepsilon$ and $h(x + \varepsilon) = h(x) + h'(x)\varepsilon$. Their sum has ε-part $g' + h' = (g + h)'$. By Equation 34.43, their product has ε-part $g'h + gh' = (gh)'$. Finally, applying a primitive φ to g produces, by the rule above with $a = g(x)$ and $b = g'(x)$, the ε-part $\varphi'(g(x))\, g'(x) = (\varphi \circ g)'(x)$. Every way of combining subexpressions preserves the property, so the full expression satisfies Equation 34.44. ■

To differentiate a program, we therefore run it on $x + \varepsilon$ and read off the ε-part of the output. Programs with loops and branches are handled automatically as well: for any fixed input, the program executes a single concrete sequence of elementary operations, which is an expression of this exact form, and the dual-number run differentiates that execution path (Section 1.5.3.1 demonstrates this in the frameworks). Note, however, that the result is the derivative of the executed path, which equals $f'(x)$ only if all nearby inputs take the same branches. A program that returns `x` if `x != 0` and `0.0` otherwise computes the identity, yet at $x = 0$ it executes the constant branch, and the dual-number run reports slope 0. This is the failure we met with $\mathrm{ReLU}(x) - \mathrm{ReLU}(-x)$ in Section 34.1.6.3. The input's ε-part, here 1, is called the *seed*. No symbolic expression for f' is ever constructed and no step size is chosen; the only error is floating-point rounding.

34.5.1.2 A Forward-Mode Engine

Operator overloading turns this algebra directly into code. The `Dual` class below stores a value a together with its derivative part b, which we call the *tangent* and write as $\dot{v}$ for a quantity v. It implements Equation 34.43 for `*` and equips `sin`, `exp`, and `log` with the primitive rule. A class counter `ops` tallies how many primitive operations have executed, which we will use to measure cost in Section 34.5.2.4.

```
class Dual:
    """A dual number a + b*eps with eps^2 = 0; b carries the derivative."""
    ops = 0                                          # primitive operations run
    def __init__(self, a, b=0.0):
        self.a, self.b = a, b                        # value, tangent
    def __add__(self, o):
        o = o if isinstance(o, Dual) else Dual(o)
        Dual.ops += 1
        return Dual(self.a + o.a, self.b + o.b)
    __radd__ = __add__
    def __mul__(self, o):
        o = o if isinstance(o, Dual) else Dual(o)
        Dual.ops += 1
        return Dual(self.a * o.a, self.a * o.b + self.b * o.a)   # product rule
    __rmul__ = __mul__
    def sin(self):
        Dual.ops += 1
        return Dual(np.sin(self.a), np.cos(self.a) * self.b)     # chain rule
    def exp(self):
        Dual.ops += 1
        return Dual(np.exp(self.a), np.exp(self.a) * self.b)
    def log(self):
        Dual.ops += 1
        return Dual(np.log(self.a), self.b / self.a)
    def __repr__(self):
        return f'{np.round(self.a, 6)} + {np.round(self.b, 6)} eps'
```

Let us test `Dual` on $f(x) = \sin(x^2) + e^x$ at $x = 1.3$ by passing in $1.3 + \varepsilon$, and compare its ε-part against the hand-derived derivative $f'(x) = 2x\cos(x^2) + e^x$:

```
def f(x):
    return (x * x).sin() + x.exp()                   # sin(x^2) + e^x, in dual algebra

x0 = 1.3
out = f(Dual(x0, 1.0))                               # seed the derivative with 1
exact = 2 * x0 * np.cos(x0**2) + np.exp(x0)          # f'(x) by hand
print('dual result :', out)
print('value f(x)  :', np.sin(x0**2) + np.exp(x0))
print("derivative  : dual %.6f  vs  exact %.6f" % (out.b, exact))
```

```
dual result : 4.6622 + 3.360101 eps
value f(x)  : 4.662200318713363
derivative  : dual 3.360101  vs  exact 3.360101
```

A single run returns both the function value 4.6622 and the derivative 3.360101, matching the analytic formula to every printed digit.

A single-variable example, however, hides the main limitation of forward mode. Consider log-sum-exp at the three-dimensional input $\mathbf{x} = (1, -0.5, 2)$, whose gradient is the softmax $\partial \ell / \partial x_j = e^{x_j} / \sum_k e^{x_k} = p_j$. How many passes does forward mode need to obtain this full gradient? Seeding input j with 1 and all other inputs with 0 yields the single partial derivative $\partial \ell / \partial x_j$ in the output's ε-part, and nothing else.

```
def lse_dual(xs):                                   # log-sum-exp, in dual algebra
    total = 0.0
    for x in xs:
        total = total + x.exp()
    return total.log()

x_lse = [1.0, -0.5, 2.0]
p = np.exp(x_lse) / np.exp(x_lse).sum()             # softmax, the known gradient
for j in range(3):                                  # one pass per input direction
    out = lse_dual([Dual(x, 1.0 if k == j else 0.0) for k, x in enumerate(x_lse)])
    print(f'seed e{j + 1}: value {out.a:.6f}, partial {out.b:.6f}, '
          f'softmax {p[j]:.6f}')
```

```
seed e1: value 2.371539, partial 0.253716, softmax 0.253716
seed e2: value 2.371539, partial 0.056612, softmax 0.056612
seed e3: value 2.371539, partial 0.689672, softmax 0.689672
```

It takes three passes. Each pass recomputes the same value 2.371539 and recovers just one softmax entry: 0.253716, 0.056612, or 0.689672. More generally, a gradient over n inputs requires n passes. For the 10^8-parameter network from the opening, that puts us right back at 10^8 runs of the program. Forward mode eliminates the finite-difference step size, but not the number of runs.

34.5.1.3 Vector Tangents and Jacobian Columns

Now consider a vector-valued map $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ with $m \times n$ Jacobian $\mathbf{J}$. What does a single forward pass compute? Suppose we seed the inputs with an arbitrary direction $\mathbf{v}$, setting input j to $x_j + v_j\varepsilon$. Applying our proposition to the one-dimensional slice $t \mapsto f_i(\mathbf{x} + t\mathbf{v})$ shows that output i emerges as $f_i(\mathbf{x}) + (\nabla f_i \cdot \mathbf{v})\,\varepsilon$. Collecting the ε-parts across all m outputs therefore produces the vector $\mathbf{Jv}$, known as the *Jacobian–vector product* (JVP). When $\mathbf{v}$ is the j-th standard basis vector $\mathbf{e}_j$, the JVP picks out column j of $\mathbf{J}$. One forward pass carries one input direction through the entire map.

Let us verify this on the plane map $\mathbf{f}(x, y) = (x + \sin y,\ y + x^2/2)$ at $(0.5, 0.5)$. Its Jacobian, whose row i holds the partial derivatives of output i, is

$$\mathbf{J} = \begin{pmatrix} 1 & \cos y \\ x & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0.877583 \\ 0.5 & 1 \end{pmatrix}.$$

Two forward passes, seeded first with $\mathbf{e}_1$ and then with $\mathbf{e}_2$, recover its two columns.

```
def plane_dual(x, y):
    return [x + y.sin(), y + 0.5 * (x * x)]         # (x + sin y, y + x^2/2)

z0 = (0.5, 0.5)
cols = []
for seed in ((1.0, 0.0), (0.0, 1.0)):               # e1, then e2
    out = plane_dual(Dual(z0[0], seed[0]), Dual(z0[1], seed[1]))
    cols.append([out[0].b, out[1].b])               # tangent = J @ seed, a column
J_fwd = np.array(cols).T                            # columns side by side
J_exact = np.array([[1.0, np.cos(0.5)], [0.5, 1.0]])
```

```
    print('forward-mode J:\n', J_fwd.round(6))
    print('matches the analytic Jacobian:', np.allclose(J_fwd, J_exact))
```

```
forward-mode J:
 [[1.       0.877583]
 [0.5      1.      ]]
matches the analytic Jacobian: True
```

Notice that nothing in `Dual` forces the tangent to be a scalar. If we give x the vector tangent $(1, 0)$ and y the vector tangent $(0, 1)$, every operation propagates both input directions simultaneously, and the tangent of output i becomes row i of $\mathbf{J}$. Can a single pass recover the entire matrix?

```
    out = plane_dual(Dual(z0[0], np.array([1.0, 0.0])),     # identity seed:
                     Dual(z0[1], np.array([0.0, 1.0])))     # both directions at once
    J_one = np.array([out[0].b, out[1].b])                  # row i = tangent of f_i
    print('one-pass J:\n', J_one.round(6))
    print('matches the analytic Jacobian:', np.allclose(J_one, J_exact))
```

```
one-pass J:
 [[1.       0.877583]
 [0.5      1.      ]]
matches the analytic Jacobian: True
```

It can, though the savings are illusory. Every tangent operation now acts on arrays of length n, so that single pass performs roughly n times as much arithmetic as a scalar pass. Counting passes and counting arithmetic are not the same thing, and we will measure arithmetic directly in Section 34.5.2.4. For a scalar loss with 10^8 inputs, forward mode demands either 10^8 scalar passes or one pass carrying tangents of length 10^8. Either way, its total work scales with n runs of the program.

34.5.2 Reverse Mode and the Tape

34.5.2.1 Two Orders of One Product

To eliminate the factor of n, we must multiply the chain of derivatives in the opposite order. In Section 34.4, we saw that the Jacobian of a K-step program $\mathbf{f} = \mathbf{f}_K \circ \cdots \circ \mathbf{f}_1$ is the product of the step Jacobians,

$$\mathbf{J} = \mathbf{J}_K \, \mathbf{J}_{K-1} \cdots \mathbf{J}_1, \tag{34.45}$$

and that the parenthesization of this product governs its cost: for three layers of width 1000 multiplied by a row vector on the left, associating from the left takes 3×10^6 multiplications, whereas associating from the right takes 2×10^9 (Figure 34.22). An automatic differentiation engine never materializes these Jacobian matrices explicitly. Instead, it propagates a seed vector through one step after another. Forward mode multiplies a column direction $\mathbf{v}$ in from the right; reverse mode multiplies a row vector $\mathbf{u}^\top$ in from the left:

$$\mathbf{J}\mathbf{v} = \mathbf{J}_K\Big(\cdots\big(\mathbf{J}_2(\mathbf{J}_1\mathbf{v})\big)\Big), \qquad \mathbf{u}^\top\mathbf{J} = \Big(\big((\mathbf{u}^\top\mathbf{J}_K)\mathbf{J}_{K-1}\big)\cdots\Big)\mathbf{J}_1. \tag{34.46}$$

Because a seed vector is present from the start, every step is a matrix–vector product and never a matrix–matrix product. On the left, an input direction travels forward to produce the JVP $\mathbf{Jv}$. On the right, an output direction travels backward to produce the *vector–Jacobian product* (VJP) $\mathbf{u}^\top \mathbf{J}$, a linear combination of the rows of $\mathbf{J}$ weighted by $\mathbf{u}$. Setting $\mathbf{u} = \mathbf{e}_i$ extracts row i of $\mathbf{J}$.

From this symmetry comes the fundamental cost rule of automatic differentiation. Forward mode requires one sweep per input direction, while reverse mode requires one sweep per output direction, and each sweep costs only a small constant multiple of evaluating the program once. For programs built from $+, -, \times, \div$, Baur and Strassen proved that computing all partial derivatives together takes at most a fixed multiple of the program's own operation count, between three and five depending on which operations are counted (Baur and Strassen 1983). A scalar loss has a $1 \times n$ Jacobian, a single row, so one reverse sweep seeded with 1 returns the entire gradient at once. Figure 34.23 illustrates this rule in terms of matrix shapes.

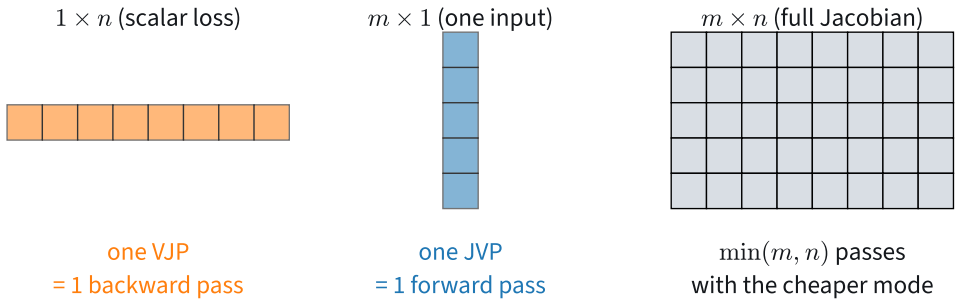


Figure 34.23: The shape of the Jacobian decides the cheaper mode. A scalar loss has a $1 \times n$ Jacobian: one row, one VJP, one backward pass. A map with one input has an $m \times 1$ Jacobian: one column, one JVP, one forward pass. A full $m \times n$ Jacobian costs $\min(m, n)$ passes with the cheaper mode.

The two modes evaluate the exact same matrix product from opposite ends. Which end wins depends solely on the shape of $\mathbf{J}$: the cheaper mode is the one whose number of seeds matches the shorter side. A neural network loss, with one scalar output and millions or billions of parameters, has the most lopsided Jacobian imaginable.

34.5.2.2 The Tape

Forward mode and reverse mode propagate fundamentally different quantities. In a program taking input x to output y, the tangent $\dot{v}$ of an intermediate variable v measures how fast v changes when the input moves in the direction $\dot{x}$. By contrast, the *adjoint* $\bar{v}$ introduced in Section 34.3 is the total derivative of the final output y with respect to v, summed over all paths from v to y. A tangent depends only on the steps *before* v, whereas an adjoint depends on the steps *after* v. In $y = v^2$ with $v = 3x$, for example, we have $\dot{v} = 3\dot{x}$ coming from below and $\bar{v} = 2v$ coming from above.

This asymmetry dictates how each mode executes (Figure 34.24). Because tangents move in the same direction as the values, forward mode finishes in a single pass alongside the program itself; `Dual` never needs to remember past intermediates. Adjoints, however, travel

against the flow of execution: computing the adjoint of a node requires the adjoints of every node that consumes it, and those exist only after the program has reached its output. Reverse mode therefore runs in two phases. First it executes the program forward while recording every elementary operation; then it replays that record in reverse. The record is called the *tape*, or Wengert list, after the list of elementary operations that Wengert (1964) introduced for forward mode; reversing the list yields reverse mode, introduced by Linnainmaa (1970) .

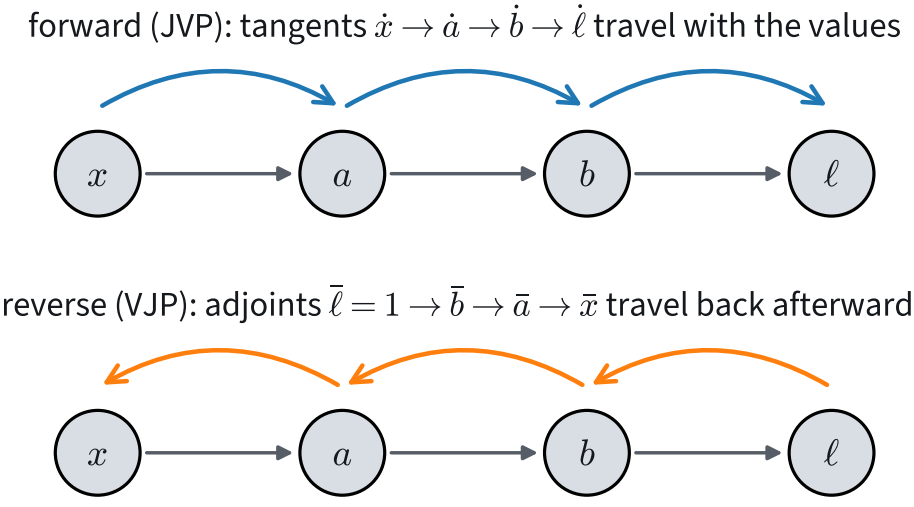


Figure 34.24: Forward and reverse mode on a chain $x \to a \to b \to \ell$. Both evaluate the chain from left to right (gray). Forward mode (top) carries the tangents $\dot{v}$ along with the values, one input direction per sweep. Reverse mode (bottom) runs the chain first and then carries the adjoints $\bar{v}$, the total derivatives of ℓ with respect to v, back from the seed $\bar{\ell} = 1$, one output direction per sweep, which for a scalar loss is the whole gradient.

We can build a working reverse-mode engine in about forty lines of Python. Each node on the tape stores its forward value, its parent nodes, and a small backward closure. Given the node's own adjoint g, that closure adds the node's contribution to each parent's adjoint: for $c = a + b$ it adds g to both parents, while for $c = ab$ it adds $b\,g$ to $\bar{a}$ and $a\,g$ to $\bar{b}$. The function `backprop` topologically sorts the graph so that every node appears after its parents, seeds the output adjoint with 1, and invokes the closures in reverse order. Just like `Dual`, the class maintains an `ops` counter across both forward and backward steps.

```
class Var:
    """A scalar node on the autodiff tape; .grad accumulates the adjoint."""
    ops = 0                                          # recorded ops + backward steps
    def __init__(self, value, parents=(), backward=lambda g: None):
        self.value = value
        self.grad = 0.0
        self._parents = parents                      # nodes this one depends on
        self._backward = backward                    # pushes adjoint to parents
```

```
    def __add__(self, o):
        o = o if isinstance(o, Var) else Var(o)
        Var.ops += 1
        out = Var(self.value + o.value, (self, o))
        def back(g):                                  # d(a+b) = 1*g to each parent
            Var.ops += 1
            self.grad += g; o.grad += g
        out._backward = back
        return out
    __radd__ = __add__
    def __mul__(self, o):
        o = o if isinstance(o, Var) else Var(o)
        Var.ops += 1
        out = Var(self.value * o.value, (self, o))
        def back(g):                                  # product rule, reversed
            Var.ops += 1
            self.grad += o.value * g; o.grad += self.value * g
        out._backward = back
        return out
    __rmul__ = __mul__

def backprop(node):
    """Topologically sort the tape and replay it backward from `node`."""
    order, seen = [], set()
    def visit(v):
        if v not in seen:
            seen.add(v)
            for p in v._parents:
                visit(p)
            order.append(v)
    visit(node)
    node.grad = 1.0                                   # seed the output adjoint
    for v in reversed(order):
        v._backward(v.grad)
```

Although `Var` implements only `+` and `*`, that is already enough for our third running example, $y = (uv + u)^2$. Before running the code, let us trace the entire backward sweep by hand at $u = 2$, $v = -3$. The forward pass records three elementary operations: $t = uv = -6$, then $r = t + u = -4$, and finally $y = r \cdot r = 16$. The backward sweep visits nodes in a reverse topological order such as y, r, t, u, v, ensuring that every node is processed only after all nodes that depend on it (`backprop` happens to visit v before u, which is equally valid). Each adjoint is the sum of contributions arriving along its outgoing edges, where each contribution multiplies the consumer's adjoint by the local derivative:

node	outgoing edges and their contributions	adjoint
y	none: the seed	$\bar{y} = 1$
r	both arguments of $y = r \cdot r$: $\bar{y}\, r + \bar{y}\, r$	$\bar{r} = -8$
t	$r = t + u$: $\bar{r} \cdot 1$	$\bar{t} = -8$
u	$r = t + u$ and $t = uv$: $\bar{r} \cdot 1 + \bar{t} \cdot v$	$\bar{u} = -8 + 24 = 16$

node	outgoing edges and their contributions	adjoint
v	$t = uv$: $\bar{t} \cdot u$	$\bar{v} = -16$

Direct differentiation of $y = (uv+u)^2$ confirms both answers: $\partial y/\partial u = 2(uv+u)(v+1) = 2(-4)(-2) = 16$ and $\partial y/\partial v = 2(uv+u)\,u = 2(-4)(2) = -16$. Notice how crucial accumulation is at two places in the graph: r feeds both arguments of the final product, and u feeds both t and r (Figure 34.25). Overwriting instead of adding would leave $\bar{r} = -4$, and even with $\bar{r}$ fixed it would leave $\bar{u}$ as either -8 or 24, depending on which closure happened to run last. Because an adjoint sums over all outgoing edges (the chain rule's sum over paths), every backward closure must accumulate with `+=`.

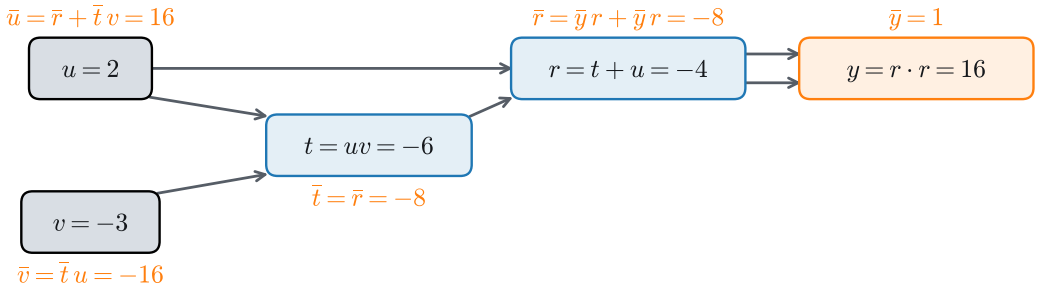


Figure 34.25: The tape of $y = (uv + u)^2$ at $u = 2$, $v = -3$. Each box shows the forward value of its node; the orange label shows its adjoint, the derivative of y with respect to that node. The graph is a diamond, not a chain: u feeds both $t = uv$ and $r = t + u$, and r feeds both arguments of $y = r \cdot r$, so the adjoints of r and u are sums of two contributions.

Recording the same expression with `Var` and running a single backward sweep gives:

```
u, v = Var(2.0), Var(-3.0)
r = u * v + u                                   # t = uv, then r = t + u
y = r * r                                       # r feeds both arguments
backprop(y)
print('our tape  : dy/du = %.4f  dy/dv = %.4f' % (u.grad, v.grad))
```

```
our tape  : dy/du = 16.0000  dy/dv = -16.0000
```

These match $\bar{u}$ and $\bar{v}$ from our hand trace. A production deep learning framework runs the exact same algorithm at tensor scale and produces the same two derivatives:

```
ut = torch.tensor(2.0, requires_grad=True)
vt = torch.tensor(-3.0, requires_grad=True)
rt = ut * vt + ut
(rt * rt).backward()                            # framework autograd
print('torch     : dy/du = %.4f  dy/dv = %.4f' % (ut.grad, vt.grad))
```

```
torch     : dy/du = 16.0000  dy/dv = -16.0000
```

A production framework differs from our toy `Var` engine in three practical respects, none of which alters the underlying algorithm. First, its nodes hold entire tensors rather than scalars, so each backward closure implements the VJP of a whole tensor operation, such as $\bar{\mathbf{X}} = \bar{\mathbf{Y}}\mathbf{W}^\top$ and $\bar{\mathbf{W}} = \mathbf{X}^\top\bar{\mathbf{Y}}$ (Section 34.4.3.2) for matrix multiplication. Second, its library of primitives is much larger, each paired with a hand-tuned VJP rule. Third, frameworks construct the computation graph in one of two ways. An *eager tape* records operations dynamically as the program executes, just as `Var` does. A *tracer*, by contrast, runs the program once on abstract placeholder inputs to capture its sequence of operations, builds the linearized forward-mode rule for each step, and then transposes and reverses the sequence. Both routes produce the exact same backward sweep. Applied to a neural network, this reverse-mode sweep is what deep learning calls *backpropagation* (Rumelhart et al. 1986).

PyTorch records eagerly: every operation on a tensor with `requires_grad=True` appends a node to the graph, and `backward()` replays those nodes in reverse.

The frameworks also inherit the hypothesis of the dual-number proposition: each primitive must be differentiable at the argument it receives. Masking is where this hypothesis fails most often in practice. Suppose that we sum $\log x_i$ over the real entries of a padded vector, and that a `where` sets the padded entries to zero. A `where` chooses between two values that have both been computed, so log still runs on the padding, here $x_2 = 0$. The forward pass is unharmed: the $-\infty$ is discarded and the loss is $\log 2 \approx 0.6931$. On the backward sweep, however, `where` sends the adjoint 0 into the discarded branch, and the VJP of log multiplies it by the local derivative $1/0 = \infty$. In floating-point arithmetic $0 \cdot \infty$ is not 0 but NaN.

```
x = torch.tensor([2.0, 0.0], requires_grad=True)    # x[1] is padding
mask = torch.tensor([True, False])
loss = torch.where(mask, torch.log(x), 0.0).sum()   # sum of log x, real entries
loss.backward()
print('one where   : loss = %.4f, grad =' % loss.item(), x.grad.numpy())
x.grad = None
safe = torch.where(mask, x, 1.0)                    # padding never reaches log
loss = torch.where(mask, torch.log(safe), 0.0).sum()
loss.backward()
print('double where: loss = %.4f, grad =' % loss.item(), x.grad.numpy())
```

```
one where   : loss = 0.6931, grad = [0.5 nan]
double where: loss = 0.6931, grad = [0.5 0. ]
```

The first line reports the gradient $(0.5, \text{NaN})$, although the loss does not depend on the padded entry at all and the true derivative there is 0. A single NaN is not a local blemish: every parameter upstream of x_2 receives it. The second line shows the standard remedy, the *double where*. An inner `where` replaces the masked entries by a harmless value, here 1, before the primitive sees them, so that no infinite local derivative is ever formed and the gradient is $(0.5, 0)$. Clamping the argument away from the edge of the domain serves the same purpose. The same trap awaits $\sqrt{x}$ at 0, and with it a norm computed as $\sqrt{\sum_i x_i^2}$ at $\mathbf{x} = \mathbf{0}$.

34.5.2.3 Parameter Gradients

So far, the Jacobian product Equation 34.45 has tracked derivatives with respect to the network's input, whereas training needs derivatives with respect to its weights. Fortunately,

the weight gradients drop out of the very same backward sweep via the layer VJP rule (Section 34.4.3.2). For a linear layer $\mathbf{h}_k = \mathbf{W}_k \mathbf{h}_{k-1}$ with weight matrix $\mathbf{W}_k$ of shape $d_k \times d_{k-1}$, that rule reads

$$\nabla_{\mathbf{W}_k} \ell = \bar{\mathbf{h}}_k \, \mathbf{h}_{k-1}^{\top}, \tag{34.47}$$

the outer product of the adjoint arriving from above and the activation stored from below. Intuitively, a weight matters in proportion to both how strongly its output affects the loss and how large its input activation was. What matters for our cost analysis is how much arithmetic this rule takes.

Forming the outer product $\bar{\mathbf{h}}_k \, \mathbf{h}_{k-1}^{\top}$ takes $d_k d_{k-1}$ multiplications, the same count as the forward matrix–vector product $\mathbf{W}_k \mathbf{h}_{k-1}$. Passing the adjoint down to the previous layer, $\bar{\mathbf{h}}_{k-1}^{\top} = \bar{\mathbf{h}}_k^{\top} \mathbf{W}_k$, takes another $d_k d_{k-1}$ multiplications. Measured in multiplications, the backward sweep therefore costs roughly twice a forward pass, so a forward pass plus a backward sweep costs about three forward passes, matching the 30 ms figure from the opening. Forward mode, by contrast, would have to treat every individual weight as its own input direction, requiring $d_k d_{k-1}$ passes for a single layer alone. That is why no framework trains neural networks in forward mode.

34.5.2.4 Cost of the Two Modes

Because we built both engines ourselves, we can measure their operation counts directly instead of relying on asymptotic arguments. Take $f(\mathbf{x}) = (\sum_j x_j^2)^2$ on $n = 200$ inputs $x_j = 0.01\,j$ for $j = 1, \dots, 200$. We choose a polynomial built solely from $+$ and $\times$ so that `Var` can record it without adding `exp` and `log` closures (which would mirror the methods of `Dual`). Its gradient is $\nabla f = 4(\sum_j x_j^2)\mathbf{x}$, and one plain evaluation runs $2n + 1 = 401$ primitive operations: n squares, n additions, and one final square. Forward mode will clearly need 200 passes of 401 operations each. Before reading the output below, pause and guess how many operations reverse mode will take.

```
n = 200
def f(xs):                          # f: R^n -> R, from + and * alone
    total = 0.0
    for x in xs:
        total = total + x * x
    return total * total            # (sum of squares)^2

xs0 = [0.01 * (j + 1) for j in range(n)]
Dual.ops = 0
grad_fwd = [f([Dual(x, 1.0 if k == j else 0.0) for k, x in enumerate(xs0)]).b
            for j in range(n)]   # forward mode: one pass per input direction
Var.ops = 0
xs = [Var(x) for x in xs0]          # reverse mode: one recorded pass...
backprop(f(xs))                     # ...and one backward sweep for all n partials
grad_rev = [x.grad for x in xs]

gap = max(abs(a - b) for a, b in zip(grad_fwd, grad_rev))
print(f'forward mode: {Dual.ops:,} operations ({n} passes of {2 * n + 1})')
print(f'reverse mode: {Var.ops:,} operations (one recorded pass, one sweep)')
```

```
print(f'ratio {Dual.ops / Var.ops:.1f}, largest gradient difference {gap:.1e}')
```

```
forward mode: 80,200 operations (200 passes of 401)
reverse mode: 802 operations (one recorded pass, one sweep)
ratio 100.0, largest gradient difference 0.0e+00
```

Forward mode executed $80{,}200 = n(2n+1)$ operations, whereas reverse mode executed only $802 = 2(2n+1)$: 401 recorded during the forward pass and 401 backward closures during the sweep. The ratio is $100 = n/2$, and the two gradients agree to every printed digit. For these engines, pass counts differ by a factor of n and operation counts by $n/2$; at $n = 10^8$, that recovers the factor from the opening up to the same small constant.

At first glance, the measured factor of two might seem to contradict the factor of three we quoted earlier. It does not: `Var.ops` counts each backward closure as a single operation, even though the closure for multiplication performs two multiplications of its own. Counting scalar multiplications directly gives 402 on the backward sweep against 201 on the forward pass, so computing the value and gradient together costs three times the value alone, just as it did for the linear layer. Including the additions that accumulate adjoints brings the backward sweep to 1204 operations and the total factor to four, still well within the Baur–Strassen bound.

Reverse mode trades memory for this speed. The tape must retain all 401 recorded intermediate values until the backward sweep consumes them, whereas forward mode, like the original program, holds only the current values alongside their tangents.

34.5.2.5 Gradient Checkpointing

A computation of T recorded steps keeps T intermediate values alive in memory until the backward sweep reaches them. In a deep neural network, those values are the activations of every layer for every example in the batch, and activation memory, rather than arithmetic, is often what caps the batch size or depth.

The simplest remedy divides the tape into segments of s steps. During the forward pass, we save only every s-th value, the T/s *checkpoints*, and discard everything in between. When the backward sweep arrives at a segment, it recomputes that segment's s steps from its saved checkpoint onto a temporary tape, sweeps that short tape backward, and immediately frees it. At any instant, memory holds only the T/s checkpoints plus one active segment of length s. Choosing s to balance these two terms minimizes peak memory:

$$M(s) = \frac{T}{s} + s, \qquad \min_s M(s) = M(\sqrt{T}) = 2\sqrt{T}. \tag{34.48}$$

The minimum follows either from setting $M'(s) = 1 - T/s^2 = 0$ or directly from the AM–GM inequality $T/s + s \geq 2\sqrt{T}$. Peak memory now grows only as the square root of the computation length rather than linearly. Because each step is recomputed once, the extra compute is just one additional forward pass. For $T = 400$, segment lengths $s = 1, 5, 10, 20, 100, 400$ yield peak memory $M = 401, 85, 50, 40, 104, 401$, with the minimum at $s = 20 = \sqrt{400}$. At $T = 10^4$, segments of length $s = 100$ store just 200 values instead of $10{,}000$ (Chen et al. 2016). Figure 34.26 illustrates this schedule.

Importantly, checkpointing alters time and memory without changing the gradient at all, provided the recomputation is deterministic: the recomputed intermediates are then iden-

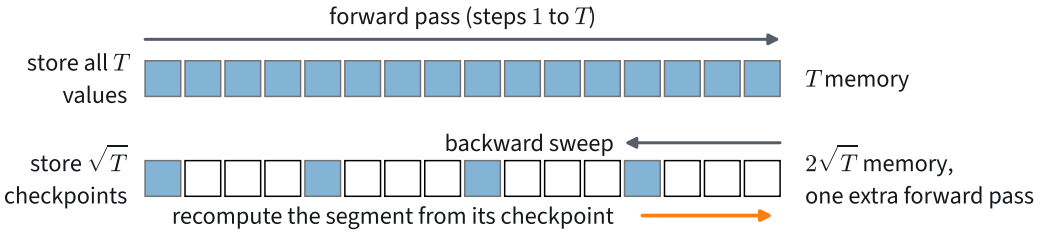


Figure 34.26: Gradient checkpointing on a tape of T steps. Plain reverse mode (top) keeps all T values until the backward sweep uses them. Checkpointing (bottom) keeps every s-th value; when the backward sweep reaches a segment, the segment is recomputed from its checkpoint and then swept. The memory is $T/s + s$, which is $2\sqrt{T}$ at $s = \sqrt{T}$, for one extra forward pass.

tical, so the resulting derivatives match plain reverse mode bit for bit. Deep learning frameworks expose this trade-off at the level of network blocks, storing only a checkpointed block's input and rerunning the block during backpropagation. There the proviso needs care. A rerun block draws a fresh dropout mask unless the state of the random number generator is restored, batch normalization updates its running statistics a second time, and nondeterministic GPU kernels need not reproduce the first run's bits. Applying the same idea recursively inside each segment pushes memory all the way down to $O(\log T)$ at the cost of $O(T \log T)$ recomputation.

34.5.3 Composing Jacobian Products

Automatic differentiation offers both linear operators our toy engines just built: the forward-mode JVP $\mathbf{J}\mathbf{v}$ and the reverse-mode VJP $\mathbf{u}^\top\mathbf{J}$. Once both are available as composable primitives, we can compute directional curvature, differentiate through numerical solvers, or assemble full Jacobians when needed.

34.5.3.1 Jacobian–Vector and Vector–Jacobian Products

In Section 1.5.4, we met the framework APIs for both products. Let us evaluate them on the plane map at $(0.5, 0.5)$ with $\mathbf{v} = \mathbf{u} = (1, 2)$. Using the Jacobian we computed earlier with `Dual`, we expect $\mathbf{J}\mathbf{v} = (1 + 2 \cdot 0.8776,\ 0.5 + 2) = (2.7552,\ 2.5)$ and $\mathbf{u}^\top\mathbf{J} = (1 + 2 \cdot 0.5,\ 0.8776 + 2) = (2,\ 2.8776)$, and the framework calls match these values directly.

```
def plane(z):                                       # (x + sin y, y + x^2/2)
    return torch.stack([z[0] + torch.sin(z[1]), z[1] + 0.5 * z[0]**2])

z0 = torch.tensor([0.5, 0.5])
v, u = torch.tensor([1.0, 2.0]), torch.tensor([1.0, 2.0])
_, Jv = torch.func.jvp(plane, (z0,), (v,))          # forward mode: J v
_, vjp_fn = torch.func.vjp(plane, z0)               # reverse mode: records once,
uJ, = vjp_fn(u)                                     # then u^T J per seed
J = torch.func.jacrev(plane)(z0)                    # one VJP per row
print('J v   =', Jv.numpy().round(4))
print('u^T J =', uJ.numpy().round(4))
```

```
print('J     =', J.numpy().astype(float).round(4).tolist())
```

```
J v   = [2.7552 2.5   ]
u^T J = [2.     2.8776]
J     = [[1.0, 0.8776], [0.5, 1.0]]
```

Both a JVP and a VJP cost a single sweep and propagate only a vector of derivative information, though the VJP must also hold the forward intermediates on its tape. Assembling a dense $m \times n$ Jacobian takes $\min(m, n)$ sweeps in the cheaper mode plus mn numbers of storage. Materializing the full matrix is the right choice when n is small, as in this 2×2 example, or when an algorithm needs its determinant, as in normalizing flows that transform probability densities by $|\det \mathbf{J}|$.

34.5.3.2 Hessian–Vector Products

For a model with $n = 10^6$ parameters, the Hessian has 10^{12} entries, which would occupy 4 TB in float32. Yet second-order optimization methods rarely need the matrix itself. Newton's method uses the Hessian only to solve $\mathbf{H}\mathbf{d} = -\nabla \ell$ for the update step $\mathbf{d}$, and Newton–CG (or truncated Newton) solves that linear system by conjugate gradients, an iterative solver that accesses $\mathbf{H}$ solely through matrix–vector products $\mathbf{H}\mathbf{v}$ (Nocedal and Wright 2006; Martens 2010). Can we compute $\mathbf{H}\mathbf{v}$ without ever forming $\mathbf{H}$?

We can, by composing reverse mode and forward mode. Because the Hessian is the Jacobian of the gradient map $\nabla \ell$, the product $\mathbf{H}\mathbf{v}$ is simply a JVP of the gradient, measuring how fast the gradient changes along $\mathbf{v}$:

$$\mathbf{H}\mathbf{v} = \mathbf{J}_{\nabla \ell}\, \mathbf{v} = \frac{d}{dt}\Big|_{t=0} \nabla \ell(\mathbf{x} + t\mathbf{v}). \tag{34.49}$$

A reverse sweep evaluates $\nabla \ell$, and a forward-mode pass through that gradient computation, seeded with tangent $\mathbf{v}$, differentiates it along $\mathbf{v}$. This composition is called *forward over reverse* (Pearlmutter 1994). Regardless of the dimension n, its cost is only a small constant multiple of computing a single gradient.

Log-sum-exp gives us a clean test with a closed-form answer. Its gradient is the softmax $\mathbf{p}$, so its Hessian is the softmax Jacobian (Section 34.4, equivalently the covariance matrix of a one-hot vector $\mathbf{e}_J$ with J drawn from $\mathbf{p}$). Therefore,

$$\mathbf{H} = \operatorname{diag}(\mathbf{p}) - \mathbf{p}\mathbf{p}^\top, \qquad \mathbf{H}\mathbf{v} = \mathbf{p} \odot \mathbf{v} - \mathbf{p}\,(\mathbf{p}^\top \mathbf{v}), \tag{34.50}$$

where $\odot$ denotes entrywise multiplication. Evaluating the right-hand side requires just $3n$ multiplications and never builds an $n \times n$ matrix. At $\mathbf{x} = (1, -0.5, 2)$ and $\mathbf{v} = (1, 0.5, -1)$, we have $\mathbf{p} = (0.2537, 0.0566, 0.6897)$ and $\mathbf{p}^\top \mathbf{v} = -0.4077$, which gives

$$\mathbf{H}\mathbf{v} = (0.3571,\ 0.0514,\ -0.4085).$$

Notice that this is a genuine check, unlike testing on a quadratic $\tfrac{1}{2}\mathbf{x}^\top \mathbf{A}\mathbf{x}$. For a quadratic, the gradient at $\mathbf{v}$ happens to equal $\mathbf{H}\mathbf{v}$, so a buggy implementation could still pass; for log-sum-exp, $\mathbf{H}$ varies with $\mathbf{x}$, and the gradient at $\mathbf{v}$, $\operatorname{softmax}(\mathbf{v}) = (0.574, 0.348, 0.078)$, is nowhere near $\mathbf{H}\mathbf{v}$.

With both modes available, forward over reverse takes a single line of code.

```
x = torch.tensor([1.0, -0.5, 2.0])
v = torch.tensor([1.0, 0.5, -1.0])
lse = lambda x: torch.logsumexp(x, 0)
_, Hv = torch.func.jvp(torch.func.grad(lse), (x,), (v,))  # forward over reverse
p = torch.softmax(x, 0)
print('forward over reverse :', Hv.numpy().round(4))
print('p*v - p (p.v)        :', (p * v - p * (p @ v)).numpy().round(4))
```

```
forward over reverse : [ 0.3571  0.0514 -0.4085]
p*v - p (p.v)        : [ 0.3571  0.0514 -0.4085]
```

The computed products match Equation 34.50 to full printed precision, without ever forming the 3×3 Hessian matrix. In Exercise 3, you will build the same product directly from our `Dual` and `Var` classes and compare forward-over-reverse with reverse-over-reverse.

34.5.3.3 Implicit Differentiation

Not every quantity of interest comes from an explicit formula. Many are defined implicitly as the solution to an equation and computed by an iterative solver: the fixed point of a recurrence, the equilibrium of a physical system, or the minimizer of an inner optimization problem. Recording twenty Newton iterations on a tape and backpropagating through all of them does work, but it stores every intermediate iterate and differentiates the particular trajectory the solver happened to follow. Can we obtain the derivative from the defining equation alone?

Start with the simplest possible case: a single equation in one unknown, $g(x, \theta) = x^2 - \theta = 0$, whose solution is $x^\star = \sqrt{\theta}$. Instead of differentiating the closed form, differentiate the identity $x^\star(\theta)^2 - \theta \equiv 0$ directly with respect to θ. That gives $2x^\star \, dx^\star/d\theta - 1 = 0$, and solving for the derivative yields

$$\frac{dx^\star}{d\theta} = \frac{1}{2x^\star} = \frac{1}{2\sqrt{\theta}},$$

which is indeed the derivative of $\sqrt{\theta}$ (equal to 0.25 at $\theta = 4$). Crucially, we never used an explicit formula for $x^\star$, only the equation $g(x, \theta) = 0$ and the value of the solution itself.

The general result is the *implicit function theorem* (Rudin 1976), which we state without proof. Let $\mathbf{g}(\mathbf{x}, \theta)$ map $\mathbb{R}^d \times \mathbb{R}^p$ to $\mathbb{R}^d$, suppose $\mathbf{g}$ is continuously differentiable near a solution $(\mathbf{x}^\star, \theta)$ of $\mathbf{g} = \mathbf{0}$, and suppose the Jacobian $\partial \mathbf{g}/\partial \mathbf{x}$ is invertible there. Then, in a neighborhood of θ, the solution is a differentiable function $\mathbf{x}^\star(\theta)$. Applying the chain rule to the identity $\mathbf{g}(\mathbf{x}^\star(\theta), \theta) \equiv \mathbf{0}$ gives $\frac{\partial \mathbf{g}}{\partial \mathbf{x}} \frac{\partial \mathbf{x}^\star}{\partial \theta} + \frac{\partial \mathbf{g}}{\partial \theta} = \mathbf{0}$, and therefore

$$\frac{\partial \mathbf{x}^\star}{\partial \theta} = -\Big(\frac{\partial \mathbf{g}}{\partial \mathbf{x}}\Big)^{-1} \frac{\partial \mathbf{g}}{\partial \theta}. \tag{34.51}$$

Both Jacobians on the right are evaluated solely at the solution $(\mathbf{x}^\star, \theta)$. Computing the full derivative matrix costs one linear solve with $\partial \mathbf{g}/\partial \mathbf{x}$ per parameter, regardless of how many iterations the solver took. Even better, in VJP form it takes just a single linear solve

for all parameters combined: $\mathbf{u}^\top \partial \mathbf{x}^\star / \partial \theta = -\mathbf{a}^\top \partial \mathbf{g} / \partial \theta$, where the adjoint vector $\mathbf{a}$ solves $(\partial \mathbf{g}/\partial \mathbf{x})^\top \mathbf{a} = \mathbf{u}$.

This same principle powers several major techniques in machine learning: the adjoint method for differentiating through ODE solvers (Section 38.3.2), deep equilibrium models (Bai et al. 2019) that define an entire layer as a fixed point $\mathbf{z}^\star = \mathbf{f}(\mathbf{z}^\star, \theta)$ and train it via Equation 34.51, and gradient-based hyperparameter tuning through an inner minimization.

Let us test Equation 34.51 on a nonlinear 2×2 system, $g_1 = x_1^2 + \theta x_2 - 2 = 0$ and $g_2 = x_1 x_2 + \theta - 1 = 0$, solved with twenty Newton steps, and check the implicit gradient against a central difference of the solver's output.

```
# Implicit gradient dx*/dtheta = -(dg/dx)^{-1} dg/dtheta on a 2x2 system,
# checked against a finite difference of the solver output
def g(x, th):                       # g(x, theta) = 0 defines x*(theta)
    return np.array([x[0]**2 + th * x[1] - 2, x[0] * x[1] + th - 1])

def dg_dx(x, th):                   # Jacobian in x, evaluated at (x, th)
    return np.array([[2 * x[0], th], [x[1], x[0]]])

def solve(th):                      # Newton iteration, from a fixed start
    x = np.array([1.0, 1.0])
    for _ in range(20):
        x = x - np.linalg.solve(dg_dx(x, th), g(x, th))
    return x

th, h = 0.3, 1e-6
xs = solve(th)
dg_dth = np.array([xs[1], 1.0])     # dg/dtheta at the solution
implicit = -np.linalg.solve(dg_dx(xs, th), dg_dth)
fd = (solve(th + h) - solve(th - h)) / (2 * h)
print('implicit-function gradient:', implicit.round(8))
print('finite-difference check   :', fd.round(8))
```

```
implicit-function gradient: [-0.11311451 -0.69322129]
finite-difference check   : [-0.11311451 -0.69322129]
```

The two vectors agree to the full precision of the finite difference, and none of the twenty Newton iterations ever appeared in the implicit formula. Because the derivative depends only on $\mathbf{g}$ and the solution $\mathbf{x}^\star$, any solver that converges to the same root produces the same gradient.

Like any theorem, the implicit function theorem has two clear boundaries, and both are easy to observe in practice. First, $\partial \mathbf{g}/\partial \mathbf{x}$ must be invertible at the solution. In our scalar example $x^2 = \theta$, at $\theta = 0$ we have $\partial g/\partial x = 2x^\star = 0$, so the formula divides by zero, reflecting the vertical tangent of $\sqrt{\theta}$ at the origin. Second, the formula assumes the solver has actually converged. To see what goes wrong when it has not, let us run Newton's method on $x^2 = 4$ starting from $x = 1$ and stop early after $k = 1, 2, 3$ steps, comparing the implicit formula $1/(2\tilde{x}_k)$ evaluated at the unconverged iterate $\tilde{x}_k$ with the derivative of the k-step unrolled solver itself (estimated by a central difference in θ).

```
def newton(theta, k, x=1.0):                        # k Newton steps on x^2 = theta
    for _ in range(k):
        x = x - (x * x - theta) / (2 * x)
    return x

theta, h = 4.0, 1e-6
for k in (1, 2, 3):
    x_k = newton(theta, k)
    unrolled = (newton(theta + h, k) - newton(theta - h, k)) / (2 * h)
    print(f'k={k}: x = {x_k:.5f}, formula {1 / (2 * x_k):.4f}, '
          f'unrolled {unrolled:.4f}, exact 0.25')
```

```
k=1: x = 2.50000, formula 0.2000, unrolled 0.5000, exact 0.25
k=2: x = 2.05000, formula 0.2439, unrolled 0.2900, exact 0.25
k=3: x = 2.00061, formula 0.2499, unrolled 0.2509, exact 0.25
```

After one, two, and three steps, Newton's method reaches 2.5, 2.05, and 2.00061. Against the true derivative 0.25, the implicit formula returns 0.2000, 0.2439, and 0.2499, while the unrolled derivative gives 0.5000, 0.2900, and 0.2509. Both are inaccurate before convergence: the implicit formula inherits the solver's residual error, and the unrolled derivative, which differentiates the truncated iteration path, is even further off here. The remedy is straightforward: let the solver converge first.

One fundamental problem remains open. Automatic differentiation computes the product $\mathbf{J}_K \cdots \mathbf{J}_1$ accurately up to floating-point rounding, but it cannot alter the value of that product. Across many layers, a long product of Jacobians can still shrink or explode geometrically (Section 34.4), and an exact gradient of 10^{-20} is no more useful for training than zero. Controlling vanishing and exploding gradients requires careful initialization and architectural choices, which we take up starting in Section 4.4.

34.5.4 Summary

The derivative of a program is the product of the Jacobians of its elementary steps, and automatic differentiation evaluates that product against a seed vector so that every step is a matrix–vector product rather than a matrix–matrix product. Forward mode runs the program on dual numbers to compute one Jacobian–vector product (one input direction) per pass; reverse mode records a tape during the forward pass and replays it backward to compute one vector–Jacobian product (one output direction) per pass. The shape of the Jacobian determines which mode is cheaper. Because a scalar loss has a single output, one reverse sweep yields all parameter gradients at roughly three times the cost of evaluating the loss. On $(\sum_j x_j^2)^2$ with $n = 200$ inputs, reverse mode took 802 operations against $80{,}200$ for forward mode. The tape's memory grows linearly with the computation length T, and gradient checkpointing lowers peak memory to $2\sqrt{T}$ for one extra forward pass. Finally, composing forward mode over reverse mode computes Hessian–vector products at a small constant multiple of one gradient evaluation, and the implicit function theorem differentiates a solver's output with a single linear solve at the converged solution.

34.5.5 Exercises

1. code **Dual and complex steps.** Take $\mathbf{f}(x, y) = (x + \sin y,\ y + x^2/2)$ at $\mathbf{x}_0 = (0.5, 0.5)$ and the second column $\mathbf{J}\mathbf{e}_2 = (\cos 0.5,\ 1)$ of its Jacobian. The central difference

$(\mathbf{f}(\mathbf{x}_0 + t\mathbf{e}_2) - \mathbf{f}(\mathbf{x}_0 - t\mathbf{e}_2))/(2t)$ has an error of order t^2 from truncation plus an error of order u/t from rounding, where $u \approx 10^{-16}$ is the float64 unit roundoff.

1. Predict the best step and the smallest error, then check the prediction for $t = 10^{-1}, 10^{-2}, \dots, 10^{-12}$ (you should find about 2×10^{-11} near $t = 10^{-5}$). Compare with one pass of `Dual` seeded with $\mathbf{e}_2$. Why does no step size enter the dual computation?
2. Repeat (a) for the first column $\mathbf{J}\mathbf{e}_1$, and explain why the central difference is exact there for every t, up to rounding. (Hint: $\mathbf{f}$ is a polynomial of degree two in x.)
3. Replacing ε by the imaginary unit i gives the complex-step estimate $\operatorname{Im} \mathbf{f}(\mathbf{x}_0 + it\mathbf{e}_2)/t$ (Squire and Trapp 1998), valid for an $\mathbf{f}$ that extends analytically to complex arguments, as this one does. Expand $\mathbf{f}(\mathbf{x}_0 + it\mathbf{e}_2)$ to third order in t and show that the error of the estimate is $O(t^2)$. Why does the estimate subtract no nearly equal numbers?
4. Evaluate the complex-step estimate with NumPy's complex arithmetic for $t = 10^{-1}, 10^{-2}, \dots, 10^{-20}$ and compare its errors with those of (a). Which difference between $\varepsilon^2 = 0$ and $i^2 = -1$ causes the $O(t^2)$ error of the complex step?

2. code **Hyperparameter gradient.** Ridge regression with one penalty $\lambda_j > 0$ per feature fits

$$\mathbf{w}^\star(\lambda) = \operatorname*{argmin}_{\mathbf{w}} \ \|\mathbf{X}\mathbf{w} - \mathbf{y}\|^2 + \sum_{j=1}^{d} \lambda_j w_j^2$$

on training data $(\mathbf{X}, \mathbf{y})$, and its validation loss on data $(\mathbf{X}_\mathrm{v}, \mathbf{y}_\mathrm{v})$ is $V(\lambda) = \|\mathbf{X}_\mathrm{v}\mathbf{w}^\star(\lambda) - \mathbf{y}_\mathrm{v}\|^2$.

1. Show that $\mathbf{w}^\star$ solves $\mathbf{g}(\mathbf{w}, \lambda) = (\mathbf{X}^\top\mathbf{X} + \operatorname{diag}(\lambda))\,\mathbf{w} - \mathbf{X}^\top\mathbf{y} = \mathbf{0}$, and use Equation 34.51 to derive $\partial\mathbf{w}^\star/\partial\lambda$.
2. Show that $\nabla_\lambda V = -\mathbf{a} \odot \mathbf{w}^\star$, where $\mathbf{a}$ solves one linear system with the matrix $\mathbf{X}^\top\mathbf{X} + \operatorname{diag}(\lambda)$, and give its right-hand side. How many linear solves does the forward route through $\partial\mathbf{w}^\star/\partial\lambda$ need for d penalties, and how many does this adjoint route need?
3. Implement V and the adjoint gradient in NumPy for random data with 50 training examples, 50 validation examples, and $d = 10$ features. Check the gradient against central differences of V.

Adapted from Alan Edelman and Steven G. Johnson, Matrix Calculus for Machine Learning and Beyond, problem set 2, MIT 18.S096, IAP 2023, Problem 1.

3. code **Hessian–vector product.** ● The function $f(\mathbf{x}) = \left(\sum_j x_j^2\right)^2$ of Section 34.5.2.4, evaluated at the inputs `xs0` with $n = 200$, is built from addition and multiplication alone, so both `Dual` and `Var` can evaluate it.

1. Derive ∇f and the Hessian $\mathbf{H}$ of f, and write $\mathbf{H}\mathbf{v}$ in a form that never builds $\mathbf{H}$.
2. Draw a random direction $\mathbf{v}$, wrap each input x_j as `Var(Dual(x_j, v_j))`, and run `backprop`. Show that the value parts of the resulting adjoints form ∇f and that their ε-parts form $\mathbf{H}\mathbf{v}$, and compare both with
   (a) and with the Hessian–vector product of Section 34.5.3.2 applied to the same f. Which operations of the backward closures run in dual arithmetic, and

why does this make the computation forward over reverse in the sense of Equation 34.49?

3. Nest the tape in itself instead. Wrap each input as `Var(Var(x_j))` and run `backprop`; the adjoints are now `Var` objects. Form $\sum_j (\nabla f)_j v_j$ from them and run `backprop` on it to obtain $\mathbf{Hv}$ by reverse over reverse. Count the `Var` objects that each method creates, for example with a counter in `Var.__init__`, and compare both counts with that of one plain gradient. Which method records the backward pass, and what does that cost in memory?

4. code **Checkpointing on the tape.** ● Let $x_k = x_{k-1} - 0.01\, x_{k-1}^2$ for $k = 1, \dots, T$ with $T = 400$ and $x_0 = 1$, and differentiate x_T with respect to x_0. `Var` has no subtraction, so record each step as `x + (-0.01) * x * x`.

   1. Record the chain on the tape with plain `backprop` and count the nodes that the tape holds when the backward sweep starts.
   2. `backprop` seeds the output adjoint with 1. Generalize it to accept any seed. Show that recording a segment $x_i \mapsto x_j$ from a fresh leaf for x_i and running the generalized `backprop` from x_j with the seed $\partial x_T / \partial x_j$ returns $\partial x_T / \partial x_i$ in the adjoint of the leaf.
   3. Implement the schedule of Figure 34.26 with segments of s steps: a forward pass in plain floats that stores $x_0, x_s, x_{2s}, \dots$, followed by a backward pass that re-records one segment at a time, from the last to the first, and applies (b). Check that the result equals the gradient of (a).
   4. For $s \in \{1, 5, 10, 20, 100, 400\}$, report the number of stored checkpoints, the largest tape in steps, and the number of evaluations of the step $x \mapsto x - 0.01x^2$, the first forward pass included. Confirm the memory $T/s + s$ of Equation 34.48, that is $401, 85, 50, 40, 104, 401$, and its minimizer $s = \sqrt{T} = 20$. Repeat the sweep for $T = 100$ and $T = 1600$ over the divisors $s \le 800$ of T (a single segment of 1600 steps exceeds Python's default recursion limit in `backprop`). How does the minimizing s scale with T, and how many step evaluations does the best schedule need?

5. [extended] **Equilibrium layer.** ●● A deep equilibrium layer (Bai et al. 2019) outputs the fixed point

$$\mathbf{z}^\star = \tanh(\mathbf{W}\mathbf{z}^\star + \mathbf{c}), \tag{34.52}$$

where $\mathbf{c}$ depends on the layer's input. Let the loss be $\ell = \frac{1}{2}\|\mathbf{z}^\star - \mathbf{y}\|^2$, and write $\mathbf{D} = \operatorname{diag}(1 - (z_1^\star)^2, \dots, 1 - (z_d^\star)^2)$ for the derivative of tanh at the fixed point.

   1. Show that the map $\mathbf{z} \mapsto \tanh(\mathbf{W}\mathbf{z} + \mathbf{c})$ is a contraction when $\|\mathbf{W}\|_2 < 1$, so that the fixed point exists and is unique, and that $\mathbf{I} - \mathbf{D}\mathbf{W}$ is then invertible. Use Equation 34.51 to derive $\nabla_{\mathbf{W}} \ell$, and show that it equals $(\mathbf{D}\mathbf{a})(\mathbf{z}^\star)^\top$ for the adjoint $\mathbf{a}$ that solves $(\mathbf{I} - \mathbf{D}\mathbf{W})^\top \mathbf{a} = \mathbf{z}^\star - \mathbf{y}$.
   2. Take $d = 20$, a standard normal $\mathbf{G} \in \mathbb{R}^{d \times d}$, $\mathbf{S} = \mathbf{G} + \mathbf{G}^\top$, and $\mathbf{W} = \rho\, \mathbf{S} / \|\mathbf{S}\|_2$ for $\rho \in \{0.5, 0.9, 0.99\}$, and draw the entries of $\mathbf{c}$ and $\mathbf{y}$ independently from $\mathcal{N}(0, 0.1^2)$. In NumPy, compute $\mathbf{z}^\star$ by fixed-point iteration and the gradient of (a) with one linear solve. Check a few of its entries against central differences of ℓ.
   3. Unrolled differentiation runs K iterations from $\mathbf{z}_0 = \mathbf{0}$ and backpropagates through all of them. Implement this backward sweep by hand, one VJP per it-

eration. Plot against K the error of the unrolled gradient relative to the gradient of (b), together with the forward error $\|\mathbf{z}_K - \mathbf{z}^\star\|$. Compare the decay rate of both errors with the spectral radius of $\mathbf{DW}$. How does the ratio of the two errors behave as K grows, and why? How does the memory of each gradient scale with K?

4. Replace the linear solve by m steps of the iteration $\mathbf{a} \leftarrow (\mathbf{z}^\star - \mathbf{y}) + (\mathbf{DW})^\top \mathbf{a}$ from $\mathbf{a} = \mathbf{0}$, a truncated Neumann series that Geng et al. (2021) use as a *phantom gradient*. Show that this truncated adjoint gives the same gradient as backpropagation through m iterations started at $\mathbf{z}^\star$. Plot the cosine between the truncated and the exact gradient against m for each ρ. The gradient for $m = 1$ needs no linear solve and is the Jacobian-free backpropagation of Wu Fung et al. (2022) . Is it still a descent direction for ℓ?

6. code **Second derivatives in one forward pass.** Adjoin ε with $\varepsilon^3 = 0$ (but $\varepsilon^2 \neq 0$), and write $a + b\varepsilon + c\varepsilon^2$ as a triple (a, b, c).

   1. Derive the product $(a, b, c)(a', b', c') = (aa',\ ab' + ba',\ ac' + bb' + ca')$, and show that for an analytic φ, $\varphi(a + b\varepsilon + c\varepsilon^2) = \varphi(a) + \varphi'(a)\, b\, \varepsilon + (\varphi'(a)\, c + \frac{1}{2}\varphi''(a)\, b^2)\varepsilon^2$.
   2. Implement `Dual2` with `__add__`, `__mul__`, `sin`, and `exp`, and evaluate $f(x) = \sin(x^2) + e^x$ on $(1.3, 1, 0)$. Show that the result is $(f(1.3),\ f'(1.3),\ \frac{1}{2}f''(1.3))$, and check $f''(1.3) \approx -3.2806$ against the hand formula $f''(x) = 2\cos x^2 - 4x^2 \sin x^2 + e^x$.
   3. For $\ell : \mathbb{R}^n \to \mathbb{R}$ with inputs seeded by a direction $\mathbf{v}$, what does the ε^2-part return? (Answer: $\frac{1}{2}\mathbf{v}^\top \mathbf{H}\mathbf{v}$, a number.) How many such passes give the vector $\mathbf{Hv}$ by polarization, against one forward-over-reverse pass?
   4. Write ε as a 3×3 matrix with $\varepsilon^3 = \mathbf{0}$ and verify the product of (a) by matrix multiplication.

34.6 Integral Calculus

Earlier chapters used continuous random variables without asking how their probabilities are computed. We described a normal distribution by its density (Section 2.1.3), wrote an expected loss as an average over that density, and treated both as given. In this section we ask how to add up a quantity that is spread continuously over an interval.

The standard normal distribution, the familiar bell curve, shows why continuous summation requires care. A standard normal random variable Z has density $p(z) = e^{-z^2/2}/\sqrt{2\pi}$. At $z = 0$ the curve reaches height $p(0) \approx 0.40$, yet the probability of landing on zero is $P(Z = 0) = 0$, and the same is true for every other single point. Intervals, on the other hand, do carry probability: $P(-1 \le Z \le 1) \approx 0.68$. Geometrically, that number is the area under p between -1 and 1, and $1/\sqrt{2\pi}$ is the constant that makes the total area under the curve equal to 1. Where does $\sqrt{2\pi}$ come from, and how do we compute 0.68?

These questions reach well beyond the bell curve. Every probability and every expected loss over a continuous variable is an area of this kind. Training a model through random sampling, as variational autoencoders and policy-gradient methods do, also requires differentiating such an area with respect to a model parameter.

Our strategy is simple: cut the area into thin slices, add their areas, and let the slices shrink to zero. The limit is the integral. The fundamental theorem of calculus then replaces

this infinite addition with the reverse of differentiation, which settles most integrals in a single line. It does not settle the bell curve, whose normalizing constant $\sqrt{2\pi}$ requires a detour through two dimensions. Three results anchor the section: the fundamental theorem Equation 34.55, the multivariable change-of-variables formula Equation 34.63, and the Gaussian integral Equation 34.65. If you already have first-year calculus at hand, you can skim through to Section 34.6.2.2. Differentiating an integral with respect to a parameter builds on all of these tools and has a section of its own in Section 34.7. We leave out measure theory and the Lebesgue integral, pointing interested readers to Section 34.8.

34.6.1 The Definite Integral

34.6.1.1 Riemann Sums

A probability density gives probability per unit length, so over a short interval of width ϵ the probability is approximately density times width. Over a longer interval, we cut it into thin slices, multiply height by width on each slice, and add the results. Nothing in this idea is specific to densities: the same recipe works for any function.

Let us start with a case we can work out by hand. Take $f(x) = x^2$ on $[0, 1]$ and divide the interval into N slices of equal width $\epsilon = 1/N$. On the i-th slice $[i/N, (i+1)/N]$, we approximate the area by a rectangle whose height is the value of f at the left endpoint, $(i/N)^2$. Adding all N rectangles gives

$$\sum_{i=0}^{N-1} \frac{1}{N}\left(\frac{i}{N}\right)^2 = \frac{1}{N^3}\sum_{i=0}^{N-1} i^2 = \frac{(N-1)N(2N-1)}{6N^3} = \frac{1}{3} - \frac{1}{2N} + \frac{1}{6N^2},$$

where the middle step uses the classical identity $\sum_{i=0}^{N-1} i^2 = (N-1)N(2N-1)/6$. As N grows, the rectangle sums converge to $\frac{1}{3}$, missing the limit by roughly $\frac{1}{2N}$, an error of first order in the slice width ϵ.

The helper function `left_sum` implements this **left-endpoint rule** for any function f: divide $[a, b]$ into slices of width ϵ and evaluate f at the left end of each slice, as shown in Figure 34.27. We will use `left_sum` for every grid sum in this section, starting with a quick check on x^2.

```
def left_sum(f, a, b, eps):
    """Left-endpoint Riemann sum of f on [a, b] with slices of width eps."""
    x = a + eps * np.arange(round((b - a) / eps))   # left ends of the slices
    return float(np.sum(eps * f(x)))

for N in [10, 100, 1000]:
    print(f'N={N:<5} left sum={left_sum(lambda x: x**2, 0, 1, 1 / N):.6f}  '
          f'formula={1/3 - 1/(2*N) + 1/(6*N**2):.6f}')
```

```
N=10    left sum=0.285000  formula=0.285000
N=100   left sum=0.328350  formula=0.328350
N=1000  left sum=0.332834  formula=0.332834
```

The numerical sums match the closed-form formula to every printed digit.

The definition. Nothing in this construction requires equal slice widths or left endpoints. More generally, a *partition* $P : a = x_0 < x_1 < \cdots < x_N = b$ cuts $[a, b]$ into subintervals, a *tag* $\xi_i \in [x_{i-1}, x_i]$ picks one sample point inside each slice, and the *mesh* $\|P\| = \max_i (x_i - x_{i-1})$ measures the width of the widest slice. A bounded function f is **Riemann integrable** on $[a, b]$ if there is a single number I that every tagged sum approaches as the mesh shrinks to zero, regardless of how the tags are chosen:

$$\int_a^b f(x)\,dx = I = \lim_{\|P\| \to 0} \sum_{i=1}^{N} f(\xi_i)(x_i - x_{i-1}). \tag{34.53}$$

In short, the definite integral is the common limit of the slice sums as the slices become arbitrarily thin. Our hand calculation above establishes $\int_0^1 x^2\,dx = \frac{1}{3}$. Every continuous function on $[a, b]$ is Riemann integrable, and so is every bounded function with finitely many jump discontinuities (Rudin 1976). Some pathological bounded functions, such as the indicator of the rational numbers, fail to have a Riemann integral; the Lebesgue integral (Section 34.8) extends integration to them while agreeing with the Riemann integral whenever the latter exists.

Two elementary properties follow straight from the definition and will power the proofs below. First, the integral is **monotone**: if $f \le g$ everywhere on $[a, b]$, then every tagged sum of f is no larger than the corresponding tagged sum of g, and taking limits preserves the inequality, giving $\int_a^b f\,dx \le \int_a^b g\,dx$. Second, it is **additive over subintervals**: for any $a \le b \le c$, $\int_a^c f\,dx = \int_a^b f\,dx + \int_b^c f\,dx$, because any partition that includes the point b splits cleanly into a partition of $[a, b]$ and a partition of $[b, c]$.

The running example. Until we reach the Gaussian integral, we will carry one concrete example through the section:

$$\int_0^2 \frac{x}{1 + x^2}\,dx = \tfrac{1}{2}\log 5 \approx 0.80472.$$

We will see why its value is $\frac{1}{2}\log 5$ once we establish the evaluation rule Equation 34.56. Figure 34.27 illustrates left-endpoint rectangles under this curve at three slice widths.

How quickly do the left sums converge to $\frac{1}{2}\log 5$? For x^2 on $[0, 1]$ the error was $\frac{1}{2N} = \frac{\epsilon}{2}$. If first-order convergence is typical, making the slices ten times thinner should reduce the error by a factor of ten. Let us check by printing the error alongside $0.2\,\epsilon$.

```
f = lambda x: x / (1 + x**2)
truth = math.log(5) / 2
for eps in [1e-1, 1e-2, 1e-3, 1e-4]:
    s = left_sum(f, 0, 2, eps)
    print(f'eps={eps:.0e}  sum={s:.6f}  truth-sum={truth - s:.2e}  '
          f'0.2*eps={0.2 * eps:.0e}')
print(f'truth = (1/2) log 5 = {truth:.6f}')
```

```
eps=1e-01  sum=0.783785  truth-sum=2.09e-02  0.2*eps=2e-02
eps=1e-02  sum=0.802710  truth-sum=2.01e-03  0.2*eps=2e-03
eps=1e-03  sum=0.804519  truth-sum=2.00e-04  0.2*eps=2e-04
```

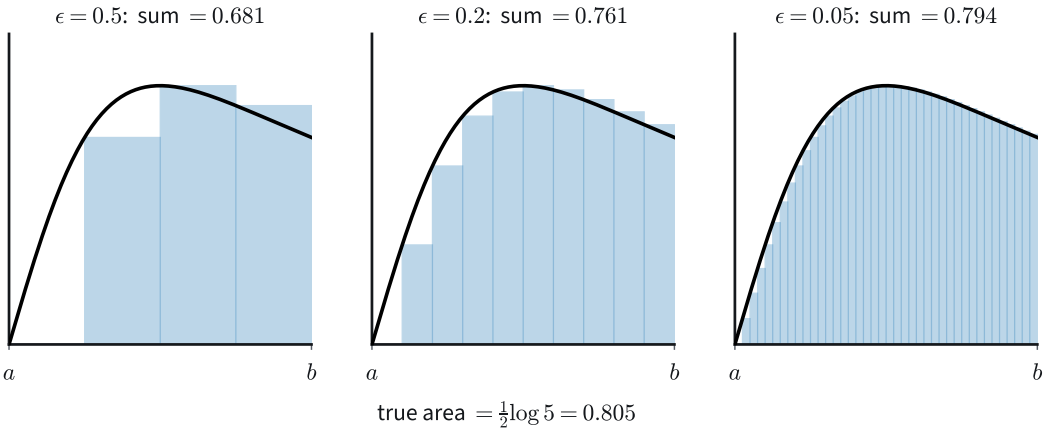


Figure 34.27: Left-endpoint rectangles under $f(x) = x/(1+x^2)$ on $[0, 2]$ for slice widths $\epsilon = 0.5$, 0.2, and 0.05. Each slice takes the height of f at its left endpoint, and the rectangle sums approach the area $\frac{1}{2}\log 5$ as $\epsilon \to 0$.

```
eps=1e-04  sum=0.804699  truth-sum=2.00e-05  0.2*eps=2e-05
truth = (1/2) log 5 = 0.804719
```

Each time ϵ shrinks by a factor of ten, the error shrinks by ten as well, and every sum sits below the true value. The constant 0.2 happens to be half of $f(2) - f(0) = 0.4$. This is no coincidence, and we will derive the formula behind it as soon as the fundamental theorem is in place.

34.6.1.2 Signed Area

Rectangles below the horizontal axis have negative height $f(\xi_i) < 0$ and enter the Riemann sum with a minus sign. The integral therefore measures *signed* area: regions where $f < 0$ contribute negatively, as in $\int_0^1 (-1)\,dx = -1$. By convention, integrating from right to left also negates the result, $\int_b^a f\,dx = -\int_a^b f\,dx$. This convention ensures that additivity holds regardless of how a, b, and c are ordered. Figure 34.28 illustrates both conventions on a single curve.

34.6.2 The Fundamental Theorem

34.6.2.1 The Theorem

Adding slices gave $\frac{1}{3}$ for x^2 only because $\sum i^2$ happens to have a closed form. For our running example, the left sum is $\sum_i \epsilon \cdot i\epsilon/(1+i^2\epsilon^2)$, and no algebraic identity collapses it. We need a way to evaluate integrals without adding slices at all.

The breakthrough comes from treating the upper limit as a variable. Fix a base point a and define the **area-so-far function**

$$F(x) = \int_a^x f(y)\,dy, \tag{34.54}$$

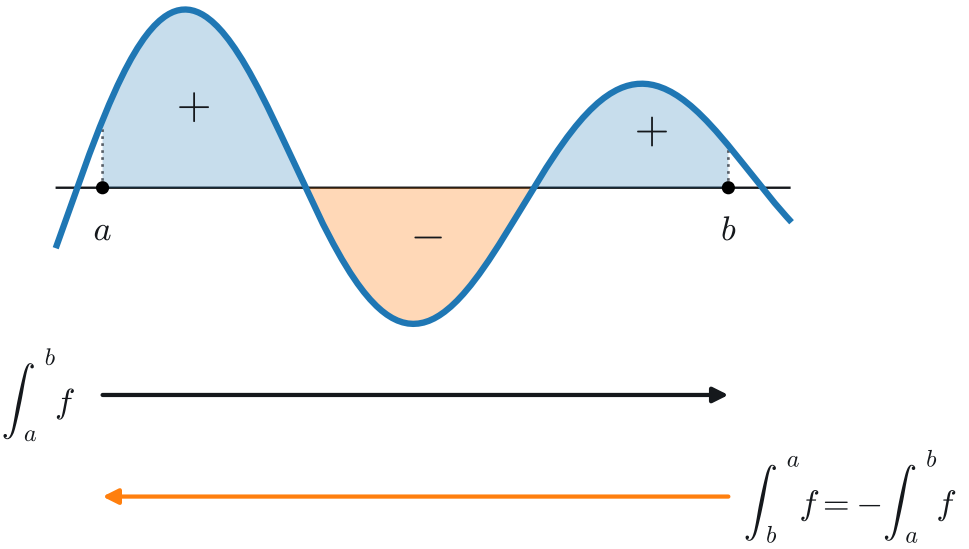


Figure 34.28: Signed area. Lobes above the axis count positively and the lobe below counts negatively, so $\int_a^b f$ totals the shaded regions with their signs; reversing the limits traverses the same area from right to left and negates the total.

which records the signed area accumulated from a out to x. This single function encodes every definite integral of f: by additivity, the area over any interval $[x_1, x_2]$ is simply the difference of two accumulated areas,

$$\int_{x_1}^{x_2} f(x)\,dx = F(x_2) - F(x_1),$$

as shown in Figure 34.29. Finding F therefore solves the whole problem.

How does F change when we nudge x slightly to the right? Figure 34.30 shows the accumulated area up to x alongside the thin strip of width ϵ added by moving to $x + \epsilon$. That strip is nearly a rectangle of height $f(x)$, so $F(x+\epsilon) - F(x) \approx \epsilon f(x)$. Dividing by ϵ suggests that $F'(x) = f(x)$.

Theorem (Fundamental theorem of calculus). *Let f be Riemann integrable on $[a, b]$, let $F(x) = \int_a^x f(y)\,dy$, and let f be continuous at a point $x \in (a, b)$. Then F is differentiable at x and*

$$\frac{dF}{dx}(x) = f(x). \tag{34.55}$$

In words, the rate at which area accumulates equals the current height of the curve.

Proof. Nudge x by $\epsilon > 0$ and look at the added area. By Equation 34.54 and additivity, the change is a single sliver,

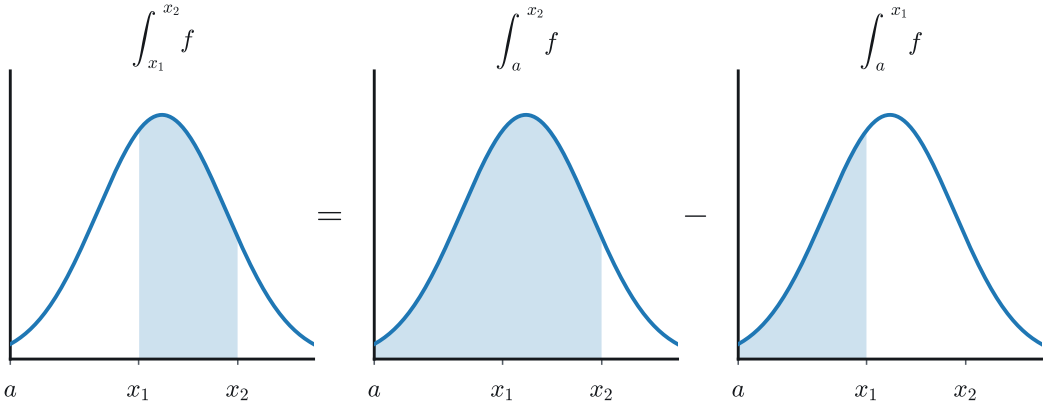


Figure 34.29: The area under f from x_1 to x_2 as the difference of two areas accumulated from the base point a at the plot's left edge: $\int_{x_1}^{x_2} f\,dx = \int_a^{x_2} f\,dx - \int_a^{x_1} f\,dx$.

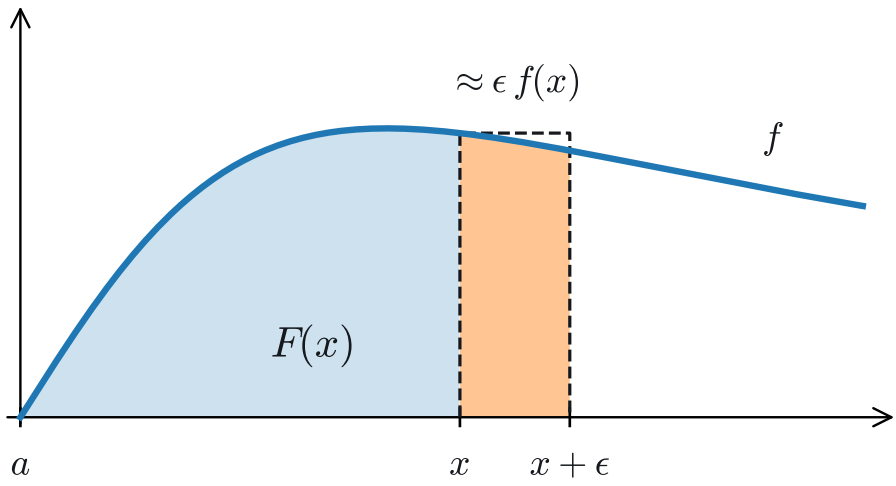


Figure 34.30: The area-so-far $F(x)$ under $f(y) = y/(1+y^2)$ from $a = 0$ to $x = 1.2$ (light), and the strip added by moving to $x + \epsilon$ (dark). The strip is close to the dashed rectangle of height $f(x)$, so $F(x+\epsilon) - F(x) \approx \epsilon f(x)$.

$$F(x+\epsilon) - F(x) = \int_x^{x+\epsilon} f(y)\,dy.$$

Let $m_\epsilon = \inf_{[x,x+\epsilon]} f$ and $M_\epsilon = \sup_{[x,x+\epsilon]} f$ be the tightest constant bounds on f over this sliver; both are finite because an integrable function is bounded. Since $m_\epsilon \le f \le M_\epsilon$ throughout $[x, x+\epsilon]$, monotonicity of the integral traps the sliver's area between two rectangles of width ϵ:

$$\epsilon\, m_\epsilon \;\le\; F(x+\epsilon) - F(x) \;\le\; \epsilon\, M_\epsilon.$$

Now divide through by ϵ. Because f is continuous at x, for any $\delta > 0$ the entire interval $[x, x+\epsilon]$ eventually lies inside the region where $|f(y) - f(x)| < \delta$. Both m_ϵ and M_ϵ therefore converge to $f(x)$ as $\epsilon \to 0$, and the squeeze forces $\lim_{\epsilon\to 0} \frac{F(x+\epsilon)-F(x)}{\epsilon} = f(x)$. When $\epsilon < 0$, the sliver sits to the left of x, and the same bounds hold with both sign flips canceling in the difference quotient. That limit is $F'(x)$. ■

Notice that the proof used only monotonicity and additivity, the two properties we read off the definition. The theorem turns the problem of finding an area into the problem of finding an **antiderivative**, a function G satisfying $G' = f$, which we can often read straight off the derivative table in Section 34.1.2. Any antiderivative yields the definite integral via the **evaluation rule**: *if f is continuous on $[a,b]$ and $G' = f$ on all of $[a,b]$, then*

$$\int_a^b f(x)\,dx = G(b) - G(a). \tag{34.56}$$

Why does any antiderivative work, not just F? Because F and G share the same derivative f on (a,b), so their difference $G - F$ has derivative zero and must be constant by the mean value theorem (Section 34.1.3). That additive constant cancels out in $G(b) - G(a) = F(b) - F(a) = \int_a^b f\,dx$.

The requirement "on all of $[a,b]$" is essential. Since $(-1/x)' = x^{-2}$, blindly applying the rule gives $\int_{-1}^{1} x^{-2}\,dx = [-1/x]_{-1}^{1} = -2$, a negative answer for a strictly positive integrand. The rule does not apply here because $G' = f$ breaks down at $x = 0$, where neither function is defined; in fact the integral diverges (Section 34.6.2.2).

Read backwards, every derivative formula becomes an integral: $(y^3)' = 3y^2$ gives $\int_0^1 y^2\,dy = \frac{1}{3}$ without a single slice, and $(e^t)' = e^t$ gives $\int_0^x e^t\,dt = e^x - 1$. For our running example, try $G(x) = \frac{1}{2}\log(1+x^2)$. By the chain rule, $G'(x) = \frac{1}{2}\cdot\frac{2x}{1+x^2} = \frac{x}{1+x^2}$, so

$$\int_0^2 \frac{x}{1+x^2}\,dx = G(2) - G(0) = \tfrac{1}{2}\log 5 - \tfrac{1}{2}\log 1 = \tfrac{1}{2}\log 5,$$

confirming the limit our Riemann sums were approaching.

The error of the left rule. The evaluation rule also explains where the constant 0.2 in the left-sum error came from. On the slice starting at x_i, a first-order expansion gives $f(x) \approx$

$f(x_i) + f'(x_i)(x - x_i)$. Integrating over $[x_i, x_i + \epsilon]$ shows that the true area of the slice exceeds its rectangle $\epsilon f(x_i)$ by roughly $\frac{\epsilon^2}{2} f'(x_i)$. Summing these triangular excesses across all slices yields $\frac{\epsilon}{2}$ times a left sum of f', which approximates $\frac{\epsilon}{2} \int_a^b f'\, dx = \frac{\epsilon}{2}(f(b) - f(a))$. To first order in ϵ,

$$\text{left sum} - \int_a^b f(x)\, dx \approx \frac{\epsilon}{2}(f(a) - f(b)).$$

For x^2 on $[0, 1]$ this gives $-\frac{\epsilon}{2} = -\frac{1}{2N}$, matching the leading error term we found by hand. For the running example, $f(0) = 0$ and $f(2) = 0.4$ give $-0.2\,\epsilon$. This estimate is numerical analysis rather than pure calculus, yet it accounts for the residual error in three more experiments in this section and serves as the baseline against which random sampling is judged in Section 36.5.1.7.

The fundamental theorem actually tells us more than $\int_0^2 f = G(2) - G(0)$: the area-so-far function $F(x)$ must equal $G(x) - G(0)$ at every x. Let us verify this by computing F as a cumulative left sum across $[0, 2]$ and comparing it with $G(x) - G(0)$ at every grid point. The two will not match to machine precision, because the left sum on $[0, x]$ carries the bias $\frac{\epsilon}{2}(f(0) - f(x))$. That bias is largest where f peaks, at $x = 1$ where $f(1) = \frac{1}{2}$, so the maximum discrepancy across the interval should be $\frac{\epsilon}{4}$.

```
G = lambda x: np.log(1 + x**2) / 2            # antiderivative of f
for eps in [1e-2, 1e-3, 1e-4]:
    x = eps * np.arange(round(2 / eps))       # left ends of the slices of [0, 2]
    F = np.cumsum(eps * f(x))                 # F[k]: left sum on [0, x_k + eps]
    gap = np.abs(F - (G(x + eps) - G(0))).max()
    print(f'eps={eps:.0e}  max over x of |F(x) - (G(x) - G(0))| = {gap:.2e}')
```

```
eps=1e-02  max over x of |F(x) - (G(x) - G(0))| = 2.51e-03
eps=1e-03  max over x of |F(x) - (G(x) - G(0))| = 2.50e-04
eps=1e-04  max over x of |F(x) - (G(x) - G(0))| = 2.50e-05
```

At every slice width, the maximum discrepancy is $\frac{\epsilon}{4}$, just as predicted. The cumulative sum tracks the antiderivative across the entire interval, not only at the right endpoint $x = 2$.

34.6.2.2 Improper Integrals

The normal, exponential, and many other probability densities live on unbounded domains, whereas Equation 34.53 defines integrals only over a finite interval $[a, b]$. We extend the definition the same way we built it, by taking a limit. The **improper integral** over $[a, \infty)$ is

$$\int_a^\infty f(x)\, dx = \lim_{b \to \infty} \int_a^b f(x)\, dx, \tag{34.57}$$

the area accumulated out to b in the limit of large b. If this limit exists and is finite, the integral *converges*; otherwise it *diverges*. To integrate over the entire real line, we split $\int_{-\infty}^{\infty} f\, dx$

at any convenient point into two one-sided improper integrals, and *both* must converge independently. A symmetric limit is not enough: $\int_{-b}^{b} x\,dx = 0$ for every b, yet $\int_0^\infty x\,dx = \infty$, so $\int_{-\infty}^{\infty} x\,dx$ does not exist.

The power law x^{-s} (with exponent $s > 0$) is the standard benchmark for deciding whether a tail decays fast enough to enclose finite area. Applying the evaluation rule on $[1, b]$ and taking $b \to \infty$ gives

$$\int_1^b x^{-s}\,dx = \frac{b^{1-s} - 1}{1-s} \xrightarrow{b\to\infty} \begin{cases} \dfrac{1}{s-1}, & s > 1 \text{ (converges)}, \\ \infty, & s \le 1 \text{ (diverges)}. \end{cases}$$

(At $s = 1$ the expression reads $0/0$; the definite integral is $\log b$, which is the limit of the formula as $s \to 1$ and diverges as $b \to \infty$.) Thus $\int_1^\infty x^{-2}\,dx = 1$ whereas $\int_1^\infty x^{-1}\,dx = \infty$: the dividing line between finite and infinite area lies right at $s = 1$.

Power laws are the canonical **heavy-tailed** distributions (Section 36.1, which also examines how heavy tails affect averages). For a density with tail $p(x) \propto x^{-s}$, the normalizing constant is finite only when $s > 1$, and the mean, tested by applying the same criterion to $x \cdot x^{-s} = x^{1-s}$, is finite only when $s > 2$.

The same limiting device handles an integrand that blows up at an endpoint: $\int_0^1 f\,dx = \lim_{a\to 0^+} \int_a^1 f\,dx$. For the power law x^{-s}, the condition reverses:

$$\int_0^1 x^{-s}\,dx = \lim_{a\to 0^+} \frac{1 - a^{1-s}}{1-s} = \frac{1}{1-s} \quad \text{for } s < 1,$$

while the integral diverges for $s \ge 1$. At $s = \frac{1}{2}$, for instance, $x^{-1/2}$ shoots to infinity at 0, yet the area under it is only 2. Near infinity, a power law must fall faster than $1/x$ to have finite area; near zero, it must grow more slowly than $1/x$.

When we approximate an improper integral numerically, two distinct errors enter: we cut off the domain at a finite b, and we sum slices of finite width ϵ. For $\int_0^\infty e^{-x}\,dx = 1$, we can separate the two. Below we print the exact partial integral $\int_0^b e^{-x}\,dx = 1 - e^{-b}$ (from the antiderivative $-e^{-x}$) next to a left sum with $\epsilon = 10^{-3}$.

```
for b in [1, 2, 5, 10, 20]:
    left = left_sum(lambda x: np.exp(-x), 0, b, 1e-3)
    print(f'b={b:<3}  left sum={left:.6f}  exact 1-e^-b={1 - math.exp(-b):.6f}')
```

```
b=1    left sum=0.632437  exact 1-e^-b=0.632121
b=2    left sum=0.865097  exact 1-e^-b=0.864665
b=5    left sum=0.993759  exact 1-e^-b=0.993262
b=10   left sum=1.000455  exact 1-e^-b=0.999955
b=20   left sum=1.000500  exact 1-e^-b=1.000000
```

The exact partial integrals converge to 1 rapidly: the *truncation error* from stopping at b is e^{-b}, already below the sixth decimal place by $b = 20$. The left sums, however, settle near 1.0005 rather than 1. That residual is *discretization error*, our left-rule bias $\frac{\epsilon}{2}(f(0)-f(b)) \to \frac{\epsilon}{2} = 0.0005$. To leading order the two errors simply add: pushing b higher removes truncation error, while shrinking ϵ removes discretization error.

34.6.2.3 Integration by Parts

Every differentiation rule, read through the evaluation rule, yields a matching integration rule. The product rule becomes integration by parts.

For continuously differentiable functions u and v on $[a, b]$,

$$\int_a^b u(x)\, v'(x)\, dx = \Big[\, u(x)\, v(x) \,\Big]_a^b - \int_a^b u'(x)\, v(x)\, dx, \tag{34.58}$$

where $[\, h(x) \,]_a^b$ is shorthand for $h(b) - h(a)$. The proof takes one line: by the product rule, uv is an antiderivative of $u'v+uv'$, so Equation 34.56 gives $\int_a^b (u'v+uv')\, dx = [\, u(x)v(x) \,]_a^b$, and subtracting $\int_a^b u'v\, dx$ from both sides yields Equation 34.58.

What this formula does is shift a derivative from one factor onto the other at the cost of a boundary term. It pays off whenever the new integrand $u'v$ is simpler than uv'. Choosing $u(x) = x$ and $v'(x) = e^{-x}$ (so $u' = 1$ and $v = -e^{-x}$), applying Equation 34.58 on $[0, b]$, and letting $b \to \infty$ gives

$$\int_0^\infty x\, e^{-x}\, dx = \Big[-x\, e^{-x} \Big]_0^\infty + \int_0^\infty e^{-x}\, dx = 0 + 1 = 1.$$

The boundary term vanishes because $xe^{-x} \to 0$ at both endpoints, leaving the improper integral we just evaluated. This integral is the mean of the exponential distribution (Section 36.3).

Because expectations are integrals of products, a recurring trick in probability and machine learning is to shift a derivative off an intractable factor and onto a tractable one. The bell curve shows this technique at its cleanest. Consider the unnormalized bell weight $w(z) = e^{-z^2/2}$, which satisfies $w'(z) = -z\, w(z)$. Let g be continuously differentiable with $g(z)w(z) \to 0$ as $z \to \pm\infty$, and suppose the two integrals below converge. Integrating by parts on $[-a, b]$ and then taking $a \to \infty$ and $b \to \infty$ separately yields

$$\int_{-\infty}^{\infty} g'(z)\, w(z)\, dz = \Big[\, g(z)\, w(z) \,\Big]_{-\infty}^{\infty} - \int_{-\infty}^{\infty} g(z)\, w'(z)\, dz = \int_{-\infty}^{\infty} z\, g(z)\, w(z)\, dz. \tag{34.59}$$

Under the bell weight, multiplying by z has the same effect as differentiating g. Both hypotheses hold for every polynomial g. Taking $g(z) = z$ gives $\int z^2 w\, dz = \int w\, dz$: the mean of z^2 under the bell weight is 1, regardless of what the normalizing constant $\int w\, dz$ turns out to be (we will show it is $\sqrt{2\pi}$ in Equation 34.66). Taking $g(z) = z^3$ gives $\int z^4 w\, dz = 3\int z^2 w\, dz = 3\int w\, dz$, so the mean of z^4 is 3. Once w is normalized into

the standard normal density $p = w/\sqrt{2\pi}$, this becomes $\mathbb{E}[g'(Z)] = \mathbb{E}[Z\, g(Z)]$, known as Stein's identity for the standard normal. Shifting a derivative from an unknown factor onto a known one is also the key step behind score matching (Section 38.6, (Hyvärinen 2005)). In $\mathbb{R}^d$, the corresponding rule moves a divergence instead of a one-dimensional derivative; Section 38.5.2.1 introduces the divergence, the Laplacian, and that multivariable identity where they are first needed.

34.6.2.4 Substitution

Reading the chain rule backwards through the evaluation rule produces the substitution rule. Suppose u is continuously differentiable on $[a, b]$ and f is continuous on an interval containing $u([a, b])$, with antiderivative G. By the chain rule, $(G \circ u)'(x) = f(u(x))\, u'(x)$, so

$$\int_a^b f(u(x))\, u'(x)\, dx = \int_a^b (G \circ u)'(x)\, dx = G(u(b)) - G(u(a)) = \int_{u(a)}^{u(b)} f(y)\, dy.$$

Read from right to left, this is the **change-of-variables formula**

$$\int_{u(a)}^{u(b)} f(y)\, dy = \int_a^b f(u(x))\, \frac{du}{dx}\, dx. \tag{34.60}$$

In words, an integral over y can be rewritten as an integral over x provided we multiply the integrand by the local stretch factor du/dx. Note that u does not even need to be monotone.

A single thin slice makes the stretch factor transparent. Partition $[a, b]$ into slices of width ϵ. The map u carries the slice at x to a slice of width approximately $\epsilon\, u'(x)$ at $y = u(x)$, where the curve has height $f(y) = f(u(x))$. Its area is therefore $f(u(x))\, u'(x)\, \epsilon$, which is the quantity the x-integral sums (Figure 34.31).

In practice, substitution works by spotting the pattern $f(u(x))\, u'(x)$ inside an integrand. In $\int_0^1 x\, e^{-x^2}\, dx$, set $u(x) = x^2$ so $u'(x) = 2x$: the integrand is $\frac{1}{2} e^{-u(x)} u'(x)$, and

$$\int_0^1 x\, e^{-x^2}\, dx = \frac{1}{2} \int_0^1 e^{-y}\, dy = \frac{1 - e^{-1}}{2} \approx 0.316.$$

Here the factor x supplied the $u'(x)$ that substitution needs. Remove that x and we are left with $\int e^{-x^2}\, dx$, which has no elementary antiderivative and is the very integral behind the bell curve's normalizer $1/\sqrt{2\pi}$. To evaluate it in Equation 34.65, we will obtain the missing factor by moving to two dimensions and switching to polar coordinates.

34.6.3 Multiple Integrals

From this point on we quote two standard theorems without proof, both from (Folland 1999): Fubini's theorem and the multivariable change-of-variables formula.

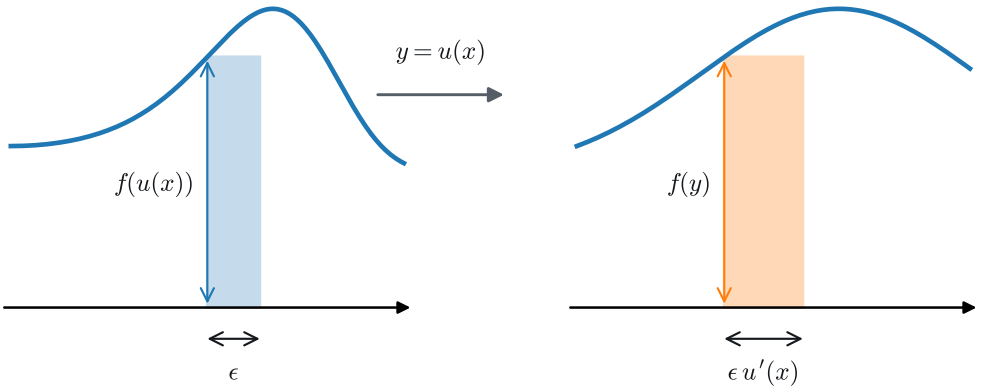


Figure 34.31: A thin sliver under the substitution $y = u(x)$. Left, the x side (the right-hand side of the formula): a sliver of width ϵ at x and height $f(u(x))$. Right, the y side: its image, of width $\epsilon\, u'(x)$ and the same height. Matching the two areas requires the factor du/dx.

34.6.3.1 Double Integrals and Fubini's Theorem

Because our route to $\int e^{-x^2}\,dx$ passes through two dimensions, we first extend integration from intervals to regions of the plane. Consider the two-dimensional bell surface $f(x,y) = e^{-x^2-y^2}$ over the square $[-2,2]^2$, which encloses a volume above the base plane. We approximate that volume just as we approximated areas: tile the base with $\epsilon \times \epsilon$ squares, erect on each square a column of height f, and sum the column volumes (Figure 34.32). For a function f on a rectangle $R = [a,b] \times [c,d]$, the limit of these box sums as $\epsilon \to 0$ is the **double integral** $\iint_R f(x,y)\,dx\,dy$, the signed volume beneath the surface.

How can we compute such a volume using our one-dimensional tools? Any finite sum can be totaled in whatever order we like. Indexing the grid squares by i along x and j along y, we can sum the boxes column by column (Figure 34.33):

$$\sum_{i,j} \epsilon^2 f(a+\epsilon i, c+\epsilon j) = \sum_{i} \epsilon \Big(\sum_{j} \epsilon\, f(a+\epsilon i, c+\epsilon j) \Big).$$

The inner sum is the left-endpoint approximation to $\int_c^d f(a+\epsilon i, y)\,dy$, the cross-sectional area of the surface at $x = a + \epsilon i$ (the highlighted column in Figure 34.32), and the outer sum integrates that slice area across x.

Reordering a finite grid sum is elementary, but ensuring that the equality survives the limit $\epsilon \to 0$ requires a theorem and a condition. **Fubini's theorem** states that if f is **absolutely integrable** on R, meaning $\iint_R |f(x,y)|\,dx\,dy < \infty$, then the double integral equals either iterated integral:

$$\iint_R f(x,y)\,dx\,dy = \int_c^d \left(\int_a^b f(x,y)\,dx \right) dy = \int_a^b \left(\int_c^d f(x,y)\,dy \right) dx. \tag{34.61}$$

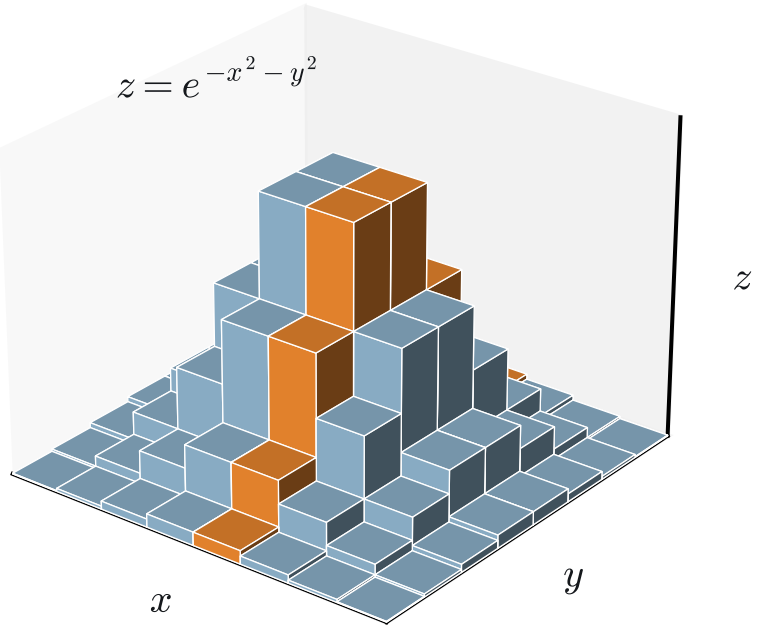


Figure 34.32: The bell surface $z = e^{-x^2-y^2}$ over the square $[-2, 2]^2$. Tiling the base with $\epsilon \times \epsilon$ squares ($\epsilon = 0.5$) and placing a box of height f on each approximates the volume under the surface; the highlighted column of boxes at a fixed x corresponds to one inner sum in Fubini's theorem.

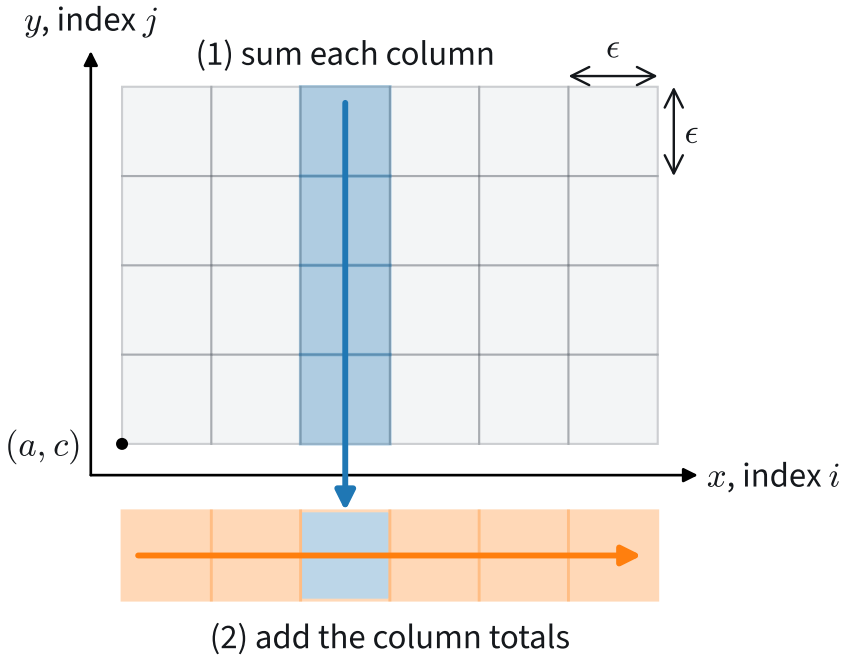


Figure 34.33: Totalling a grid of $\epsilon \times \epsilon$ cells with corner (a, c), indexed by i along x and j along y. (1) Sum each column, at fixed i, over j. (2) Add the column totals from left to right. Reordering a finite sum changes nothing; this discrete fact underlies iterated integration.

In geometric terms, a volume can be computed slice by slice in either direction. When $f \geq 0$, the same equality holds with no integrability check at all, allowing all three sides to equal $+\infty$ (*Tonelli's theorem*). Probability densities, squared losses, and other nonnegative integrands therefore require no verification; for a signed integrand, however, we must check $\iint |f| < \infty$ first. Fubini's theorem is at work whenever we marginalize a variable out of a joint density or swap an expectation with another integral.

The most common special case is a product integrand $f(x, y) = g(x)h(y)$. Because $g(x)$ is constant with respect to y, it pulls straight out of the inner integral, $\int_c^d g(x)h(y)\,dy = g(x)\int_c^d h(y)\,dy$, leaving

$$\iint_R g(x)\,h(y)\,dx\,dy = \int_a^b g(x)\,dx \int_c^d h(y)\,dy. \tag{34.62}$$

A product integrand integrates factor by factor. Crucially, the two-dimensional bell factors in this way, $e^{-x^2-y^2} = e^{-x^2}e^{-y^2}$, and our evaluation of the Gaussian integral below rests on this observation.

The absolute-integrability hypothesis is not a mere formality. Consider $f(x, y) = (x^2 - y^2)/(x^2 + y^2)^2$ on $[0, 1]^2$. A quick differentiation shows that $\partial_y(y/(x^2 + y^2)) = f(x, y)$, so the evaluation rule gives $\int_0^1 f(x, y)\,dy = \frac{1}{1+x^2}$ for $x > 0$, and integrating over x yields $\int_0^1 \frac{dx}{1+x^2} = \arctan 1 = \frac{\pi}{4}$. Yet $f(x, y) = -f(y, x)$, so integrating in the reverse order gives $-\frac{\pi}{4}$. The two orders disagree because $\iint |f| = \infty$: near the origin, $|f|$ grows like the inverse square of the distance, and its total volume diverges.

When the domain D is not a rectangle, $\int_D f(\mathbf{x})\,d\mathbf{x}$ is defined by *zero-extension*: set f to 0 outside D and integrate over any rectangle that contains D.

34.6.3.2 Change of Variables in Many Dimensions

The one-dimensional stretch factor du/dx generalizes naturally to higher dimensions, and linear maps already show the form it must take. A linear map scales every area or volume by the absolute value of its determinant (Section 33.2), while the sign of the determinant merely tracks orientation. For example, the map $\phi(u, v) = (au, bv)$ with $a, b > 0$ has determinant ab and stretches the unit disc (of area π) into the ellipse $x^2/a^2 + y^2/b^2 \leq 1$, whose area is therefore πab.

Locally, a smooth nonlinear map looks linear, with its Jacobian matrix serving as the linear approximation; the volume scale factor simply varies from point to point. Let $\Omega \subseteq \mathbb{R}^n$ be open and let $\phi : \Omega \to \mathbb{R}^n$ be a C^1**-diffeomorphism** onto its image: ϕ is continuously differentiable and injective, and its Jacobian $\mathbf{J}_\phi$ (the matrix of partial derivatives from Section 34.4) satisfies $\det \mathbf{J}_\phi(\mathbf{x}) \neq 0$ everywhere on Ω. Injectivity prevents the map from folding space onto itself, and a nonzero determinant prevents it from flattening a region to a lower dimension. Under these conditions, for any absolutely integrable f,

$$\int_{\phi(\Omega)} f(\mathbf{y})\,d\mathbf{y} = \int_\Omega f(\phi(\mathbf{x}))\,\bigl|\det \mathbf{J}_\phi(\mathbf{x})\bigr|\,d\mathbf{x}. \tag{34.63}$$

An infinitesimal cube of volume $d\mathbf{x}$ at $\mathbf{x}$ maps to an infinitesimal parallelepiped of volume $|\det \mathbf{J}_\phi(\mathbf{x})|\, d\mathbf{x}$ at $\mathbf{y} = \phi(\mathbf{x})$, and the Jacobian determinant adjusts the integrand for that local distortion. Figure 34.34 places the one- and two-dimensional pictures side by side.

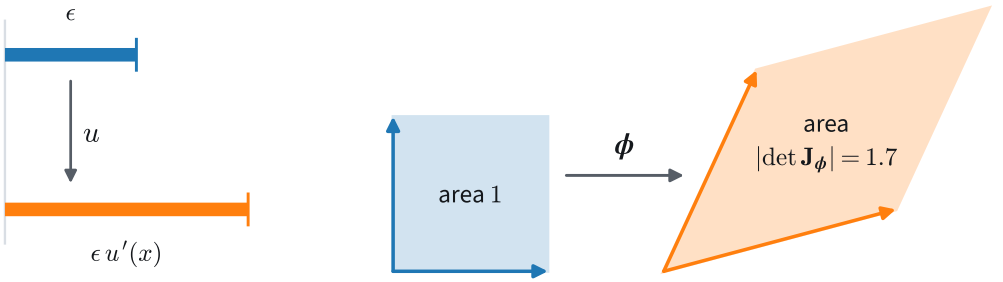


Figure 34.34: Left: one-dimensional substitution scales a segment of length ϵ by $u'(x)$. Right: in two dimensions, a linear map ϕ sends the unit square to a parallelogram whose area is the volume factor $|\det \mathbf{J}_\phi|$.

Why does Equation 34.63 carry the absolute value $|\det \mathbf{J}_\phi|$ rather than $\det \mathbf{J}_\phi$? Because the left-hand side integrates over an unoriented set, not between ordered endpoints. In one dimension, the order of the limits carries the sign: if u is decreasing, the limits in Equation 34.60 run backwards while $u' < 0$, so the two minus signs cancel and we can equivalently integrate over the image interval with $|u'|$. Regions in $\mathbb{R}^n$ have no natural ordering, so only this unsigned form generalizes.

When f is a probability density, Equation 34.63 tells us how a density transforms under a smooth invertible map, which is how a normalizing flow computes the density of its output (Section 36.2).

34.6.3.3 The Gaussian Integral

The normalizing constant $1/\sqrt{2\pi}$ of the bell curve comes from the one-dimensional integral $\int_{-\infty}^{\infty} e^{-x^2}\, dx$. Here the fundamental theorem cannot give us a closed form: an antiderivative certainly exists because e^{-x^2} is continuous, but it is a fundamentally new function that cannot be written in terms of elementary ones. Surprisingly, the integral over the entire real line can still be evaluated in closed form by taking a detour through two dimensions. The argument takes three steps and is the most elegant calculation in the section. (Normalized, the antiderivative is the **error function** $\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt$, which tends to 1 as $x \to \infty$ by the result we are about to prove; standard libraries provide it as `math.erf`.)

I. Square and factor. On the square $[-b, b]^2$, the product rule Equation 34.62 gives $\iint_{[-b,b]^2} e^{-x^2-y^2}\, dx\, dy = \left(\int_{-b}^{b} e^{-x^2}\, dx\right)^2$. Letting $b \to \infty$ on both sides yields

$$\left(\int_{-\infty}^{\infty} e^{-x^2}\, dx\right)^2 = \lim_{b\to\infty} \iint_{[-b,b]^2} e^{-x^2-y^2}\, dx\, dy = \iint_{\mathbb{R}^2} e^{-x^2-y^2}\, dx\, dy. \tag{34.64}$$

The square of our one-dimensional integral is simply the volume under the two-dimensional bell, and that surface is rotationally symmetric around the origin.

II. Polar coordinates. Rotational symmetry invites polar coordinates, $\phi(r,\theta) = (r\cos\theta, r\sin\theta)$, on the open set $\Omega = (0,\infty)\times(0,2\pi)$, where the map is injective. Its Jacobian determinant is

$$\left|\det \mathbf{J}_\phi\right| = \left|\det \begin{bmatrix} \cos\theta & -r\sin\theta \\ \sin\theta & r\cos\theta \end{bmatrix}\right| = r(\cos^2\theta + \sin^2\theta) = r.$$

Geometrically, a polar cell with radial side dr and arc length $r\,d\theta$ has area approximately $r\,dr\,d\theta$ (Figure 34.35), reflecting the fact that cells farther from the origin are wider. Crucially, this factor r supplies the missing factor x from our substitution example $\int_0^1 x\,e^{-x^2}\,dx$.

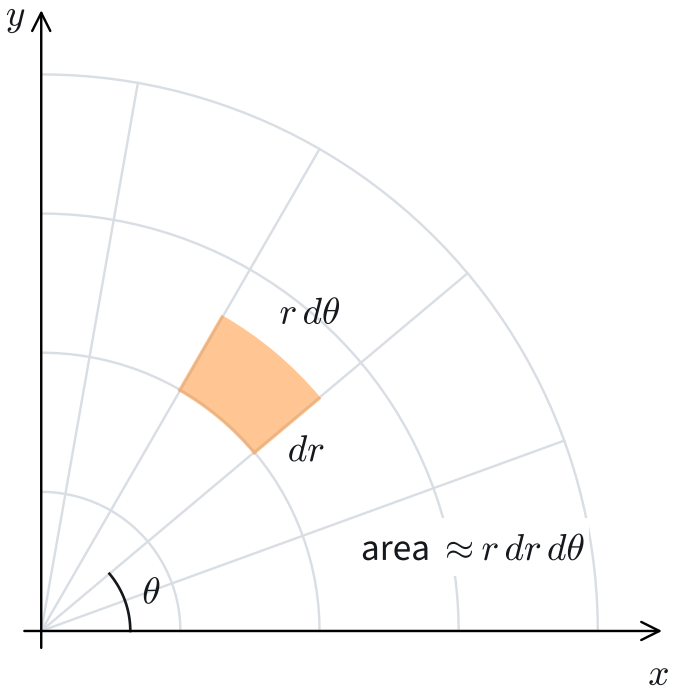


Figure 34.35: A polar grid with one cell shaded: radial side dr and arc side $r\,d\theta$, giving an area of approximately $r\,dr\,d\theta$. Switching to polar coordinates multiplies the integrand by $|\det \mathbf{J}_\phi| = r$.

III. Integrate. Applying Equation 34.63 and using the antiderivative $-\frac{1}{2}e^{-r^2}$ of re^{-r^2}, we obtain

$$\iint_{\mathbb{R}^2} e^{-x^2-y^2}\,dx\,dy = \int_0^{2\pi}\int_0^{\infty} r\,e^{-r^2}\,dr\,d\theta = 2\pi\cdot\frac{1}{2} = \pi.$$

Taking the positive square root of Equation 34.64 gives

$$\int_{-\infty}^{\infty} e^{-x^2}\,dx = \sqrt{\pi}. \tag{34.65}$$

Thus the total area under e^{-x^2} is $\sqrt{\pi} \approx 1.7725$.

Three technical points make this argument completely rigorous. First, because the integrand is nonnegative, integrals over expanding squares and over expanding discs converge to the same limit: every disc fits inside some square and every square inside some disc, so the shape of the growing region does not matter. Second, the polar map omits the nonnegative ray $\theta = 0$, which has zero area and contributes nothing to the integral. Third, we apply the change of variables on the bounded rectangles $(0, b) \times (0, 2\pi)$, where the integrand is bounded and continuous, and let $b \to \infty$ afterward.

The normalizing constant of the standard normal now follows from a single substitution. Setting $z = \sqrt{2}\,x$ in Equation 34.60 gives

$$\int_{-\infty}^{\infty} e^{-z^2/2}\,dz = \sqrt{2}\int_{-\infty}^{\infty} e^{-x^2}\,dx = \sqrt{2\pi}. \tag{34.66}$$

Dividing the bell $e^{-z^2/2}$ by $\sqrt{2\pi} \approx 2.5066$ scales its total area to 1, which is where the constant in the standard normal density comes from.

Let us check step I numerically: do the expanding squares $[-b, b]^2$ really exhaust the plane and converge to π? The code below computes the two-dimensional grid sum over $[-b, b]^2$ at $\epsilon = 0.01$ alongside the exact square volume $\left(\int_{-b}^{b} e^{-x^2}\,dx\right)^2 = \pi\,\mathrm{erf}(b)^2$ and the limiting value π.

```
eps = 0.01
for b in [1, 2, 3, 6]:
    g = -b + eps * np.arange(round(2 * b / eps))    # left ends of the slices
    x, y = np.meshgrid(g, g, indexing='ij')
    grid = np.sum(eps**2 * np.exp(-x**2 - y**2))
    print(f'b={b}  grid sum={grid:.6f}  pi*erf(b)^2={math.pi * math.erf(b)**2:.6f}'
          f'  pi={math.pi:.6f}')
```

```
b=1  grid sum=2.230949  pi*erf(b)^2=2.230985  pi=3.141593
b=2  grid sum=3.112266  pi*erf(b)^2=3.112270  pi=3.141593
b=3  grid sum=3.141454  pi*erf(b)^2=3.141454  pi=3.141593
b=6  grid sum=3.141593  pi*erf(b)^2=3.141593  pi=3.141593
```

The grid sums track $\pi\,\mathrm{erf}(b)^2$ closely and climb to π; by $b = 3$, the volume left outside the square is only about 10^{-4}. Agreement to four or more decimal places at $\epsilon = 0.01$ has a simple explanation: for an even integrand on a symmetric interval, the left rule's first-order bias $\frac{\epsilon}{2}\big(f(-b) - f(b)\big)$ vanishes identically.

We can now compute the standard normal probability $P(-1 \le Z \le 1) \approx 0.68$ as well. Substituting $z = \sqrt{2}\,t$ turns the area under $p(z) = e^{-z^2/2}/\sqrt{2\pi}$ into an error function:

$$P(-1 \le Z \le 1) = \frac{1}{\sqrt{2\pi}} \int_{-1}^{1} e^{-z^2/2}\, dz = \frac{1}{\sqrt{\pi}} \int_{-1/\sqrt{2}}^{1/\sqrt{2}} e^{-t^2}\, dt = \operatorname{erf}(1/\sqrt{2}) \approx 0.683.$$

Both questions we asked about the bell curve are now answered, and the apparent paradox that $p(0) \approx 0.40$ while $P(Z = 0) = 0$ disappears: a density measures probability per unit length, and a single point is an interval of length zero.

One question from the opening remains open. Training a model through random sampling requires knowing how an area of this kind changes when a parameter of the density moves, and for that we need to differentiate an integral.

34.6.4 Summary

The definite integral is the limit of sums of thin slices, measuring signed area under a curve. The fundamental theorem of calculus replaces that limit with antidifferentiation: because the area-so-far function satisfies $F' = f$, any antiderivative G gives $\int_a^b f\, dx = G(b) - G(a)$. Integration by parts and substitution are the product and chain rules read backwards. In multiple dimensions, Fubini's theorem reduces volume integrals to iterated one-dimensional integrals, and the change-of-variables stretch factor becomes $|\det \mathbf{J}_\phi|$, the local volume distortion; combined with polar coordinates, this yields $\int_{-\infty}^{\infty} e^{-x^2}\, dx = \sqrt{\pi}$ and the standard normal normalizing constant $\sqrt{2\pi}$.

34.6.5 Exercises

1. **Activations as accumulated areas.** Let H be the unit step, with $H(t) = 1$ for $t > 0$ and $H(t) = 0$ for $t \le 0$. Let $\phi(t) = e^{-t^2/2}/\sqrt{2\pi}$ be the standard Gaussian density and $\Phi(x) = \int_{-\infty}^{x} \phi(t)\, dt$ its distribution function.
   1. Show that $\int_{-\infty}^{x} H(t)\, dt = \mathrm{ReLU}(x)$. Which hypothesis of the fundamental theorem Equation 34.55 fails at $x = 0$? What are the one-sided derivatives of the integral there?
   2. Write $\Phi(x) = \Phi(0) + \int_0^x \phi(t)\, dt$ to conclude that $\Phi' = \phi$. Then substitute $t = \sqrt{2}\, s$ to show that $\Phi(x) = \frac{1}{2}(1 + \operatorname{erf}(x/\sqrt{2}))$.
   3. The GELU activation of Section 4.1 is $x\Phi(x)$. Derive its derivative, find the points where the derivative is largest and smallest, and evaluate it there with `math.erf`. In which interval does the slope of GELU lie, and how does that interval compare with the slopes 0 and 1 of ReLU?
2. **Heavy tails.** The *Pareto density* with tail index $\alpha > 0$ is $p(x) = \alpha x^{-\alpha-1}$ for $x \ge 1$ and $p(x) = 0$ otherwise. The *Cauchy density* is $p(x) = \frac{1}{\pi(1+x^2)}$ on $\mathbb{R}$.
   1. Show that the Pareto density is normalized for every $\alpha > 0$. For which α are $\mathbb{E}[X]$ and $\mathbb{E}[X^2]$ finite? Compute $\mathbb{E}[X]$ where it is finite.
   2. Show that the Cauchy density is normalized, using the antiderivative $\arctan x$ of $\frac{1}{1+x^2}$.
   3. Show that $\int_{-b}^{b} x\, p(x)\, dx = 0$ for the Cauchy density and every $b > 0$. Why does the Cauchy density nevertheless have no mean under the definition of $\int_{-\infty}^{\infty}$ that

follows Equation 34.57? How fast does $\int_0^b x\,p(x)\,dx$ grow with b?

3. **Gamma function by parts.** The *Gamma function* is $\Gamma(t) = \int_0^\infty u^{t-1}e^{-u}\,du$. For $t < 1$ the integrand is unbounded at $u = 0$, and the integral near 0 is defined by a limit, $\int_0^1 = \lim_{a\to 0^+}\int_a^1$, as in Section 34.6.2.2.
   1. For which t does the integral converge at $u = 0$? Compare the condition with the test for $\int_0^1 x^{-s}\,dx$ in Section 34.6.2.2.
   2. Integrate by parts Equation 34.58 at a general $t > 0$ to show that $\Gamma(t+1) = t\,\Gamma(t)$, and conclude that $\Gamma(n+1) = n!$ for every integer $n \geq 0$. Where does the argument use $t > 0$? Use the result to compute the variance of the exponential density e^{-u} on $u \geq 0$, whose mean we computed in Section 34.6.2.3.
   3. Substitute $u = x^2$ and use the Gaussian integral Equation 34.65 to evaluate $\Gamma(\tfrac{1}{2})$.
   4. The product rule Equation 34.62, applied d times, gives $\int_{\mathbb{R}^d} e^{-\|\mathbf{x}\|^2}\,d\mathbf{x} = \pi^{d/2}$. In polar form the same integral is $S_{d-1}\int_0^\infty r^{d-1}e^{-r^2}\,dr$, where S_{d-1} is the surface area of the unit sphere in $\mathbb{R}^d$. Show that the radial integral is $\tfrac{1}{2}\Gamma(d/2)$ and deduce $S_{d-1} = 2\pi^{d/2}/\Gamma(d/2)$. Integrate $S_{d-1}r^{d-1}$ over $r \in [0,1]$ to show that the unit ball has volume $V_d = \pi^{d/2}/\Gamma(d/2+1)$. Evaluate V_d for $d \in \{1,2,3,5,10,20\}$ and explain why $V_d \to 0$. A midpoint grid with three points per axis on the cube $[-1,1]^{10}$ has 3^{10} points; what fraction of them lies inside the unit ball, and what does that say about grids in high dimensions?
4. **Swapping the order.** Fubini's theorem Equation 34.61, together with zero-extension for regions that are not rectangles, lets us choose the order of integration.
   1. The inner integral of $\int_0^1 \big(\int_y^1 e^{-x^2}\,dx\big)\,dy$ has no elementary antiderivative. Sketch the triangular region of integration, reverse the order, and evaluate the integral in closed form.
   2. Let $X \geq 0$ have density p. Write $x = \int_0^x dt$ inside $\mathbb{E}[X] = \int_0^\infty x\,p(x)\,dx$ and swap the two integrals to show that $\mathbb{E}[X] = \int_0^\infty P(X > t)\,dt$. Why is the swap valid even when $\mathbb{E}[X] = \infty$? Check the formula on the density $\lambda e^{-\lambda x}$ for $x \geq 0$.
5. **Gaussian normalizers and moments.** The text normalizes the standard normal and computes its moments with the Stein identity Equation 34.59 for the bell weight. Let $p(x) = \frac{1}{\sqrt{2\pi\sigma^2}}\exp\!\big(-\frac{(x-\mu)^2}{2\sigma^2}\big)$ with $\sigma > 0$.
   1. Substitute $x = \mu + \sigma z$ and use Equation 34.66 to show that p is normalized. Which μ and σ^2 does the density $\frac{1}{\sqrt{\pi}}e^{-x^2}$ have?
   2. Let X have density p, and let g be continuously differentiable with $g(x)\,p(x) \to 0$ as $x \to \pm\infty$. Use $p'(x) = -\frac{x-\mu}{\sigma^2}\,p(x)$ and integration by parts to show that
$$\mathbb{E}\big[(X-\mu)\,g(X)\big] = \sigma^2\,\mathbb{E}\big[g'(X)\big]. \tag{34.67}$$
   Deduce that $\mathbb{E}[(X-\mu)^2] = \sigma^2$ and $\mathbb{E}[(X-\mu)^4] = 3\sigma^4$.
   3. Let $\Sigma = \mathbf{L}\mathbf{L}^\top$ be a $d \times d$ covariance matrix with invertible $\mathbf{L}$. Use the change of variables $\mathbf{x} = \mathbf{L}\mathbf{z}$ in Equation 34.63 and the product rule Equation 34.62 to show that
$$\int_{\mathbb{R}^d} \exp\!\big(-\tfrac{1}{2}\mathbf{x}^\top\Sigma^{-1}\mathbf{x}\big)\,d\mathbf{x} = (2\pi)^{d/2}\,(\det\Sigma)^{1/2}. \tag{34.68}$$

34.7 Expectations and Their Gradients

Almost every loss we have minimized so far is an average: over a minibatch of examples, over random dropout masks, or over noise injected into an input. As long as the source of randomness does not depend on the model parameters, differentiating such an average is straightforward. The derivative of an average equals the average of the derivatives, so minibatch gradient descent computes it directly without any extra machinery. The difficulty arises when the parameters control the probability distribution itself. A variational autoencoder draws its latent code from a distribution whose mean and variance are produced by its encoder; a policy-gradient agent (Section 14.7) samples actions from the very policy it is trying to improve. In both cases, the objective is an expectation over samples whose distribution shifts with the parameters, yet a sample, once drawn, is just a number. It carries no memory of the parameter that generated it.

Consider the simplest problem of this kind. Let $X \sim \mathcal{N}(\theta, 1)$ be a normal random variable with mean θ and unit variance, and take the expected squared loss

$$\ell(\theta) = \mathbb{E}\big[X^2\big] = \theta^2 + 1.$$

The closed form follows immediately by writing $X = \theta + Z$ with Z standard normal: $\mathbb{E}[(\theta + Z)^2] = \theta^2 + 2\theta\,\mathbb{E}[Z] + \mathbb{E}[Z^2]$, where $\mathbb{E}[Z] = 0$ by symmetry and $\mathbb{E}[Z^2] = 1$ (Section 34.6.2.3). At $\theta = 1.5$, the exact derivative is $\ell'(1.5) = 3$. In practice, however, a model can draw samples of X but cannot evaluate $\ell(\theta)$ in closed form. How, then, can we estimate $\ell'(\theta)$ from samples alone?

Calculus offers two fundamentally different answers. Both express the expectation as an integral (Section 34.6) and pass the derivative inside it. From there they diverge: the *score-function* estimator differentiates the probability density while leaving the samples untouched, whereas the *pathwise* estimator differentiates the samples while leaving the noise density untouched. Neither method dominates in every setting, and comparing them (Section 34.7.3) is the central goal of this section. First, we briefly recall how expectations are written as integrals and establish when passing a derivative inside an integral is valid, because that step is where subtle errors can creep in.

34.7.1 Expectations as Integrals

Before differentiating anything, let us fix our notation for densities and expectations. The probability chapter develops both notions in detail (Section 36.1); here we restate them in the integral language of the preceding section. A probability **density** on a domain $\mathcal{X}$ is a nonnegative function whose total integral equals 1:

$$p(x) \geq 0, \qquad \int_{\mathcal{X}} p(x)\,dx = 1. \tag{34.69}$$

The probability that a random variable X with density p lands in an interval $[a, b]$ is the area under p over that interval, $P(a \leq X \leq b) = \int_a^b p(x)\,dx$ (Section 36.1). Because a density measures probability per unit length rather than probability itself, its value can exceed 1; for example, the uniform density on $[0, \frac{1}{2}]$ equals 2 everywhere on its support. Our main running example is the **standard normal** distribution $\mathcal{N}(0, 1)$, whose density $p(z) = e^{-z^2/2}/\sqrt{2\pi}$ is normalized by the Gaussian integral of Section 34.6.

An **expectation** computes the weighted average of X, or of a function $g(X)$, weighting each outcome by its probability density:

$$\mathbb{E}[X] = \int_{\mathcal{X}} x\, p(x)\, dx, \qquad \mathbb{E}[g(X)] = \int_{\mathcal{X}} g(x)\, p(x)\, dx. \tag{34.70}$$

The expectation $\mathbb{E}[g(X)]$ is well defined whenever the integral converges absolutely, $\int_{\mathcal{X}} |g(x)|\, p(x)\, dx < \infty$. Heavy-tailed distributions can violate this condition; the Cauchy density in exercise 2, for instance, has no finite mean.

In deep learning, expectations rarely have closed-form expressions, and they often live in thousands of dimensions where numerical quadrature on a grid is out of the question. Instead, we draw independent samples $x_1, \dots, x_n$ from p and form the sample average $\frac{1}{n}\sum_{i=1}^{n} g(x_i)$, known as the *Monte Carlo* estimate. If a single sample $g(X)$ has standard deviation σ, the Monte Carlo average has root-mean-square error $\sigma/\sqrt{n}$, regardless of the dimension (Section 36.5.1.7). Both gradient estimators in this section are Monte Carlo averages of this form. Because both are unbiased for the same true gradient, they differ only in the spread of a single sample, making the single-sample standard deviation σ the natural yardstick for comparing them.

34.7.2 Differentiation under the Integral Sign

34.7.2.1 The Swap

Our toy loss $\ell(\theta) = \mathbb{E}[X^2]$ is an integral that varies with a parameter θ. In general, a parameter can affect an integral in two places: through the limits of integration, which the fundamental theorem of calculus (Section 34.6) handles, or through the integrand itself.

When the limits of integration are fixed and only the integrand depends on θ, we differentiate by passing the derivative inside the integral:

$$\frac{d}{d\theta} \int f(x, \theta)\, dx = \int \frac{\partial f}{\partial \theta}(x, \theta)\, dx. \tag{34.71}$$

Geometrically, the rate of change of the total area equals the integral of the pointwise rate of change of the height. We will refer to this exchange of derivative and integral as the *swap*; both gradient estimators depend on it. At first glance the identity looks harmless, since a derivative is a limit of difference quotients, an integral is a limit of sums, and finite sums and differences always commute. Yet exchanging two limits is never automatic and requires regularity conditions. On a bounded interval of x, continuity of $\partial_\theta f$ in both variables is enough. On an unbounded domain, we also need an integrable function of x that dominates $|\partial_\theta f(x, \theta')|$ for all θ' near θ (an immediate consequence of the dominated convergence theorem, quoted without proof from (Folland 1999)). For our running example $f(x, \theta) = x^2 p_\theta(x)$ with a normal density, such a dominating bound always exists because the Gaussian tail decays faster than any polynomial grows.

34.7.2.2 The Leibniz Rule

When the limits of integration also move with θ, the plain swap is incomplete and gives the wrong answer. The Leibniz rule supplies the missing boundary terms: the interior change and the sweeping boundaries simply add (Figure 34.36).

Proposition (Leibniz rule). *Let f and $\partial_\theta f$ be continuous on a region containing $\{(x,\theta) : x \in [a(\theta), b(\theta)]\}$ for θ in an open interval, and let a and b be differentiable. Then*

$$\frac{d}{d\theta}\int_{a(\theta)}^{b(\theta)} f(x,\theta)\,dx = \int_{a(\theta)}^{b(\theta)} \frac{\partial f}{\partial \theta}(x,\theta)\,dx + f(b(\theta),\theta)\,b'(\theta) - f(a(\theta),\theta)\,a'(\theta). \tag{34.72}$$

Proof. Define $H(\alpha,\beta,\theta) = \int_\alpha^\beta f(x,\theta)\,dx$. By the fundamental theorem of calculus, $\partial_\beta H = f(\beta,\theta)$ and $\partial_\alpha H = -f(\alpha,\theta)$, while the swap on the bounded interval $[\alpha,\beta]$ gives $\partial_\theta H = \int_\alpha^\beta \partial_\theta f\,dx$. Since all three partial derivatives are continuous, the multivariable chain rule (Section 34.3) applies to the composition $\theta \mapsto H(a(\theta),b(\theta),\theta)$ and combines the three terms weighted by $a'(\theta)$, $b'(\theta)$, and 1. $\blacksquare$

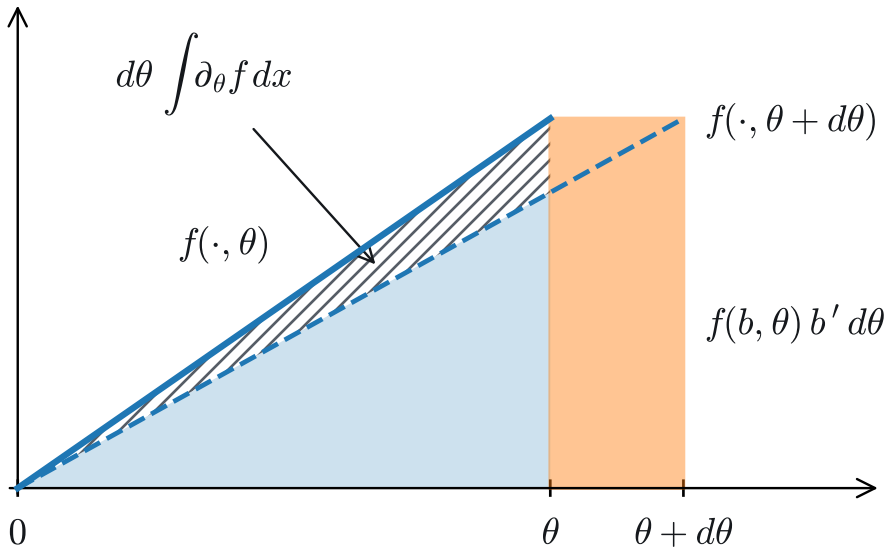


Figure 34.36: The Leibniz rule for $f(x,\theta) = x/\theta$ on $[0,\theta]$, the uniform example. When θ increases by $d\theta$, the integrand curve shifts to $f(\cdot,\theta+d\theta)$ (dashed; here it drops, so the hatched region represents lost area), while the right boundary moves outward and sweeps in a strip of height $f(b,\theta)$ and width $b'\,d\theta$.

Ignoring the boundary term is not a minor inaccuracy; it can flip the sign of the derivative. Suppose X is uniform on $[0,\theta]$, with density $1/\theta$ on that interval. Its mean is $\mathbb{E}[X] = \int_0^\theta \frac{x}{\theta}\,dx = \frac{\theta}{2}$, whose derivative is $+\frac{1}{2}$. Differentiating only the integrand gives $\int_0^\theta -\frac{x}{\theta^2}\,dx = -\frac{1}{2}$, which has the wrong sign. Adding the upper boundary contribution $f(\theta,\theta)\,b'(\theta) = \frac{\theta}{\theta}\cdot 1 = 1$ restores the true derivative, $-\frac{1}{2}+1 = +\frac{1}{2}$. Whenever the support of a distribution depends on the parameter, the plain swap alone is not valid (exercise 1 works out $\mathbb{E}[X^2]$).

34.7.2.3 Log-Integral-Exp

The swap also produces the continuous counterpart of the log-sum-exp function. Applying the chain rule and Equation 34.71 to $\log \int e^{\theta x} p(x)\, dx$ yields

$$\frac{d}{d\theta} \log \int e^{\theta x} p(x)\, dx = \frac{\int x\, e^{\theta x} p(x)\, dx}{\int e^{\theta x} p(x)\, dx}.$$

The right-hand side is the mean of x under the exponentially *tilted* density proportional to $e^{\theta x} p(x)$. For the standard normal density, completing the square via $\theta z - \frac{1}{2} z^2 = \frac{1}{2}\theta^2 - \frac{1}{2}(z-\theta)^2$ shows that the tilted density is $\mathcal{N}(\theta, 1)$ and that $\log \mathbb{E}[e^{\theta Z}] = \theta^2/2$, whose derivative θ is indeed the tilted mean. Just as the gradient of log-sum-exp is the softmax vector, the gradient of log-integral-exp is an expectation. We will encounter this identity again as the derivative $A'(\eta)$ of the log-partition function in Section 36.4.

34.7.3 Score-Function and Pathwise Gradients

Applying the swap to an expected loss gives the integral $\int g(x)\, \partial_\theta p_\theta(x)\, dx$. Unfortunately, we cannot estimate this integral directly by Monte Carlo sampling: $\partial_\theta p_\theta$ is not a probability density (it takes both positive and negative values and integrates to zero), so we cannot draw samples from it. Two simple identities rewrite this integral back into an expectation that samples can estimate.

34.7.3.1 The Score-Function Estimator

Let p_θ be a density parameterized by θ. Applying the chain rule to $\log p_\theta(x)$ yields the **log-derivative identity**:

$$\nabla_\theta\, p_\theta(x) = p_\theta(x)\, \nabla_\theta \log p_\theta(x). \tag{34.73}$$

In words, the gradient of a density equals the density itself multiplied by the gradient of its logarithm, a vector known as the **score** $\nabla_\theta \log p_\theta$. Whenever the swap is valid and the support of p_θ does not move with θ, the score has expectation zero: $\mathbb{E}[\nabla_\theta \log p_\theta(X)] = \int \nabla_\theta p_\theta\, dx = \nabla_\theta \int p_\theta\, dx = \nabla_\theta 1 = \mathbf{0}$. The uniform density on $[0, \theta]$ shows why fixed support is essential: its score is $\partial_\theta \log(1/\theta) = -1/\theta$, whose expectation is $-1/\theta \neq 0$, because differentiating $\int_0^\theta p_\theta(x)\, dx = 1$ also picks up the boundary term $p_\theta(\theta) = 1/\theta$. The mean-zero property of the score is also the starting point for Fisher information (Section 36.6).

Multiplying Equation 34.73 by a function $g(x)$ that does not depend on θ and integrating gives the **score-function identity**:

$$\nabla_\theta\, \mathbb{E}_{X \sim p_\theta}[g(X)] = \int g(x)\, \nabla_\theta p_\theta(x)\, dx = \mathbb{E}\big[g(X)\, \nabla_\theta \log p_\theta(X)\big]. \tag{34.74}$$

This identity turns the gradient of an expected loss back into an expectation under p_θ: each sample's loss $g(x)$ is weighted by how strongly θ increases that sample's log-probability. To estimate it by Monte Carlo, we draw samples $x_i \sim p_\theta$ and average $g(x_i) \nabla_\theta \log p_\theta(x_i)$. In

reinforcement learning, this is the REINFORCE estimator of policy-gradient methods (Section 14.7) (Williams 1992). (If g also depends directly on θ, the product rule simply adds the term $\mathbb{E}[\nabla_\theta g(X,\theta)]$.)

Because the score has mean zero, we may subtract any constant **baseline** b (independent of x) from $g(X)$ without biasing the estimator (Section 14.8): $\mathbb{E}[(g(X)-b)\nabla_\theta \log p_\theta(X)] = \mathbb{E}[g(X)\nabla_\theta \log p_\theta(X)]$. Choosing b well can substantially reduce the variance of the estimate.

34.7.3.2 The Pathwise Estimator

The second identity restores the sample's memory of θ. Often, a sample from p_θ can be generated by applying a deterministic transformation $x = T_\theta(\mathbf{z})$ to an auxiliary noise variable $\mathbf{z} \sim q$ whose distribution does not depend on θ. A normal random variable $\mathcal{N}(\mu,\sigma^2)$, for instance, is simply $\mu + \sigma z$ with $z \sim \mathcal{N}(0,1)$. Rewriting the expectation as an integral over $\mathbf{z}$ removes θ from the density entirely, so the swap yields the **pathwise identity**:

$$\nabla_\theta \, \mathbb{E}_{X\sim p_\theta}[g(X)] = \nabla_\theta \int g(T_\theta(\mathbf{z}))\, q(\mathbf{z})\, d\mathbf{z} = \mathbb{E}_{\mathbf{z}\sim q}\Big[\nabla_\theta \, g(T_\theta(\mathbf{z}))\Big]. \tag{34.75}$$

Instead of differentiating the density, this estimator differentiates right through the sample: automatic differentiation traces the path from θ through T_θ into g. Known as the *reparameterization trick*, this is how variational autoencoders backpropagate through Gaussian latent variables (Kingma and Welling 2014; Rezende et al. 2014). The estimator itself is older: the simulation literature had long used it under the names pathwise and infinitesimal-perturbation derivative (Mohamed et al. 2020). Valid use requires T_θ to be differentiable in θ, g to be differentiable along the path (a Lipschitz g suffices when X has a continuous density), and an integrable dominating bound for the swap. Notice that no boundary term ever appears, even if the support of X moves with θ, because the integral is taken over $\mathbf{z}$ whose density q stays fixed.

The differentiability requirement on g is not a technicality we can ignore. Consider the step function $g(x) = \mathbf{1}(x > 0)$, which equals 1 for positive x and 0 otherwise, with $X = \theta + z$. The true expectation $\mathbb{E}[g(X)] = P(z > -\theta)$ has derivative $\varphi(\theta) = e^{-\theta^2/2}/\sqrt{2\pi}$, which evaluates to about 0.13 at $\theta = 1.5$. Yet the pathwise sample $\partial_\theta \mathbf{1}(\theta + z > 0)$ is 0 for every z except the single point $z = -\theta$ (an event of probability zero), so the pathwise estimator returns 0 with probability 1. By contrast, the score-function sample $\mathbf{1}(x > 0)\, z$ has expectation $\int_{-\theta}^{\infty} z\,\varphi(z)\,dz = \big[-\varphi(z)\big]_{-\theta}^{\infty} = \varphi(\theta)$, recovering the exact derivative. Because the score-function estimator never differentiates through g or the sampling step, it applies equally well to discontinuous losses and to discrete distributions such as Bernoulli or categorical choices, where no differentiable path from θ to the sample exists.

34.7.3.3 The Two Estimators Compared

We now have two unbiased estimators of the same derivative, and our running example $\ell(\theta) = \mathbb{E}[X^2]$ with $X = \theta + z$ is simple enough to compute the variance of both by hand. For the pathwise estimator, a single sample contributes $\partial_\theta(\theta + z)^2 = 2X$, whose variance is $\mathrm{Var}(2X) = 4$. For the score-function estimator, the score of the normal location family is $\partial_\theta \log p_\theta(X) = X - \theta = z$, so one sample contributes $X^2 z$, or $(X^2 - b)z$ when using a baseline b. To find the variance of $X^2 z$, we use the standard normal moments $\mathbb{E}[z^2] = 1$,

$\mathbb{E}[z^4] = 3$, and $\mathbb{E}[z^6] = 15$, all of which follow from Stein's identity $\mathbb{E}[g'(Z)] = \mathbb{E}[Z\, g(Z)]$ (Section 34.6.2.3). By symmetry, all odd powers of z vanish in the second moment:

$$\mathbb{E}\big[(\theta + z)^4 z^2\big] = \theta^4\, \mathbb{E}[z^2] + 6\theta^2\, \mathbb{E}[z^4] + \mathbb{E}[z^6] = \theta^4 + 18\theta^2 + 15.$$

Subtracting the squared mean $(2\theta)^2$ leaves a variance of $\theta^4 + 14\theta^2 + 15$. The variance-minimizing constant baseline turns out to be $b^\star = \theta^2 + 3$, reducing the variance to $8\theta^2 + 6$ (exercise 2). At $\theta = 1.5$, the three single-sample standard deviations are 2 for the pathwise estimator, 7.18 for the plain score-function estimator, and 4.90 for the score-function estimator with the optimal baseline. Although the baseline helps noticeably, it cannot close the gap for this loss, since $8\theta^2 + 6 > 4$ for every θ.

Let us test all three closed-form predictions in a single Monte Carlo simulation. Because the pathwise derivative is simply $2x_i$, we do not even need automatic differentiation here. If our algebra is right, the three sample standard deviations should land near 2, 7.18, and 4.90, and each sample mean should fall within a few standard errors $\mathrm{sd}/\sqrt{n}$ of the true derivative 3.

```
theta, n = 1.5, 200_000
z = np.random.default_rng().standard_normal(n)     # noise, independent of theta
x = theta + z                                      # samples of N(theta, 1)
pathwise = 2 * x                                   # d/dtheta (theta + z_i)^2
score = x**2 * z                                   # g(x) times the score x - theta
baseline = (x**2 - (theta**2 + 3)) * z             # with the optimal baseline
exact_sd = {'pathwise': 2, 'score': math.sqrt(theta**4 + 14 * theta**2 + 15),
            'score+baseline': math.sqrt(8 * theta**2 + 6)}
print(f'true derivative 2*theta = {2 * theta}')
for name, s in zip(exact_sd, [pathwise, score, baseline]):
    print(f'{name:15s} mean={s.mean():.4f}  sd={s.std():.3f} '
          f'(exact {exact_sd[name]:.3f})  se={s.std() / math.sqrt(n):.4f}')
```

```
true derivative 2*theta = 3.0
pathwise        mean=3.0041  sd=2.000 (exact 2.000)  se=0.0045
score           mean=3.0068  sd=7.146 (exact 7.181)  se=0.0160
score+baseline  mean=2.9961  sd=4.858 (exact 4.899)  se=0.0109
```

In typical runs, all three means lie within a couple of standard errors of 3, and the empirical standard deviations match the closed-form predictions to within a few percent. For a fixed sample budget, the pathwise estimate is about 3.6 times more accurate than the plain score-function estimate; equivalently, the plain score-function estimator needs roughly thirteen times as many samples to reach the same precision.

It would be a mistake, however, to conclude that the pathwise estimator always has lower variance. Consider an oscillating loss $g(x) = \sin(\omega x)$, for which the true derivative of the expectation is $\omega \cos(\omega\theta) e^{-\omega^2/2}$. The pathwise sample $\omega \cos(\omega X)$ scales with the frequency ω and has standard deviation close to $\omega/\sqrt{2}$, whereas the score-function sample $\sin(\omega X)\, z$ has standard deviation at most $\sqrt{\mathbb{E}[z^2]} = 1$ because $|\sin| \le 1$. Whenever an integrand oscillates rapidly, its derivative can be huge even while its values remain bounded. Let us compare the two estimators at $\omega = 1$ and $\omega = 3$, reusing the samples already drawn.

```
for omega in [1, 3]:
    true = omega * math.cos(omega * theta) * math.exp(-omega**2 / 2)
    path, score = omega * np.cos(omega * x), np.sin(omega * x) * z
    print(f'omega={omega}  true={true:+.4f}  pathwise {path.mean():+.4f} '
          f'(sd {path.std():.2f})  score {score.mean():+.4f} (sd {score.std():.2f})')
```

```
omega=1  true=+0.0429  pathwise +0.0414 (sd 0.66)  score +0.0432 (sd 0.54)
omega=3  true=-0.0070  pathwise -0.0067 (sd 2.12)  score -0.0080 (sd 0.71)
```

Even at $\omega = 1$, the score-function estimator already has the smaller spread (about 0.5 versus 0.7), and at $\omega = 3$ it wins by a factor of three (about 0.7 versus 2.1). Both sample means agree with the exact derivatives within a few standard errors.

Neither estimator dominates in every problem. The pathwise estimator differentiates g, so its variance grows whenever g' is large; the score-function estimator multiplies g by the score, so its variance grows whenever g itself is large. In both cases, the single-sample standard deviation σ governs how many samples are needed for a given accuracy. Because σ also depends on the distribution from which we draw samples, choosing a proposal distribution that makes σ small leads directly to importance sampling, which we take up in Section 36.8.

34.7.4 Summary

When a model's parameters govern the distribution of its samples, the gradient of an expected loss must be rewritten as an expectation before Monte Carlo sampling can estimate it. Both ways of doing so pass the derivative inside the integral, a step that requires an integrable dominating bound and, whenever the limits of integration move with the parameter, the boundary terms of the Leibniz rule. The score-function identity differentiates the log-density; it works for arbitrary losses g and discrete random variables, though its samples scale with the magnitude of g and can be noisy without a good baseline. The pathwise identity differentiates through a reparameterized sample; it requires a differentiable g and often, though not always, achieves much lower variance. Because both estimators have root-mean-square error $\sigma/\sqrt{n}$ (Section 36.5.1.7), the single-sample standard deviation σ decides which one is preferable for a given problem.

34.7.5 Exercises

1. **A moving support.** Let X be uniform on $[0, \theta]$ with $\theta > 0$, so that $\ell(\theta) = \mathbb{E}[X^2] = \int_0^\theta \frac{x^2}{\theta}\, dx = \frac{\theta^2}{3}$ and $\ell'(\theta) = \frac{2\theta}{3}$.
    1. Differentiate ℓ with the Leibniz rule Equation 34.72. Which part of the result does the plain swap Equation 34.71 produce, and which part does it miss?
    2. On the support, $\log p_\theta(x) = -\log\theta$. Compute the expectation of the score-function estimator $X^2\, \partial_\theta \log p_\theta(X)$ of Equation 34.74. Then let g be continuous on $[0, \infty)$. Show that the mean of $g(X)\, \partial_\theta \log p_\theta(X)$ equals the plain swap Equation 34.71 applied to $\mathbb{E}[g(X)] = \int_0^\theta g(x)\, p_\theta(x)\, dx$. What is its bias as an estimate of $\frac{d}{d\theta}\mathbb{E}[g(X)]$, in terms of $g(\theta)$ and θ?
    3. Write $X = \theta U$ with U uniform on $[0, 1]$ and compute the expectation of the pathwise estimator of Equation 34.75. Why does it need no boundary term, although the support of X moves with θ?

2. code **Estimator variance for a quartic.** ● Let $X \sim \mathcal{N}(\theta, 1)$, write $X = \theta + z$, and let $s(X) = \partial_\theta \log p_\theta(X) = z$ be the score.
   1. Take $g(X) = X^4$, so that $\frac{d}{d\theta}\mathbb{E}[X^4] = 4\theta^3 + 12\theta$. Derive the single-sample pathwise estimator from Equation 34.75 and the single-sample score-function estimator from Equation 34.74. Use the identity of exercise 5 (b) to show that both have mean $4\theta^3 + 12\theta$.
   2. For a constant baseline b, show that the variance of $(g(X) - b)\, s(X)$ is minimized by $b^\star = \mathbb{E}[g(X)s(X)^2]/\mathbb{E}[s(X)^2]$. For $g(X) = X^2$ show that $b^\star = \theta^2 + 3$, and use $\mathbb{E}[z^2] = 1$, $\mathbb{E}[z^4] = 3$, $\mathbb{E}[z^6] = 15$ to derive the three variances 4, $\theta^4 + 14\theta^2 + 15$ and $8\theta^2 + 6$ that the text quotes.
   3. Evaluate $b^\star$ for $g(X) = X^4$. With 10^6 samples at $\theta \in \{0, 1, 2\}$, estimate the standard deviations of the pathwise estimator, the score-function estimator, and the score-function estimator with the baseline. For which θ is the baseline's gain largest, and does it ever close the gap to the pathwise estimator? Explain the ordering from the sizes of g' and of g.
   4. Take $g(x) = \mathbf{1}(x > 0)$ at $\theta = 1.5$. The text shows that the pathwise estimator is 0 for almost every sample, while the score-function estimator $\mathbf{1}(x > 0)\, z$ has the correct mean $\varphi(\theta) = 0.1295$. Verify both by simulation.
3. [extended] **Gradients through a coin flip.** ●● When x is a discrete choice there is no differentiable path from θ to the sample, so the pathwise estimator does not exist. Continuous relaxations such as Gumbel-softmax (Jang et al. 2017; Maddison et al. 2017) create a differentiable surrogate, and straight-through estimators (Bengio, Léonard, et al. 2013) use the hard sample in the forward pass and the surrogate in the backward pass; both are biased for the original objective. This problem measures that bias. Tucker et al. (2017) illustrate it on a toy problem. A single Bernoulli variable $b \in \{0, 1\}$ has $P(b = 1) = \sigma(\theta)$ for a logit θ, and the objective is $\ell(\theta) = \mathbb{E}[(b - t)^2]$ with target $t = 0.45$. Let U be uniform on $(0, 1)$, let $\zeta = \log U - \log(1 - U)$, and, for a temperature $\tau > 0$, let $z = \sigma\big((\theta + \zeta)/\tau\big)$, the two-class case of the Gumbel-softmax relaxation.
   1. Compute $\ell(\theta)$ and $\ell'(\theta)$ in closed form. Which probability $\sigma(\theta)$ minimizes ℓ, and does a finite θ attain it?
   2. Show that $b = \mathbf{1}(z > \frac{1}{2})$ satisfies $P(b = 1) = \sigma(\theta)$ for every τ.
   3. Implement three single-sample estimators of $\ell'(\theta)$: REINFORCE Equation 34.74 with the baseline $\ell(\theta)$; the relaxed estimator $\frac{d}{d\theta}(z - t)^2$; and the straight-through estimator $2(b - t)\frac{dz}{d\theta}$, which uses the hard sample in the loss and the relaxed sample in the derivative. Tabulate the mean and the standard deviation of each estimator against $\ell'(\theta)$ for $\theta \in \{-2, 0, 2\}$ and $\tau \in \{1, 0.5, 0.1\}$. How does the bias of each estimator depend on θ and τ?
   4. Run stochastic gradient descent on θ from $\theta = 0$ with step size 1 for 5000 steps with REINFORCE and with the relaxed and straight-through estimators at each τ, replacing the REINFORCE baseline by an exponential moving average of past losses. Plot $\ell(\theta)$ along each run. Compute the relaxed objective $R_\tau(\theta) = \mathbb{E}[(z - t)^2]$ by quadrature over U, and compare its minimizer with the values of θ at which the relaxed runs settle. How does lowering τ trade the bias of the relaxed estimator against its variance?

34.8 Further Reading

Throughout this chapter, we proved our main results under clean assumptions that deep learning practice routinely bends: a ReLU network is full of sharp kinks, a reverse-mode tape can easily outgrow GPU memory, and a thousand-dimensional integral cannot be placed on a grid. The short notes below summarize what happens when those assumptions break down, followed by a curated list of textbooks, courses, and tutorials covering the chapter's core material. Two topics bear directly on everyday training and are worth reading first: conservative fields, which explain in what sense backpropagation through kinks is still valid, and recursive checkpointing, which keeps memory manageable on long computations. Nothing elsewhere in the book depends on this section.

34.8.1 Derivatives at Kinks

34.8.1.1 Conservative Fields

As Section 34.1 illustrates, adopting the standard convention $\mathrm{ReLU}'(0) = 0$ leads to a curious anomaly: if we write the identity function as $\mathrm{ReLU}(x) - \mathrm{ReLU}(-x) = x$, backpropagation returns slope 0 at $x = 0$, a number that is neither the true derivative nor even a valid subgradient of x. Classical subdifferentials cannot explain this behavior, because their sum and chain rules hold only as set inclusions once the constituent functions are no longer convex. To close this gap, Bolte and Pauwels (2021) introduce *conservative fields*, set-valued generalizations of the derivative that obey the chain rule by construction and agree with the ordinary gradient everywhere outside a set of measure zero. Backpropagation with any fixed choice of slope at each kink always returns an element of such a field. And why is the exceptional set of kinks harmlessly small? By Rademacher's theorem (Rademacher 1919), every Lipschitz function on $\mathbb{R}^n$ is differentiable almost everywhere, so the kinks of a ReLU network's loss occupy a set of measure zero.

The same work settles the practical question of whether stochastic gradient descent driven by these backpropagation values still converges. For losses built from standard neural network layers, assuming bounded iterates and the usual conditions on step sizes and gradient noise, the iterates converge to the set of critical points where the conservative field contains zero. That guarantee holds in exact arithmetic, whereas in finite-precision floating point, iterates actually land on kinks. Section 34.1 reports how often this occurs across three numerical precisions, as measured by Bertoin et al. (2021) and corrected in their 2023 erratum, and shows how little the choice of $\mathrm{ReLU}'(0)$ ends up affecting training.

34.8.1.2 Subgradient Methods

Smooth optimization relies on the descent lemma, which guarantees that a small step against the gradient lowers the objective. Nonsmooth functions break that guarantee: at the minimum of $|x|$, the valid subgradient $\frac{1}{2}$ actually points uphill, as shown in Section 34.1. Remarkably, the subgradient method converges anyway, because its analysis tracks the Euclidean distance to a minimizer rather than the objective value itself. With diminishing step sizes satisfying $\sum_t \eta_t = \infty$ and $\sum_t \eta_t^2 < \infty$, the best objective value seen so far converges to the minimum. Obtaining the optimal convergence rate calls for a different schedule: for a convex function whose subgradients have norm at most G, starting at distance R from a minimizer, a constant step size $R/(G\sqrt{T})$ brings the best of T iterates within $RG/\sqrt{T}$ of the minimum. When the dimension exceeds T, no method relying solely on subgradients can improve on this rate by more than a constant factor (Nesterov 2018). For a systematic

development of subdifferential calculus, including exact conditions under which the sum rule holds with equality, see Rockafellar (1970) .

34.8.2 Derivatives of Vector and Matrix Maps

34.8.2.1 The Mean Value Theorem for Vector-Valued Maps

In one dimension, the mean value theorem of Section 34.1 guarantees an intermediate point ξ such that $f(b) - f(a) = f'(\xi)(b - a)$. It is tempting to assume the same equality holds for vector-valued curves $\gamma : \mathbb{R} \to \mathbb{R}^n$, but it fails completely. Take the unit circle $\gamma(t) = (\cos t, \sin t)$: after a full turn at $t = 2\pi$, the curve returns to its starting point so $\gamma(2\pi) - \gamma(0) = \mathbf{0}$, yet the velocity $\gamma'(\xi)$ has unit length and is never $\mathbf{0}$ for any ξ. What survives in higher dimensions is only the *mean value inequality*, $\|\gamma(b) - \gamma(a)\| \le \sup_{\xi} \|\gamma'(\xi)\| \, |b - a|$ (Theorem 5.19 of Rudin (1976)). Fortunately, the Taylor bounds in Section 34.2 never need a vector version of the theorem, because they restrict a scalar-valued loss to a line segment, where the ordinary one-variable theorem applies directly.

34.8.2.2 Jacobians of Matrix-Valued Maps

In Section 34.4, we differentiate scalar losses, so every gradient conveniently has the same shape as its matrix variable. Once we differentiate a matrix-valued function with respect to a matrix input, however, the full derivative becomes a four-index array. A standard way to keep the algebra linear and familiar is to flatten both matrices with the column-stacking operator vec, turning the four-index tensor into an ordinary Jacobian matrix. Under this flattening, a matrix differential $d\mathbf{F} = \mathbf{A}\, d\mathbf{X}\, \mathbf{B}$ becomes $d \operatorname{vec} \mathbf{F} = (\mathbf{B}^\top \otimes \mathbf{A})\, d \operatorname{vec} \mathbf{X}$, and the chain rule simply multiplies Kronecker products. Mike B. Giles (2008) provides a handy catalog of forward and reverse differentiation rules for matrix products, inverses, determinants, and standard matrix decompositions.

34.8.3 Automatic Differentiation

34.8.3.1 Optimal Jacobian Accumulation

As Section 34.4 shows, simply changing the order in which we multiply a chain of Jacobians can alter the computational cost by roughly a factor of the layer width. For a straight chain of K factors, finding the cheapest parenthesization is the classical matrix-chain multiplication problem, which dynamic programming solves in $O(K^3)$ time. A general computation graph with branches and skip connections opens up a much larger space of elimination orders: each intermediate node can be eliminated by multiplying the partial derivatives on its incoming and outgoing edges, in any sequence. Finding the elimination order with the fewest multiplications turns out to be NP-complete (Naumann 2008). Deep learning frameworks therefore do not attempt global optimization of the elimination order; instead, they provide forward mode, reverse mode, and their compositions, leaving the choice of mode to the user. Griewank and Walther (2008) develop vertex, edge, and face elimination in full generality and review the heuristics used by derivative compilers.

34.8.3.2 Recursive Checkpointing

The one-level checkpointing scheme in Section 34.5 reduces peak activation memory to $T/s + s$ stored states, reaching $2\sqrt{T}$ at the optimal block size s at the cost of one extra forward pass (Chen et al. 2016). For very long sequences or unrolled iterative solvers, even

$O(\sqrt{T})$ memory can be too much. Applying the same idea recursively inside each recomputed segment brings memory all the way down to $O(\log T)$ while requiring $O(T \log T)$ recomputation (Griewank 1992). When every step has the same cost, Griewank's binomial checkpointing schedule is provably optimal for any fixed memory budget (Griewank and Walther 2008). In practice, checkpointing utilities implement one level of checkpointing per marked block.

PyTorch provides `torch.utils.checkpoint`.

34.8.3.3 Taylor Mode

Computing a k-th directional derivative by nesting forward-mode automatic differentiation k times is wasteful, because each nesting doubles the state and carries 2^k components per value. *Taylor mode* avoids this exponential blow-up by replacing dual numbers with truncated power series $a_0 + a_1\varepsilon + \cdots + a_k\varepsilon^k$ subject to $\varepsilon^{k+1} = 0$. Each elementary operation propagates all $k+1$ coefficients in a single pass: multiplication convolves the two coefficient sequences in $O(k^2)$ operations, and transcendental functions such as exp, log, and sin follow from recurrences of the same $O(k^2)$ cost (Griewank and Walther 2008). The exercise on computing second derivatives in one forward pass in Section 34.5 is the $k = 2$ case.

34.8.3.4 Forward Gradients

Reverse-mode differentiation requires storing a tape of intermediate values, whereas forward mode avoids the tape but requires one pass per input parameter. Baydin et al. (2022) sidestep both costs by running a single forward pass along a random Gaussian tangent $\mathbf{v} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ and taking $(\nabla \ell^\top \mathbf{v})\,\mathbf{v}$ as a stochastic gradient estimate. Because $\mathbb{E}[\mathbf{v}\mathbf{v}^\top] = \mathbf{I}$, the estimate is unbiased, and for n parameters its mean squared error is $(n+1)\|\nabla \ell\|^2$. The memory tape disappears completely, but the trade-off is stark: the typical estimation error is roughly $\sqrt{n}$ times the norm of the true gradient, which is manageable for small models yet prohibitive at modern neural network scales.

34.8.4 Integration

34.8.4.1 The Lebesgue Integral

The Riemann integral partitions the domain into intervals, a strategy that breaks down on wildly discontinuous functions: as shown in Section 34.6, the indicator of the rational numbers has tagged sums that never settle, and even a pointwise limit of Riemann-integrable functions need not remain Riemann integrable. Lebesgue's insight is to slice the range instead of the domain: measure the size of the set where f exceeds each height level, and add up those measures. Sets of measure zero, such as the rational numbers or the ray $\theta = 0$ in polar coordinates, contribute nothing at all, so the indicator of the rationals cleanly integrates to 0. Both Section 34.6 and Section 34.7 invoke three workhorse theorems of this theory without proof: Fubini–Tonelli, the n-dimensional change of variables, and differentiation under an integral over an infinite domain. All three rest on the monotone and dominated convergence theorems, which specify when a limit may be swapped with an integral: if $f_k \to f$ pointwise and $|f_k| \le g$ for a single integrable function g, then $\int f_k \to \int f$ (Folland 1999).

34.8.4.2 Quasi-Monte Carlo

In Section 36.5.1.7, we compared a second-order grid, whose error decays as $N^{-2/d}$, with Monte Carlo integration, whose error decays as $N^{-1/2}$ regardless of dimension. Purely random samples, however, inevitably leave accidental clumps and empty gaps. *Low-discrepancy* point sets (such as Sobol sequences, Halton sequences, and lattice rules) are constructed to fill the unit cube far more evenly, achieving an error of $O(N^{-1}(\log N)^d)$ for integrands of bounded variation (in the sense of Hardy and Krause). Taken purely as a worst-case guarantee, this bound is sobering: ignoring constants, it drops below $N^{-1/2}$ only past several thousand points for $d = 2$ and past 10^{15} points for $d = 5$. In practice, quasi-Monte Carlo still frequently outperforms ordinary Monte Carlo even in high dimensions, usually because real integrands depend primarily on a small number of effective coordinates at a time. Because a deterministic point set provides no built-in error bar, *randomized quasi-Monte Carlo* randomly shifts or scrambles the points so that independent replicates yield an unbiased variance estimate. Owen (2013) covers both deterministic and randomized methods in Chapters 15–17.

34.8.4.3 Gradient Estimators beyond Score and Pathwise

In Section 34.7, we derived the score-function and pathwise estimators and reduced the variance of the score-function estimator using a constant baseline. More sophisticated control variates exploit the structure of the model itself: REBAR (Tucker et al. 2017) subtracts the pathwise gradient of a continuous relaxation from the score-function estimator and adds back its analytic expectation, lowering variance while keeping the estimator unbiased for discrete random variables. The continuous relaxations themselves, such as Gumbel-softmax (Jang et al. 2017; Maddison et al. 2017) and straight-through estimators (Bengio, Léonard, et al. 2013), offer low variance at the cost of introducing bias relative to the original discrete objective. A third approach, *measure-valued derivatives*, expresses the derivative of a probability density as a scaled difference of two densities and estimates the gradient by sampling from both. Mohamed et al. (2020) unite all three families in a single framework and spell out the conditions under which each applies.

34.8.5 Books, Courses, and Tutorials

Books

- Mathematics for Machine Learning by Deisenroth, Faisal, and Ong: Chapter 5 (“Vector Calculus”) covers gradients, Jacobians, and the chain rule for machine learning (Section 34.1 and Section 34.2).
- Calculus by Gilbert Strang (MIT OpenCourseWare): a free textbook whose chapters on integrals, the fundamental theorem, and multiple integrals accompany Section 34.6.
- The Matrix Cookbook by Petersen and Pedersen: a reference table of matrix-derivative identities useful for checking results derived by the differential method (Section 34.4).
- Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation by Griewank and Walther, and The Art of Differentiating Computer Programs by Naumann: the standard monographs on forward and reverse mode, VJPs and JVPs, checkpointing, and derivative compilers (Section 34.5).

Courses and Tutorials

- Essence of Calculus by 3Blue1Brown, Single Variable Calculus (18.01SC), and Multivariable Calculus (18.02SC) on MIT OpenCourseWare.

- Matrix Calculus for Machine Learning and Beyond (18.S096) by Edelman and Johnson (MIT OCW), The Matrix Calculus You Need For Deep Learning by Parr and Howard, and matrixcalculus.org by Laue, Mitterreiter, and Giesen.
- Calculus on Computational Graphs: Backpropagation by Christopher Olah, Automatic Differentiation in Machine Learning: a Survey by Baydin et al., The Autodiff Cookbook and Autodidax in the JAX documentation (Section 34.5), and Monte Carlo Gradient Estimation in Machine Learning by Mohamed et al. for the score-function (REINFORCE) estimator, pathwise reparameterization, and variance reduction with control variates (Section 34.7).

Chapter 35

Optimization

Every neural network in this book is trained by the same basic loop: compute the gradient of the loss, take a step in the opposite direction, and repeat. In Chapter 9, we saw empirically what momentum, adaptive step sizes, and learning-rate schedules do for that loop on real networks, leaving three questions open. Why is there a hard ceiling on the learning rate? Why can a step safely below that ceiling still make painfully slow progress? And why does gradient noise keep a training run from settling down?

Optimization theory ties each question to the geometry of the objective. The steepest curvature sets the largest safe step size; the ratio of the steepest curvature to the flattest sets the iteration count; and convexity decides whether a stationary point is globally optimal. Throughout this chapter, we return to the narrow bowl $\frac{1}{2}(x_1^2 + 1000\, x_2^2)$, which is a thousand times steeper in one direction than in the other: a step size that stays safe along the steep wall barely nudges the iterate along the flat valley. While Chapter 9 focuses on practice and measurement, this chapter supplies the self-contained proofs and underlying geometry.

The chapter unfolds in three parts. Under exact gradients, Section 35.1 proves the step-size ceiling and iteration count and shows how momentum replaces the curvature ratio with its square root, while Section 35.3 establishes when stationary points are global minima; between them, Section 35.2 covers Newton, quasi-Newton, and trust-region methods that actively model curvature. Turning to noisy gradients, Section 35.4 shows why fixed-step SGD stalls at a noise floor proportional to the step size, and Section 35.5 tests per-coordinate scaling and AdamW against heavy-ball momentum. Finally, Section 35.6 and Section 35.7 develop KKT conditions, projections, proximal steps, and duality behind norm budgets (Section 2.7), and Section 35.8 analyzes floating-point precision, cancellation, and conditioning (Section 5.5).

We assume familiarity with gradients and Hessians (Section 34.2) and eigendecompositions (Section 33.3). Because deep-network losses are neither globally smooth nor convex, our theorems serve as local models whose limits each section notes explicitly; interior-point and operator-splitting methods are surveyed in Section 35.9.

Notation

Throughout this chapter, we follow the conventions summarized in Table 35.1. Each section restates the symbols it uses most, and notation specific to a single section is introduced where it first appears.

Table 35.1: Symbols of the optimization chapter. A star $\star$ marks an optimum and an asterisk $*$ a convex conjugate. Matrices are bold.

Symbol	Meaning
f, $\ell(\mathbf{w})$	objective; training loss as a function of the weights
$\mathbf{x}^\star$, $f^\star$	a minimizer and the minimum value
∇f, $\nabla^2 f$, $\mathbf{g}_k$	gradient, Hessian; the gradient (or a stochastic gradient) used at step k
L, μ	smoothness and strong-convexity constants, $\mu\mathbf{I} \preceq \nabla^2 f \preceq L\mathbf{I}$
$\kappa = L/\mu$; $\kappa(\mathbf{A})$	condition number of the problem; of a matrix, $\sigma_{\max}(\mathbf{A})/\sigma_{\min}(\mathbf{A})$
λ_i, $\lambda_{\max}(\mathbf{A})$	eigenvalues of a Hessian or of a quadratic's matrix $\mathbf{A}$
η, β	step size (learning rate); momentum coefficient
k, t	iteration index; optimizer step in the adaptive methods
ϵ	target accuracy; in Adam, the constant that keeps a division finite
n, d, $\mathcal{B}$, b	number of examples; number of variables; a minibatch and its size
Σ	covariance of a per-example gradient
$\mathbf{m}_t$, $\mathbf{v}_t$; β_1, β_2	Adam's first- and second-moment estimates; their decay rates
$\mathbf{P}$	preconditioner: the step is $-\eta\,\mathbf{P}^{-1}\mathbf{g}$
g_i, h_j	inequality constraints $g_i(\mathbf{x}) \le 0$; equality constraints $h_j(\mathbf{x}) = 0$
λ, ν	multipliers of inequality and equality constraints; λ also weights the decay term $\frac{\lambda}{2}\lVert\mathbf{w}\rVert^2$
$\mathcal{L}$, q	Lagrangian; dual function
$p^\star$, $d^\star$	optimal values of the primal and the dual problem
f^*	convex conjugate of f
Π_C, prox_ψ	Euclidean projection onto a set C; proximal operator of ψ
u, $\varepsilon_{\text{mach}}$	unit roundoff and machine epsilon of a floating-point format, $u = \varepsilon_{\text{mach}}/2$

35.1 Gradient Descent and Curvature

In practice, a learning rate is usually chosen by trial and error: we pick a value, watch the loss, and turn it down if training blows up. Because every step moves opposite the gradient of the loss, the gradient fixes the direction and the learning rate sets only the distance. Yet a poor choice of distance turns training either painfully slow or unstable. How far can we trust a step along the gradient, and how many iterations does that limit force upon us?

A function of just two variables already exposes the difficulty. Consider the elliptical bowl $f(x, y) = \frac{1}{2}(x^2 + 10\,y^2)$, which curves ten times more steeply along y than along x, and start at $(2, 1)$, where $f = 7$ and $\nabla f = (2, 10)$. Taking a step of size η against the gradient lands at $(2 - 2\eta,\ 1 - 10\eta)$. With $\eta = 0.01$, the loss barely budges, dropping to 6.01. With $\eta = 0.1$, it falls sharply to 1.62. With $\eta = 0.25$, it *rises* to 12.4. Every one of those steps pointed downhill; only the step length changed.

The culprit is curvature. A step short enough to stay safe along the steep y-axis creeps along the flat x-axis. Even the best fixed learning rate needs 69 iterations to shrink the error by a factor of 10^6 (six digits of accuracy). Stretch the curvature ratio to 1000—our chapter's running example $\frac{1}{2}(x_1^2 + 1000\,x_2^2)$—and gradient descent needs $6{,}908$ steps. Adding momentum brings that count down to 315. We will work by hand on the gentler bowl, whose Hessian $\mathrm{diag}(1, 10)$ keeps every number easy to check, and quote the corresponding counts for the stiffer running example.

We build the analysis in three stages. First, we formalize smoothness to turn the local linear approximation of Section 34.2 into a guarantee on finite steps, yielding the descent lemma Equation 35.5. Next, we solve gradient descent exactly on quadratics, where the ratio of curvatures emerges as the condition number and fixes the optimal step Equation 35.10. Finally, we introduce momentum and derive its $\sqrt{\kappa}$ convergence law Equation 35.15. Throughout this section, our methods only respect curvature without modeling it; methods that model curvature directly, such as Newton's method, appear in Section 35.2. We also assume exact gradients here, leaving the noisy gradients of minibatch training to Section 35.4. All code below is plain NumPy, so each update can be read directly against its equation.

35.1.1 Descent Directions and Smoothness

Every gradient step settles two questions: which way to move, and how far. The first has an immediate answer from the first-order Taylor expansion. The second requires knowing how quickly the gradient itself can change—an assumption that turns a local linear approximation into a guarantee for steps of finite length.

35.1.1.1 The Steepest Direction

In our opening example, we stepped along $-\nabla f$ without asking whether another direction might be better. To see why the negative gradient is the natural choice, expand $f : \mathbb{R}^d \to \mathbb{R}$ to first order. A small step of size $\eta > 0$ along a direction $\mathbf{d}$ changes the function value by

$$f(\mathbf{x} + \eta\,\mathbf{d}) = f(\mathbf{x}) + \eta\,\nabla f(\mathbf{x})^\top \mathbf{d} + o(\eta).$$

To first order, the change is simply η times the directional derivative $\nabla f(\mathbf{x})^\top \mathbf{d}$. We call $\mathbf{d}$ a **descent direction** at $\mathbf{x}$ whenever $\nabla f(\mathbf{x})^\top \mathbf{d} < 0$.

Proposition (descent directions, and the steepest one). *Let f be differentiable at $\mathbf{x}$ with $\nabla f(\mathbf{x}) \neq \mathbf{0}$.*

1. *If $\nabla f(\mathbf{x})^\top \mathbf{d} < 0$, then $f(\mathbf{x} + \eta\,\mathbf{d}) < f(\mathbf{x})$ for every sufficiently small $\eta > 0$.*
2. *Among all unit vectors, the directional derivative is minimized uniquely by the normalized negative gradient:*

$$\min_{\|\mathbf{d}\|=1} \nabla f(\mathbf{x})^\top \mathbf{d} = -\|\nabla f(\mathbf{x})\|, \qquad \text{attained only at } \mathbf{d} = -\frac{\nabla f(\mathbf{x})}{\|\nabla f(\mathbf{x})\|}. \tag{35.1}$$

Proof. For the first claim, define $\varphi(\eta) = f(\mathbf{x}+\eta\,\mathbf{d})$. By the chain rule, $\varphi'(0) = \nabla f(\mathbf{x})^\top \mathbf{d} < 0$, so the difference quotient $(\varphi(\eta) - \varphi(0))/\eta$ is strictly negative for all sufficiently small $\eta > 0$; hence $\varphi(\eta) < \varphi(0)$. For the second claim, the Cauchy–Schwarz inequality gives $\nabla f(\mathbf{x})^\top \mathbf{d} \geq -\|\nabla f(\mathbf{x})\|\,\|\mathbf{d}\| = -\|\nabla f(\mathbf{x})\|$ for every unit vector $\mathbf{d}$, with equality if and only if $\mathbf{d}$ is a negative scalar multiple of $\nabla f(\mathbf{x})$. The unique such unit vector is $-\nabla f(\mathbf{x})/\|\nabla f(\mathbf{x})\|$, and it attains the bound. ■

On our opening bowl, the steepest unit direction at $(2, 1)$ is $-(2, 10)/\sqrt{104} = (-0.196, -0.981)$, with slope $-\|\nabla f\| = -10.198$. The coordinate direction $(-1, 0)$ also points downhill, but with a much gentler slope of -2. In fact, descent directions fill the entire open half-space of vectors making an acute angle with $-\nabla f$; the negative gradient simply picks out the steepest one. Because this result governs slopes (the limit of vanishingly short steps), we next check how faithfully those slopes predict the actual decrease over a small finite step, $\eta = 0.01$, across four unit directions.

```
def f(v):
    return 0.5 * (v[0]**2 + 10 * v[1]**2)

def grad(v):
    return np.array([v[0], 10 * v[1]])

x = np.array([2.0, 1.0])
g = grad(x)
steepest = -g / np.linalg.norm(g)
candidates = [('steepest -grad/|grad|', steepest),
              ('diagonal -(1,1)/sqrt(2)', -np.ones(2) / np.sqrt(2)),
              ('axis     (-1,0)', np.array([-1.0, 0.0])),
              ('uphill   +(1,1)/sqrt(2)', np.ones(2) / np.sqrt(2))]
eta = 0.01
for name, d in candidates:
    print(f'{name}:  slope = {g @ d:+8.4f},  '
          f'f(x + eta d) - f(x) = {f(x + eta * d) - f(x):+.6f}')
```

```
steepest -grad/|grad|:  slope = -10.1980,  f(x + eta d) - f(x) = -0.101498
diagonal -(1,1)/sqrt(2):  slope =  -8.4853,  f(x + eta d) - f(x) = -0.084578
axis     (-1,0):  slope =  -2.0000,  f(x + eta d) - f(x) = -0.019950
uphill   +(1,1)/sqrt(2):  slope =  +8.4853,  f(x + eta d) - f(x) = +0.085128
```

At $\eta = 0.01$, each actual change matches η times the directional slope to within one percent, so the first-order model is a reliable guide for short steps. Differentiability promises nothing beyond that: a decrease for sufficiently small steps. How small is "sufficiently small" is the question we turn to next.

35.1.1.2 Smoothness and the Descent Lemma

Gradient descent repeats the steepest-descent step with a fixed **step size** (or learning rate) $\eta > 0$:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \eta \nabla f(\mathbf{x}_k). \tag{35.2}$$

Throughout this section, k indexes iterations and we abbreviate $\mathbf{g}_k = \nabla f(\mathbf{x}_k)$. The first-order Taylor expansion justified the direction $-\mathbf{g}_k$, but it says nothing about how far to trust it: once the iterate moves far enough that the gradient changes significantly, $\mathbf{g}_k$ no longer describes f. To certify a finite step size, we need a bound on how fast the gradient can vary.

Definition (L-smoothness). A differentiable function f is **L-smooth** if its gradient changes by at most L per unit of distance:

$$\|\nabla f(\mathbf{x}) - \nabla f(\mathbf{y})\| \;\le\; L\,\|\mathbf{x} - \mathbf{y}\| \qquad \text{for all } \mathbf{x}, \mathbf{y}. \tag{35.3}$$

When f is twice continuously differentiable, L-smoothness is equivalent to requiring every eigenvalue of its Hessian to lie in $[-L, L]$. Bounded eigenvalues imply Equation 35.3 by integrating the Hessian along the line segment from $\mathbf{y}$ to $\mathbf{x}$, and the converse follows by taking a difference-quotient limit (exercise 2a). In short, L caps the curvature in every direction. Our opening bowl has constant Hessian $\operatorname{diag}(1, 10)$, so $L = 10$. By contrast, a function whose curvature grows without bound has no global smoothness constant: $\tfrac{1}{4}x^4$ has $f''(x) = 3x^2$, so no finite global L exists. We will handle that case shortly using backtracking line search.

Smoothness upgrades the local tangent plane into a global upper bound—a quadratic ceiling:

$$f(\mathbf{y}) \;\le\; f(\mathbf{x}) + \nabla f(\mathbf{x})^{\top}(\mathbf{y} - \mathbf{x}) + \tfrac{L}{2}\,\|\mathbf{y} - \mathbf{x}\|^2. \tag{35.4}$$

Geometrically, the graph of f can never rise above its tangent plane at $\mathbf{x}$ by more than $\tfrac{L}{2}$ times the squared distance. Before we prove this inequality, plug in a gradient step $\mathbf{y} = \mathbf{x} - \eta\,\mathbf{g}$ and watch the two terms compete. The linear first-order decrease $\eta\,\|\mathbf{g}\|^2$ grows proportionally to η, whereas the quadratic curvature penalty $\tfrac{L}{2}\eta^2\|\mathbf{g}\|^2$ grows as η^2. For small η, the linear decrease wins; at $\eta = 2/L$, the two terms exactly cancel.

Proposition (descent lemma). *If f is L-smooth, then the gradient step Equation 35.2 satisfies*

$$f(\mathbf{x}_{k+1}) \;\le\; f(\mathbf{x}_k) - \eta\left(1 - \tfrac{L\eta}{2}\right)\|\mathbf{g}_k\|^2. \tag{35.5}$$

In particular, f strictly decreases whenever $0 < \eta < 2/L$ and $\mathbf{g}_k \neq \mathbf{0}$. The guaranteed decrease is maximized at $\eta = 1/L$, where $f(\mathbf{x}_{k+1}) \le f(\mathbf{x}_k) - \tfrac{1}{2L}\|\mathbf{g}_k\|^2$.

Proof. We first establish the quadratic ceiling Equation 35.4. Restrict f to the line segment between $\mathbf{x}$ and $\mathbf{y}$ via $\psi(t) = f(\mathbf{x} + t(\mathbf{y} - \mathbf{x}))$ for $t \in [0, 1]$, so that $\psi'(t) = \nabla f(\mathbf{x} + t(\mathbf{y} - \mathbf{x}))^{\top}(\mathbf{y} - \mathbf{x})$. Applying the fundamental theorem of calculus, Cauchy–Schwarz, and the Lipschitz condition Equation 35.3 yields

$$
\begin{aligned}
f(\mathbf{y}) - f(\mathbf{x}) - \nabla f(\mathbf{x})^{\top}(\mathbf{y}-\mathbf{x}) &\\
&= \int_0^1 \big(\nabla f(\mathbf{x} + t(\mathbf{y}-\mathbf{x})) - \nabla f(\mathbf{x})\big)^{\top}(\mathbf{y}-\mathbf{x})\, dt \\
&\le \int_0^1 L\, t\, \|\mathbf{y}-\mathbf{x}\|^2\, dt = \tfrac{L}{2}\|\mathbf{y}-\mathbf{x}\|^2.
\end{aligned}
$$

Substituting the gradient step $\mathbf{y} = \mathbf{x}_k - \eta\,\mathbf{g}_k$ gives

$$
f(\mathbf{x}_{k+1}) \;\le\; f(\mathbf{x}_k) - \eta\,\|\mathbf{g}_k\|^2 + \tfrac{L}{2}\,\eta^2\,\|\mathbf{g}_k\|^2.
$$

The factor $\eta(1 - L\eta/2)$ is strictly positive if and only if $0 < \eta < 2/L$, and it reaches its maximum at $\eta = 1/L$. ■

Notice that the proof only compares gradients between $\mathbf{x}_k$ and points along the segment to $\mathbf{x}_{k+1}$, so it suffices for ∇f to be L-Lipschitz on that segment. Let us test the lemma on our opening bowl, where $\|\mathbf{g}\|^2 = 104$ and $L = 10$. At $\eta = 1/L = 0.1$, the bound guarantees $f \le 7 - 104/20 = 1.8$. At $\eta = 0.01$, it guarantees $f \le 7 - 104 \cdot 0.01 \cdot 0.95 = 6.012$. At $\eta = 0.25 > 2/L$, the factor $1 - L\eta/2$ turns negative and the bound rises above the starting value, offering no guarantee at all. How tight are these bounds compared with the values the three steps actually reach?

```
x, L = np.array([2.0, 1.0]), 10.0
g = grad(x)
for eta in (0.01, 0.1, 0.25):
    bound = f(x) - eta * (1 - L * eta / 2) * (g @ g)
    print(f'eta = {eta:4.2f}:  bound {bound:7.4f},  actual {f(x - eta * g):7.4f}')
```

```
eta = 0.01:  bound  6.0120,  actual  6.0102
eta = 0.10:  bound  1.8000,  actual  1.6200
eta = 0.25:  bound 13.5000,  actual 12.3750
```

For the two stable steps, the bounds are remarkably tight: 6.0102 lies just under 6.012, and 1.62 sits cleanly below 1.8. At $\eta = 0.25$, the bound 13.5 lies well above the starting value $f = 7$, and indeed the step overshoots to 12.375. Figure 35.1 plots both curves as functions of η. Along the search direction, $\mathbf{g}^{\top}\mathrm{diag}(1, 10)\,\mathbf{g} = 1004$, so the true objective is the parabola $7 - 104\,\eta + 502\,\eta^2$, while the quadratic ceiling is $7 - 104\,\eta + 520\,\eta^2$. The two parabolas hug each other so closely because this gradient points almost along the stiff axis, where the directional curvature is nearly L.

In short, the step $\eta = 1/L$ minimizes the quadratic ceiling, and f is guaranteed to lie beneath that ceiling. Section 34.1 develops the same geometric argument in one variable.

35.1.1.3 Stationary Points Without Convexity

Surprisingly, the descent lemma alone, without any convexity assumption, already places a quantitative limit on a long run: the gradient cannot stay large forever.

Proposition (gradient descent finds approximate stationary points). *Let f be L-smooth and bounded below by f_{low}, and run Equation 35.2 with $\eta = 1/L$. Then for every $K \ge 1$,*

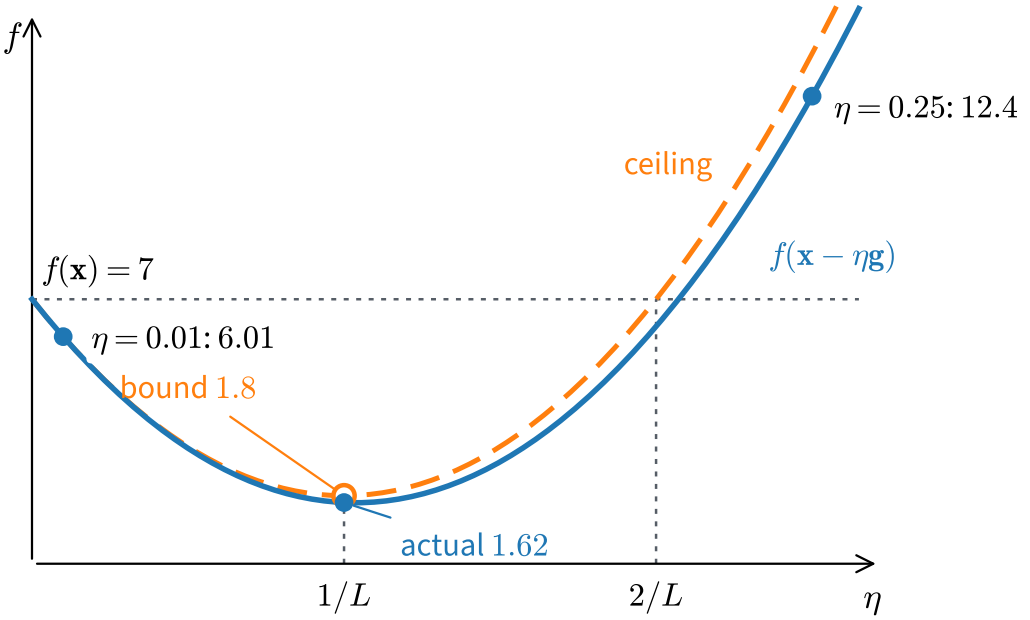


Figure 35.1: The descent lemma on the opening example. Along $-\mathbf{g}$ from $(2, 1)$, the true objective is the lower parabola and the quadratic ceiling $f - \eta\|\mathbf{g}\|^2 + \frac{L}{2}\eta^2\|\mathbf{g}\|^2$ is the upper one, tangent at $\eta = 0$. Minimizing the ceiling at $\eta = 1/L = 0.1$ guarantees $f \le 1.8$, and the step achieves 1.62. Beyond $2/L = 0.2$ the ceiling rises above the starting value and promises nothing; at $\eta = 0.25$ the objective climbs to 12.4.

$$\min_{0\le k<K} \|\nabla f(\mathbf{x}_k)\|^2 \;\le\; \frac{2L\,(f(\mathbf{x}_0) - f_{\text{low}})}{K}. \tag{35.6}$$

Proof. At $\eta = 1/L$, the descent lemma rearranges to $\|\mathbf{g}_k\|^2 \le 2L\,(f(\mathbf{x}_k) - f(\mathbf{x}_{k+1}))$. Summing over $k = 0, \dots, K-1$, the right-hand side telescopes to $2L\,(f(\mathbf{x}_0) - f(\mathbf{x}_K)) \le 2L\,(f(\mathbf{x}_0) - f_{\text{low}})$. Because the minimum of K numbers is at most their average, dividing by K gives the bound. ■

Thus, among the first K iterates, at least one has squared gradient norm $O(1/K)$, using nothing beyond a lower bound on f and a smoothness constant. Keep in mind that this result assumes a global L; ReLU networks are not smooth, so for them the bound serves as an idealized model rather than a formal guarantee.

A small gradient, however, does not guarantee a minimum. Consider $f(x, y) = \frac{1}{2}x^2 + \cos y$, which is bounded below by -1 and 1-smooth (its Hessian $\operatorname{diag}(1, -\cos y)$ has eigenvalues in $[-1, 1]$). Starting gradient descent with $\eta = 1$ at $(1, 0)$ drives x to 0 in a single step and never changes y, since $\partial f/\partial y = -\sin y$ vanishes at $y = 0$. The algorithm halts at $(0, 0)$ with $f = 1$—a saddle point where f curves upward along x and downward along y, far from the true minimum value -1 at $(0, \pi)$. The theorem still holds, because the gradient vanishes at the saddle, yet it cannot tell us which kind of stationary point we reached. The stall itself is an accident of the starting point. Start at $(1, y_0)$ with any small $y_0 \neq 0$ instead, and each step $y \leftarrow y + \eta \sin y$ moves y further from 0 until the run settles at a minimum. In general, the starting points from which gradient descent converges to a saddle with a direction of negative curvature form a set of measure zero (Section 9.1.2), so a random initialization avoids it, although the iterates can linger nearby for a long time. For convex objectives, the question does not arise: every stationary point is automatically a global minimum (Section 35.3).

35.1.1.4 Backtracking Line Search

Using $\eta = 1/L$ presupposes that we know L. In practice we rarely do, and even when a global L exists, the local curvature varies across the loss landscape, making a single global constant overly conservative. A **line search** adapts the step size at run time. Given a candidate step η, it accepts the step only if the actual decrease achieves at least a fixed fraction $c \in (0, 1)$ of the decrease predicted by the first-order model—the **Armijo condition** (or sufficient decrease condition) (Armijo 1966):

$$f(\mathbf{x} - \eta\,\nabla f(\mathbf{x})) \;\le\; f(\mathbf{x}) - c\,\eta\,\|\nabla f(\mathbf{x})\|^2. \tag{35.7}$$

The right-hand side is simply the linear Taylor prediction with its slope scaled down by c. **Backtracking** starts from an optimistic trial step η_0 and repeatedly halves η until Equation 35.7 is satisfied. By the descent lemma, the test is guaranteed to pass as soon as $1 - L\eta/2 \ge c$, or equivalently $\eta \le 2(1-c)/L$. Consequently, the loop always terminates in finitely many halvings and returns a step $\eta \ge \min\,(\eta_0,\, (1-c)/L)$—within a factor of two of the theoretical threshold—without ever needing to know L in advance.

Better still, this guarantee only requires L on the region the iterates actually visit. Take the quartic $f(x) = \frac{1}{4}x^4$, which has no global L, starting at $x_0 = 3$ with $c = \frac{1}{2}$. Because backtracking never increases f, every iterate remains in $[-3, 3]$, where $f''(x) = 3x^2 \le 27$. A trial step with $\eta \le 1/27$ sends x to $x(1 - \eta x^2)$, which lies between $\frac{2}{3}x$ and x, keeping the

entire segment inside $[-3, 3]$ so that $L = 27$ suffices locally. Since the quartic's curvature is steep far from the origin and vanishes near zero, no single fixed step size suits the entire trajectory. Let us compare both strategies from $x_0 = 3$: a fixed step $\eta = 0.3$ (which would be completely safe near the origin) and backtracking starting from $\eta_0 = 1$.

```
quartic = lambda v: 0.25 * (v ** 4).sum()
quartic_grad = lambda v: v ** 3

x = np.array([3.0])
for _ in range(6):                          # fixed step, too large at x0
    x = x - 0.3 * quartic_grad(x)
print(f'fixed eta = 0.3 from x0 = 3:  |x_6| = {abs(x[0]):.2e}  (diverged)')

x, etas = np.array([3.0]), []
for _ in range(25):                         # backtracking line search
    g = quartic_grad(x)
    eta = 1.0
    while quartic(x - eta * g) > quartic(x) - 0.5 * eta * (g * g).sum():
        eta *= 0.5                          # Armijo fails: halve the step
    x = x - eta * g
    etas.append(eta)
print('accepted steps:', np.array(etas[:8]), '...')
print(f'backtracking from x0 = 3:  f(x_25) = {quartic(x):.2e}  (monotone descent)')
```

```
fixed eta = 0.3 from x0 = 3:  |x_6| = 6.52e+103  (diverged)
accepted steps: [0.03125 0.0625  0.125   0.25    0.5     1.      1.      1.     ] ...
backtracking from x0 = 3:  f(x_25) = 1.09e-04  (monotone descent)
```

The fixed step explodes immediately: after just six iterations, $|x_6| \approx 6.5 \times 10^{103}$. Because the update $x \mapsto x(1 - \eta x^2)$ shrinks $|x|$ if and only if $\eta x^2 < 2$, a fixed step from $x_0 = 3$ converges for $\eta < 2/9 \approx 0.22$ and diverges above it. Notice that this stability threshold is $2/9$, not $2/f''(x_0) = 2/27$; the $2/f''(x_0)$ rule belongs to quadratics. A step toward the origin experiences the average of f'' between 0 and x (which is x^2) rather than its peak value $3x^2$. The descent lemma must guard against the worst-case curvature along the segment and therefore certifies only a third of the true safe range.

Backtracking, by contrast, accepts 0.031 on the first step. Its previous trial, 0.0625, fails the Armijo test, which on this quartic holds if and only if $\eta x^2 \leq 0.456$ (roughly $\eta \leq 0.05$ at $x_0 = 3$). As the iterate moves toward the origin and the curvature flattens, the accepted steps grow automatically ($0.0625, 0.125, 0.25, 0.5, 1.0$, each close to $0.3/x^2$) because the line search probes the local curvature on the fly. Deep learning training rarely uses backtracking, since a minibatch loss is too noisy to compare two candidate steps reliably and each trial costs an extra forward pass. Instead, training usually specifies the step size in advance via a schedule (Section 9.7).

35.1.1.5 Relaxed Smoothness and Gradient Clipping

A schedule cannot react when the iterate enters a region of high curvature. The safeguard that training uses for this is gradient clipping, and the quartic shows why it works. The curvature $f''(x) = 3x^2$ is unbounded, but it is large only where the gradient $f'(x) = x^3$ is large as well: $3x^2 \leq 4 + |x|^3$ for every x. A twice differentiable function is (L_0, L_1)-**smooth** if its curvature is bounded in this way by its gradient (Zhang, He, et al. 2020):

$$\|\nabla^2 f(\mathbf{x})\| \;\le\; L_0 + L_1 \|\nabla f(\mathbf{x})\| \qquad \text{for all } \mathbf{x}, \tag{35.8}$$

where the norm on the left is the largest absolute eigenvalue of the Hessian. Setting $L_1 = 0$ recovers L-smoothness, and the quartic qualifies with $L_0 = 4$ and $L_1 = 1$. Under this condition, the step that the descent lemma can certify is no longer a constant $1/L$. It is of order $1/(L_0 + L_1\|\mathbf{g}\|)$: a fixed step where the gradient is small, and a fixed *distance* $\eta\,\|\mathbf{g}\|$ of order $1/L_1$ where it is large. Gradient clipping, which replaces $\mathbf{g}$ by $\min(1, c/\|\mathbf{g}\|)\,\mathbf{g}$ for a threshold c, takes steps of this form at no extra cost. On the quartic with $c = 1$, the step $\eta = 0.3$ that diverged above now moves x by 0.3 per iteration until $|x| \le 1$ and converges from there (Exercise 3).

35.1.2 The Quadratic Model and the Condition Number

In the opening, we promised to explain why one bowl takes 69 steps and another takes $6{,}908$. Both objectives are quadratics, and on a quadratic we can solve gradient descent in closed form. Remarkably, the iteration count boils down to a single quantity: the ratio of the largest curvature to the smallest.

35.1.2.1 Ill-Conditioned Valleys

Before writing any algebra, look at Figure 35.2, which contrasts gradient descent on two quadratics. On a round bowl, the iterates march almost straight to the minimum. In an elongated valley, the walls are steep while the floor is flat: any step near the stability ceiling imposed by the walls overshoots across the valley and bounces from wall to wall, while barely inching along the floor. The result is the classic zigzag trajectory. (The valley shown uses $\mathrm{diag}(1, 20)$ with $\eta = 0.097$, just below its stability ceiling 0.1, producing the contraction factors noted in the caption.)

Quadratics deserve a complete analysis because every smooth function looks like one near a strict minimum. Around a minimizer $\mathbf{x}^\star$ with positive-definite Hessian $\mathbf{A}$, the second-order Taylor expansion approximates f by $f(\mathbf{x}^\star) + \frac{1}{2}(\mathbf{x} - \mathbf{x}^\star)^\top \mathbf{A}\,(\mathbf{x} - \mathbf{x}^\star)$. Shifting the minimizer to the origin and dropping the additive constant leaves our model problem:

$$f(\mathbf{x}) = \tfrac{1}{2}\,\mathbf{x}^\top \mathbf{A}\,\mathbf{x},$$
$$\mathbf{A} \text{ symmetric positive definite, with eigenvalues } 0 < \lambda_{\min} = \lambda_1 \le \cdots \le \lambda_d = \lambda_{\max}.$$

Its gradient is $\mathbf{A}\mathbf{x}$ and its Hessian is $\mathbf{A}$, so its smoothness constant is simply $L = \lambda_{\max}$. The ratio of the largest curvature to the smallest,

$$\kappa = \frac{\lambda_{\max}}{\lambda_{\min}},$$

is the **condition number** of the problem. It equals 10 for our opening bowl $\mathrm{diag}(1, 10)$ and 10^3 for the chapter's running example. Because $\mathbf{A}$ is symmetric positive definite, κ coincides with the singular-value ratio $\kappa(\mathbf{A})$ introduced in Section 33.5.3.2.

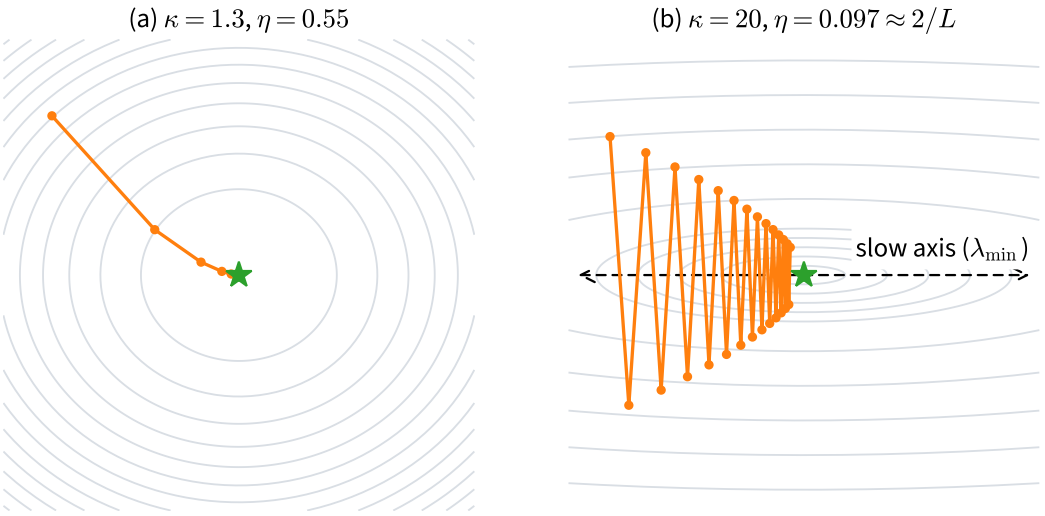


Figure 35.2: Gradient descent on two quadratics $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{A}\mathbf{x}$. Left: on a well-conditioned bowl, the iterates head nearly straight to the minimum. Right: in an ill-conditioned valley, a step near the stability ceiling makes the steep mode overshoot and alternate (factor -0.94) while the slow axis contracts by only 0.90 per step.

35.1.2.2 Per-Mode Contraction and the Stability Ceiling

The zigzagging in Figure 35.2 comes directly from the step-size ceiling enforced by the steepest direction. To pinpoint that ceiling and see how severely it slows down the flat direction, let us solve gradient descent on a quadratic exactly—starting, as always, with a single variable.

Take $f(u) = \frac{\lambda}{2}u^2$ with $\lambda > 0$. One gradient step sends u_k to $u_{k+1} = u_k - \eta\lambda u_k$, so

$$u_{k+1} = (1 - \eta\lambda)\, u_k, \qquad u_k = (1 - \eta\lambda)^k\, u_0.$$

Every iteration simply multiplies the error by the same scalar factor $1 - \eta\lambda$. Convergence occurs if and only if $|1 - \eta\lambda| < 1$, meaning $0 < \eta < 2/\lambda$. At the sweet spot $\eta = 1/\lambda$, the iterate lands on 0 in a single step. For $1/\lambda < \eta < 2/\lambda$, the factor is negative, so u_k flips sign at every step while shrinking toward zero. For $\lambda = 10$, for instance, $\eta = 0.1$ gives factor 0, $\eta = 0.19$ gives -0.9, and $\eta = 0.21$ gives -1.1, which diverges.

The d-dimensional quadratic keeps the exact same structure. Since $\nabla f(\mathbf{x}) = \mathbf{A}\mathbf{x}$, the update Equation 35.2 becomes

$$\mathbf{x}_{k+1} = (\mathbf{I} - \eta\mathbf{A})\, \mathbf{x}_k.$$

Each step applies the same linear operator, so after k steps $\mathbf{x}_k = (\mathbf{I} - \eta\mathbf{A})^k \mathbf{x}_0$. Because $\mathbf{A}$ is symmetric, it admits an orthonormal basis of eigenvectors (Section 33.3). Express $\mathbf{x}_k$ in this eigenbasis with coordinates $u_i^{(k)}$. The matrix $\mathbf{I} - \eta\mathbf{A}$ shares the same eigenvectors with eigenvalues $1 - \eta\lambda_i$, so the coordinates decouple completely and evolve independently:

$$u_i^{(k+1)} = (1 - \eta\lambda_i)\, u_i^{(k)}, \qquad \text{hence} \qquad u_i^{(k)} = (1 - \eta\lambda_i)^k\, u_i^{(0)}. \tag{35.9}$$

Thus, gradient descent on a d-dimensional quadratic is nothing more than d independent copies of the one-variable recursion—one per eigenvalue, or **mode**. Two consequences follow immediately from the factors $1 - \eta\lambda_i$:

- **Convergence** requires $|1 - \eta\lambda_i| < 1$ for every mode, which reduces to $0 < \eta < 2/\lambda_{\max} = 2/L$. This threshold is the **stability ceiling**. On quadratics, the bound $2/L$ that the descent lemma guaranteed as sufficient is also strictly necessary. Push η past it, and the stiffest mode's factor drops below -1: that coordinate oscillates with growing amplitude even while all other modes decay cleanly.
- **Speed** is governed by the slowest mode. The error norm decays as $\rho(\eta)^k$, where $\rho(\eta) = \max_i |1 - \eta\lambda_i|$ is the **spectral radius** of $\mathbf{I} - \eta\mathbf{A}$—the largest eigenvalue magnitude.

On $\mathbf{A} = \mathrm{diag}(1, 10)$, the stability ceiling is $\eta = 0.2$. Before running the sweep below, predict which step size leaves the smallest error after 60 steps from $(1, 1)$: the cautious 0.02, the descent-lemma favorite $0.1 = 1/L$, or something closer to the ceiling?

```
A = np.diag([1.0, 10.0])                    # eigenvalues 1 and 10, so L = 10
x0 = np.array([1.0, 1.0])

def run_gd(A, x0, eta, steps):
    xs = [x0]
    for _ in range(steps):
        xs.append(xs[-1] - eta * (A @ xs[-1]))
    return np.array(xs)

print(' eta    |1-eta*1|  |1-eta*10|  spectral radius   |x_60|')
for eta in [0.02, 0.10, 2 / 11, 0.19, 0.20, 0.21]:
    f1, f10 = abs(1 - eta * 1.0), abs(1 - eta * 10.0)
    print(f'{eta:5.3f}    {f1:6.3f}     {f10:6.3f}        {max(f1, f10):6.3f}     '
          f'{np.linalg.norm(run_gd(A, x0, eta, 60)[-1]):9.1e}')
```

```
 eta    |1-eta*1|  |1-eta*10|  spectral radius   |x_60|
0.020     0.980      0.800         0.980       3.0e-01
0.100     0.900      0.000         0.900       1.8e-03
0.182     0.818      0.818         0.818       8.3e-06
0.190     0.810      0.900         0.900       1.8e-03
0.200     0.800      1.000         1.000       1.0e+00
0.210     0.790      1.100         1.100       3.0e+02
```

The cautious $\eta = 0.02$ is safe but sluggish ($\rho = 0.98$, leaving an error of 0.3 after 60 steps). At $\eta = 0.1$, the stiff mode vanishes in a single step, yet the slow mode still limits the overall rate to $\rho = 0.9$. Interestingly, $\eta = 0.19$ shares that exact same rate $\rho = 0.9$, for the opposite reason: now the stiff mode oscillates with factor -0.9. Between them, $\eta = 2/11 \approx 0.182$ balances both extremes and beats every other choice by two orders of magnitude ($\rho = 0.818$, error 8.3×10^{-6}). Right at the ceiling $\eta = 0.2$, the stiff mode has factor -1, flipping between ± 1 forever so $\|\mathbf{x}_{60}\| = 1.0$. Step just past it to $\eta = 0.21$, and sixty iterations inflate the error to 3×10^2.

35.1.2.3 The Optimal Step

Within the stable range, which fixed step size η converges fastest? The sweep points directly to the answer: balance the two extreme modes. For any fixed η, the map $\lambda \mapsto |1-\eta\lambda|$ traces a V-shape with its vertex at $\lambda = 1/\eta$. On the interval $[\lambda_{\min}, \lambda_{\max}]$, a V-shaped function attains its maximum at one of the endpoints, so $\rho(\eta) = \max(|1-\eta\lambda_{\min}|, |1-\eta\lambda_{\max}|)$. Once η passes $1/\lambda_{\max}$, increasing it further helps the slow mode at $\lambda_{\min}$ while hurting the stiff mode at $\lambda_{\max}$.

Proposition (optimal fixed step on a quadratic). *For $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{A}\mathbf{x}$, the convergence factor $\rho(\eta) = \max_i |1-\eta\lambda_i|$ is minimized by*

$$\eta^\star = \frac{2}{\lambda_{\min}+\lambda_{\max}}, \qquad \rho(\eta^\star) = \frac{\kappa-1}{\kappa+1}, \qquad \kappa = \frac{\lambda_{\max}}{\lambda_{\min}}. \tag{35.10}$$

Proof. For every η, we have $\rho(\eta) \geq \max(1-\eta\lambda_{\min},\ \eta\lambda_{\max}-1)$, and the maximum of two numbers is at least any convex combination of them. Choosing the weights $\lambda_{\max}/(\lambda_{\min}+\lambda_{\max})$ and $\lambda_{\min}/(\lambda_{\min}+\lambda_{\max})$ makes the η-terms cancel cleanly:

$$\rho(\eta) \ \geq\ \frac{\lambda_{\max}(1-\eta\lambda_{\min}) + \lambda_{\min}(\eta\lambda_{\max}-1)}{\lambda_{\min}+\lambda_{\max}} = \frac{\lambda_{\max}-\lambda_{\min}}{\lambda_{\max}+\lambda_{\min}} = \frac{\kappa-1}{\kappa+1}.$$

Equality holds only when the two terms agree, $1-\eta\lambda_{\min} = \eta\lambda_{\max}-1$, which solves to $\eta = \eta^\star$. At $\eta^\star$, both endpoints equal $(\kappa-1)/(\kappa+1)$, and the V-shape keeps every interior eigenvalue below that height, so the bound is attained. ■

Figure 35.3 illustrates this proof on $\mathrm{diag}(1, 10)$: the optimal V is the one whose two endpoints sit at the same height. Here $\eta^\star = 2/11$ and $\rho = 9/11 = 0.818$, so gaining six digits takes $\log 10^6/\log(11/9) \approx 69$ steps. On the chapter's running example ($\kappa = 1000$), $\rho = 999/1001$, and those same six digits demand $6{,}908$ steps.

When $\kappa = 1$ (a perfectly round bowl), $\rho = 0$ and a single step lands on the minimum. For large κ, $(\kappa-1)/(\kappa+1) \approx 1-2/\kappa$, so $\log(1/\rho) \approx 2/\kappa$, and shrinking the error by a factor of ϵ takes roughly $\frac{\kappa}{2}\log\frac{1}{\epsilon}$ iterations: **the iteration count of gradient descent grows linearly with the condition number**. The bottleneck is easy to see in one sentence: stability forces the step to obey the stiffest mode, $\eta \lesssim 2/\lambda_{\max}$, leaving the flattest mode to advance by only $\eta\,\lambda_{\min} \approx 2/\kappa$ per iteration.

At $\eta^\star$, the factor ρ is even stronger than an asymptotic rate. On $\mathrm{diag}(1, 10)$, the two per-mode factors are $+9/11$ and $-9/11$ (the $\eta = 2/11$ row of the sweep): the slow mode contracts monotonically while the stiff mode alternates in sign. Because both coordinates shrink by the same magnitude, $\|\mathbf{x}_k\|$ contracts by $(\kappa-1)/(\kappa+1)$ on every single step, not just on average.

35.1.2.4 The Edge of Stability

The stability ceiling suggests a simple recipe for neural networks: compute the **sharpness**, the largest Hessian eigenvalue $\lambda_{\max}(\nabla^2 f)$, and set $\eta < 2/\lambda_{\max}$. In practice, training a neural network does not cooperate, because the Hessian evolves along the trajectory. Under full-batch gradient descent, the sharpness typically climbs until it hits roughly $2/\eta$ and then hovers right around that threshold; meanwhile, the loss decreases over the long run even

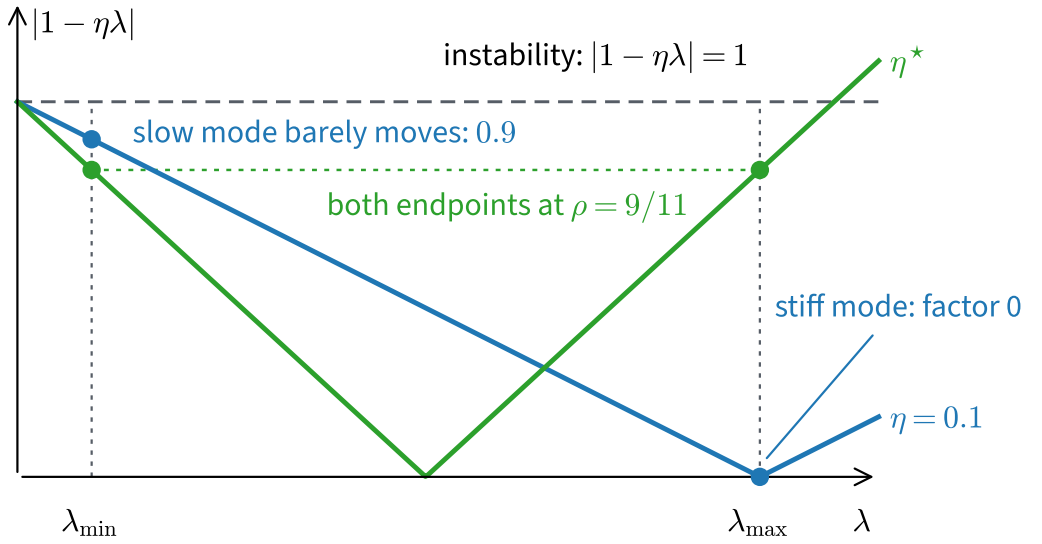


Figure 35.3: The per-mode contraction factors $|1-\eta\lambda|$ form a V with its vertex at $\lambda = 1/\eta$, and the convergence factor $\rho(\eta)$ is the larger of its values at the extreme eigenvalues $\lambda_{\min} = 1$ and $\lambda_{\max} = 10$. Choosing $\eta = 0.1$ eliminates the stiff mode immediately but leaves the slow mode contracting at 0.9; the optimal step $\eta^\star = 2/(\lambda_{\min}+\lambda_{\max})$ equalizes both endpoints at $(\kappa-1)/(\kappa+1) = 9/11$. Lowering either endpoint would raise the other.

though it jumps upward on many individual steps (Cohen et al. 2021). Section 9.1.4 runs this experiment on a small tanh network and examines it in detail. Here we ask a narrower question: how much of this behavior does our quadratic model explain?

Roughly half. In the Hessian's eigenbasis, the top mode's multiplier is $1-\eta\lambda_{\max}$, which drops below -1 precisely when $\lambda_{\max} > 2/\eta$. That coordinate then alternates in sign with growing amplitude, explaining how a loss that trends downward overall can spike on individual steps. On a fixed quadratic, however, that growth would run away to infinity. What pulls the sharpness back down instead lies outside the quadratic approximation: the third-order Taylor term, which the quadratic model ignores, turns the oscillation into a gradual drift toward flatter regions of lower sharpness (Damian et al. 2023). Exercise 8 shows that adding momentum shifts this stability threshold to $(2+2\beta)/\eta$.

35.1.2.5 From Quadratics to Convex Functions

What carries over beyond quadratics is the linear dependence on κ. A differentiable function is convex if it lies on or above each of its tangent planes, and μ-strongly convex if it clears them by a quadratic margin, $f(\mathbf{y}) \geq f(\mathbf{x}) + \nabla f(\mathbf{x})^\top(\mathbf{y}-\mathbf{x}) + \frac{\mu}{2}\|\mathbf{y}-\mathbf{x}\|^2$ for all $\mathbf{x}, \mathbf{y}$; Section 35.3 develops both notions. If f is L-smooth and μ-strongly convex, then at every point it is sandwiched between two quadratics of curvatures μ and L. The ratio $\kappa = L/\mu$ replaces $\lambda_{\max}/\lambda_{\min}$, and gradient descent with $\eta = 1/L$ satisfies

$$f(\mathbf{x}_k) - f^\star \leq \left(1 - \frac{1}{\kappa}\right)^k (f(\mathbf{x}_0) - f^\star), \tag{35.11}$$

which again requires $O(\kappa \log \frac{1}{\epsilon})$ iterations. If f is smooth and convex without being strongly convex, gradient descent still converges to the global optimum, though at the slower sublinear rate $O(1/k)$; drop convexity altogether, and only the stationarity bound Equation 35.6 remains. The proofs rely on convexity and are developed in Section 35.3.3.3, which also compares these general bounds against the sharper rate attained on quadratics.

35.1.3 Momentum and Acceleration

Gradient descent requires $O(\kappa)$ iterations because a single step size must simultaneously accommodate the stiffest and flattest modes. Momentum keeps the same single gradient evaluation per step, yet changes the update rule so that both modes can contract at the same rate.

35.1.3.1 Heavy-Ball Momentum

The geometry of the valley immediately suggests what to change. Along the steep walls, consecutive gradients point in opposite directions (the zigzagging in Figure 35.2), so averaging recent gradients cancels the oscillation. Along the flat floor, consecutive gradients are small but consistently aligned, so accumulating them builds speed. One simple mechanism does both: give the iterate a velocity with memory. **Heavy-ball momentum** (Polyak 1964) replaces the plain gradient step with

$$\mathbf{v}_{k+1} = \beta\,\mathbf{v}_k - \eta\,\nabla f(\mathbf{x}_k), \qquad \mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{v}_{k+1}, \tag{35.12}$$

where $\beta \in [0, 1)$ is the momentum parameter. The velocity $\mathbf{v}_k$ is an exponentially decaying sum of past gradients. Since $\mathbf{v}_k = \mathbf{x}_k - \mathbf{x}_{k-1}$, eliminating $\mathbf{v}$ gives the equivalent two-step form

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \eta\,\nabla f(\mathbf{x}_k) + \beta\,(\mathbf{x}_k - \mathbf{x}_{k-1}).$$

Each update takes a gradient step and adds β times the previous displacement; setting $\beta = 0$ recovers ordinary gradient descent.

On a quadratic, the modes once again decouple. In a single mode with gradient λu_k, the two-step update reads $u_{k+1} = (1 + \beta - \eta\lambda)\,u_k - \beta\,u_{k-1}$, so each iterate depends on its two predecessors. A second-order recurrence of this form is the discrete counterpart of a damped harmonic oscillator—a heavy ball rolling in a bowl with friction $1 - \beta$—where β controls the damping.

Figure 35.4 displays the three resulting regimes on $\mathrm{diag}(1, 10)$, all using gradient descent's optimal step $\eta = 2/11$. With too little momentum, the path zigzags much like plain gradient descent. With just the right amount, it settles smoothly after a single overshoot. With too much momentum, the ball carries excess inertia and spirals around the minimum. Next, we attach exact convergence rates to all three regimes—and discover that achieving the fastest possible rate also calls for a larger step size.

35.1.3.2 The $\sqrt{\kappa}$ Law

We now show that with the right pair (η, β), every mode contracts at the identical rate $(\sqrt{\kappa}-1)/(\sqrt{\kappa}+1)$. For $\kappa = 10$, this optimal pair is $\beta^\star \approx 0.27$ and $\eta_{\mathrm{HB}} \approx 0.23$, dropping

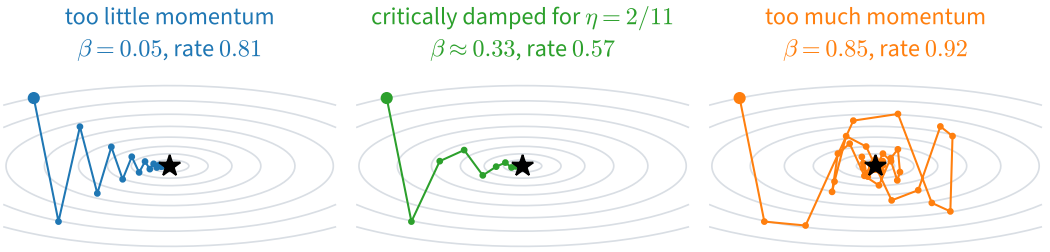


Figure 35.4: Heavy ball on the $\kappa = 10$ valley $\frac{1}{2}(x^2+10y^2)$, with all three runs using gradient descent's step $\eta = 2/11$. Too little momentum zigzags like gradient descent (the stiff mode's coefficient $1 + \beta - \eta\lambda_{\text{max}}$ is negative, giving a negative real root); the critically damped β for this step is fastest; too much momentum spirals around the minimum. Reaching the optimal $\sqrt{\kappa}$ rate of 0.52 requires the larger step η_{HB} of Equation 35.14, at which $\beta = 0.05$ would diverge.

the contraction factor from gradient descent's 0.82 to 0.52—reaching six digits in 27 steps instead of 69. Remarkably, the entire result follows from a single algebraic fact about one mode.

Proposition (heavy ball contracts every mode at $\sqrt{\beta}$). *Let $0 < \beta < 1$. On $f(u) = \frac{\lambda}{2}u^2$, the heavy-ball iterates obey $u_{k+1} = (1 + \beta - \eta\lambda)u_k - \beta u_{k-1}$. Whenever $(1 - \sqrt{\beta})^2 \le \eta\lambda \le (1 + \sqrt{\beta})^2$, both roots of the characteristic polynomial have modulus $\sqrt{\beta}$, and $|u_k| \le C\, k\, \beta^{k/2}$ for $k \ge 1$ and a constant C depending on u_0, u_1, and $\eta\lambda$. The linear factor k is needed only at the two endpoints, where the root is repeated.*

Proof. Substitute the trial solution $u_k = z^k$ into the recurrence and divide by z^{k-1} to obtain the characteristic equation

$$z^2 - (1 + \beta - \eta\lambda)\, z + \beta = 0. \tag{35.13}$$

The product of the two roots is β. They form a complex-conjugate pair whenever the discriminant $(1+\beta-\eta\lambda)^2-4\beta$ is negative, that is, when $|1+\beta-\eta\lambda| < 2\sqrt{\beta}$, which rearranges to $(1 - \sqrt{\beta})^2 < \eta\lambda < (1 + \sqrt{\beta})^2$. Because complex conjugates $z, \bar{z}$ satisfy $|z|^2 = z\bar{z} = \beta$, both roots have modulus $\sqrt{\beta}$ regardless of λ, and the general solution $u_k = az^k + b\bar{z}^k$ satisfies $|u_k| \le (|a| + |b|)\beta^{k/2}$. At the two endpoints, the discriminant vanishes and we get a double real root $(1 + \beta - \eta\lambda)/2 = \pm\sqrt{\beta}$. A repeated root z yields the general solution $u_k = (a + bk)z^k$, which accounts for the extra factor k. ∎

Corollary (the tuning). Place the two extreme modes right at the endpoints of the complex-root interval: $\eta\lambda_{\text{min}} = (1 - \sqrt{\beta})^2$ and $\eta\lambda_{\text{max}} = (1 + \sqrt{\beta})^2$. Every intermediate eigenvalue then falls inside the interval, so every single mode has root modulus $\sqrt{\beta}$. Taking square roots gives $\sqrt{\eta\lambda_{\text{max}}} - \sqrt{\eta\lambda_{\text{min}}} = 2\sqrt{\beta}$ and $\sqrt{\eta\lambda_{\text{max}}} + \sqrt{\eta\lambda_{\text{min}}} = 2$. Solving the second equation for η and dividing the first by the second gives $\sqrt{\beta} = (\sqrt{\kappa} - 1)/(\sqrt{\kappa} + 1)$. With the tuning

$$\eta_{\text{HB}} = \left(\frac{2}{\sqrt{\lambda_{\max}} + \sqrt{\lambda_{\min}}} \right)^2, \qquad \beta^\star = \left(\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1} \right)^2, \tag{35.14}$$

heavy ball contracts every mode at the rate

$$\rho_{\text{HB}} = \sqrt{\beta^\star} = \frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1} \;\approx\; 1 - \frac{2}{\sqrt{\kappa}}. \tag{35.15}$$

Compare this with Equation 35.10: momentum replaces κ with $\sqrt{\kappa}$, slashing the iteration complexity from $O(\kappa \log \frac{1}{\epsilon})$ to $O(\sqrt{\kappa} \log \frac{1}{\epsilon})$ (Polyak 1964). At finite k, the double roots at the two extreme modes add a modest logarithmic overhead: the error scales as $k\rho^k$, so with $\log(1/\rho) \approx 2/\sqrt{\kappa}$, reaching accuracy ϵ solves $k \approx \frac{\sqrt{\kappa}}{2}(\log \frac{1}{\epsilon} + \log k)$—the asymptotic law plus a correction of order $\sqrt{\kappa} \log \kappa$.

Figure 35.5 makes this proof visual for $\kappa = 10$. On the left, the modulus of the dominant root sits flat at $\sqrt{\beta}$ across the entire complex-root interval. On the right, in the complex plane, every mode's root pair lies squarely on the circle of radius $\sqrt{\beta^\star} = 0.52$, whereas gradient descent's real multipliers stretch out to ± 0.82. By allowing the roots to move off the real axis, momentum lets the stiff and flat modes share a single small modulus instead of forcing the flat mode to crawl at $1 - 2/\kappa$.

The proposition also explains the exact rates of the three trajectories in Figure 35.4, where $\eta = 2/11$ places the two modes at $\eta\lambda = 0.18$ and 1.82. At $\beta = 0.85$, both modes lie strictly inside the interval: the contraction rate is $\sqrt{0.85} = 0.92$, and the complex roots produce the spiraling overshoot. The smallest β whose interval still captures the slow mode, $\beta = (1 - \sqrt{2/11})^2 \approx 0.33$, is the fastest choice for this step size, achieving rate 0.57. At $\beta = 0.05$, the interval $[0.60, 1.50]$ misses both modes: the slow mode gets a real root 0.81, while the stiff mode gets a negative real root that flips sign on every step—an alternation that a continuous-time physical oscillator can never produce.

Be aware that this $\sqrt{\kappa}$ theorem holds strictly for quadratics. Lessard et al. (2016) constructed a one-dimensional, smooth, strongly convex function on which heavy ball with the tuning Equation 35.14 fails to converge from certain initializations, trapping the iterates in a stable limit cycle (exercise 6). Thus, the widespread use of heavy-ball momentum in deep learning (Sutskever et al. 2013) rests on the local quadratic picture and empirical success rather than a global convergence theorem.

35.1.3.3 Nesterov's Look-Ahead

One small modification recovers a global guarantee: evaluate the gradient not at the current position $\mathbf{x}_k$, but at the look-ahead point $\mathbf{x}_k + \beta \mathbf{v}_k$ where momentum is about to carry us. **Nesterov's accelerated gradient** (Nesterov 1983) updates via

$$\mathbf{v}_{k+1} = \beta \mathbf{v}_k - \eta \nabla f(\mathbf{x}_k + \beta \mathbf{v}_k), \qquad \mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{v}_{k+1}. \tag{35.16}$$

That shift in evaluation point is the only difference from heavy ball. Intuitively, if momentum is about to carry the iterate past the minimum, the gradient at the look-ahead point already points back uphill and applies the brakes before the overshoot occurs.

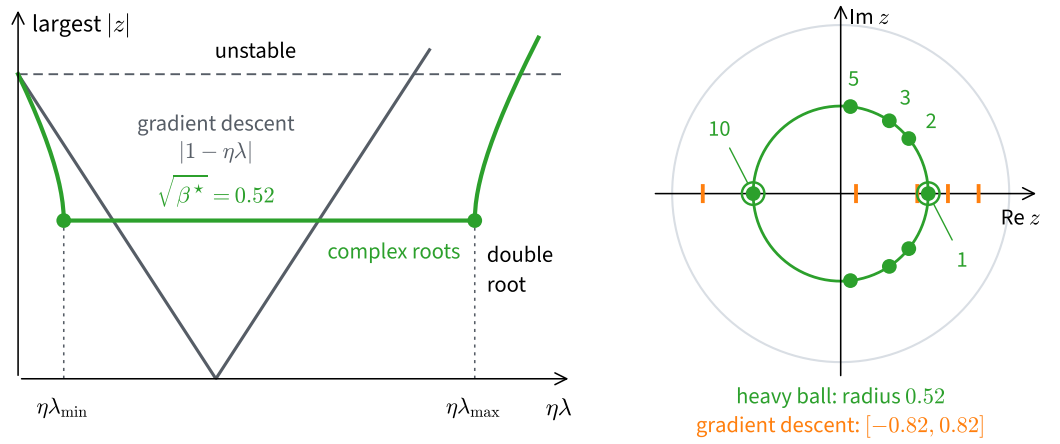


Figure 35.5: Heavy ball at the tuning of Equation 35.14 on the $\kappa = 10$ example. (a) For fixed β, the modulus of the larger root is flat at $\sqrt{\beta}$ wherever the roots are complex; the optimal tuning places the two extreme modes at the ends of that interval. (b) In the complex plane, every mode's root pair (labeled by its eigenvalue λ) lies on the circle of radius $\sqrt{\beta^\star} = 0.52$, while gradient descent's factors span $[-0.82, 0.82]$ on the real axis. The extreme modes are double roots, producing the factor k in the error.

Proposition (Nesterov on quadratics). *Let $\kappa = L/\mu$. With $\eta = 1/L$ and $\beta = (\sqrt{\kappa}-1)/(\sqrt{\kappa}+1)$, every mode of a quadratic with eigenvalues in $[\mu, L]$ has characteristic roots of modulus at most $1 - 1/\sqrt{\kappa}$, with a double root at the slow mode $\lambda = \mu$. Consequently, the error is bounded by $C\,k\,(1-1/\sqrt{\kappa})^k$.*

Proof. In a single mode with eigenvalue λ, write $a = 1 - \eta\lambda \in [0, 1 - 1/\kappa]$. The velocity update becomes $v_{k+1} = a\beta v_k - \eta\lambda u_k$, giving $u_{k+1} = a u_k + a\beta v_k$. Substituting $v_k = u_k - u_{k-1}$ yields the recurrence $u_{k+1} = (1+\beta)a\,u_k - \beta a\,u_{k-1}$, whose characteristic polynomial $z^2 - (1+\beta)a\,z + \beta a$ has discriminant $a\,[(1+\beta)^2 a - 4\beta]$. For our choice of β, $(1+\beta)^2 = 4\kappa/(\sqrt{\kappa}+1)^2$. At the slow mode $a = (\kappa-1)/\kappa$, the bracket evaluates to $4(\kappa-1)/(\sqrt{\kappa}+1)^2 - 4\beta = 4\beta - 4\beta = 0$, giving a double root at $(1+\beta)a/2 = (\sqrt{\kappa}-1)/\sqrt{\kappa} = 1 - 1/\sqrt{\kappa}$. Every stiffer mode has a smaller a, making the bracket strictly negative and producing complex roots (or roots at 0 when $a = 0$) of modulus $\sqrt{\beta a} < \sqrt{\beta(1-1/\kappa)} = 1 - 1/\sqrt{\kappa}$. ■

On quadratics, Nesterov's contraction factor is 0.684 for $\kappa = 10$, 0.900 for $\kappa = 100$, and 0.968 for $\kappa = 1000$—slightly slower than heavy ball's 0.519, 0.818, and 0.939, yet preserving the same $\sqrt{\kappa}$ scaling. Crucially, unlike heavy ball, Nesterov's method keeps its guarantees beyond quadratics, which we state here without proof (Nesterov 2018). On every μ-strongly convex, L-smooth function, Nesterov's update with the same constant β satisfies $f(\mathbf{x}_k) - f^\star \le C\,(1-1/\sqrt{\kappa})^k$. (Mind the units: this bounds the function gap $f - f^\star$, which scales as the squared distance, so its exponent is half the exponent in the quadratic proposition.) On smooth convex functions without strong convexity, a schedule $\beta_k \to 1$ achieves $f(\mathbf{x}_k) - f^\star = O(1/k^2)$, improving on gradient descent's $O(1/k)$. Moreover, this speed cannot be beaten in general: no method that builds its iterates solely from past gradients can surpass $O(1/k^2)$ or the $\sqrt{\kappa}$ dependence in the worst case, provided the dimension is large relative to the number of iterations (Nemirovski and Yudin 1983). Exercise 7 constructs the

worst-case function that proves this limit.

Let us now verify the iteration counts we quoted at the start of the section. On $\mathrm{diag}(1, \kappa)$ starting from $(1, 1)$, gaining six digits should cost gradient descent roughly $\frac{\kappa}{2} \log 10^6 \approx 6.9\,\kappa$ steps and heavy ball somewhat more than $6.9\sqrt{\kappa}$.

```
def iterations(lam, eta, beta=0.0, lookahead=False, tol=1e-6):
    """Steps until |x_k| <= tol |x_0| on f(x) = sum_i lam_i x_i^2 / 2."""
    x, v = np.array([1.0, 1.0]), np.zeros(2)
    n0 = np.linalg.norm(x)
    for k in range(1, 10**6):
        g = lam * (x + beta * v) if lookahead else lam * x
        v = beta * v - eta * g
        x = x + v
        if np.linalg.norm(x) <= tol * n0:
            return k

print('kappa      GD   heavy ball   Nesterov     GD/kappa   HB/sqrt(kappa)')
for kappa in [10.0, 100.0, 1000.0]:
    lam, sk = np.array([1.0, kappa]), np.sqrt(kappa)
    gd = iterations(lam, 2 / (1 + kappa))
    hb = iterations(lam, (2 / (1 + sk))**2, ((sk - 1) / (sk + 1))**2)
    na = iterations(lam, 1 / kappa, (sk - 1) / (sk + 1), lookahead=True)
    print(f'{kappa:5.0f} {gd:7d} {hb:9d} {na:10d}      '
          f'{gd / kappa:7.1f}     {hb / sk:7.1f}')
```

```
kappa      GD   heavy ball   Nesterov     GD/kappa   HB/sqrt(kappa)
   10      69         27         43          6.9          8.5
  100     691         93        155          6.9          9.3
 1000    6908        315        508          6.9         10.0
```

The table confirms every prediction. Gradient descent's step count divided by κ stays pinned at 6.9, reaching $6{,}908$ iterations at $\kappa = 1000$, while heavy ball finishes in just 315. Notice that heavy ball's ratio $k/\sqrt{\kappa}$ drifts gently upward (8.5, 9.3, 10.0) because each extreme mode has a double root: its 2×2 iteration matrix forms a Jordan block, a repeated eigenvalue with only a single eigenvector (Section 33.3.1.5), whose k-th power brings in the factor k. The refined prediction $k \approx \frac{\sqrt{\kappa}}{2}(\log 10^6 + \log k)$ matches closely (8.6, 9.2, 9.8), and at $\kappa = 10^4$, heavy ball takes $1{,}056$ steps, a ratio of 10.6 (exercise 5). Nesterov's 508 iterations at $\kappa = 1000$ sit cleanly between the two, just as its contraction factor 0.968 versus heavy ball's 0.939 predicts. Because all three methods evaluate only one gradient per step, momentum's speedup costs no extra gradient work. Yet $\sqrt{\kappa}$ is also the limit of what gradients alone can achieve: by the first-order lower bound, no method built solely from past gradients can beat it in high dimensions. Removing the dependence on κ altogether requires a step that actually models the curvature (Section 35.2).

35.1.4 Summary

Smoothness caps how far a gradient step can be trusted: on an L-smooth objective, descent is guaranteed up to $\eta = 2/L$. On a quadratic, the best fixed step contracts the error by $(\kappa - 1)/(\kappa + 1)$ per iteration, forcing the step count to grow linearly with the condition number κ because stability is governed by the stiffest mode while progress is governed by the flattest. Momentum equalizes every mode at the shared contraction rate $\sqrt{\beta}$, and its tuned factor

$(\sqrt{\kappa}-1)/(\sqrt{\kappa}+1)$ cuts the iteration count from $O(\kappa)$ to $O(\sqrt{\kappa})$ at no extra gradient cost. On neural networks, sharpness rises to the stability ceiling $2/\eta$ of the chosen step size, a phenomenon that the quadratic model explains only in part. Three questions remain open: what an optimizer can achieve when it models curvature directly (Section 35.2), when a stationary point is guaranteed to be a global minimum (Section 35.3), and what changes when gradients are noisy (Section 35.4).

35.1.5 Exercises

1. **Preconditioned descent directions.** The proposition of Section 35.1.1.1 measures the length of a direction with the Euclidean norm. Choosing a different norm yields a different steepest-descent direction.
   1. Show that $\mathbf{d} = -\mathbf{P}^{-1}\nabla f(\mathbf{x})$ is a descent direction for every positive definite matrix $\mathbf{P}$.
   2. Measure length by $\|\mathbf{d}\|_{\mathbf{P}} = \sqrt{\mathbf{d}^\top \mathbf{P}\,\mathbf{d}}$. Show that the minimizer of $\nabla f(\mathbf{x})^\top \mathbf{d}$ over $\|\mathbf{d}\|_{\mathbf{P}} = 1$ is proportional to $-\mathbf{P}^{-1}\nabla f(\mathbf{x})$, and identify the $\mathbf{P}$ that recovers Newton's direction $-\big(\nabla^2 f(\mathbf{x})\big)^{-1}\nabla f(\mathbf{x})$, the subject of Section 35.2. The change of variables $\mathbf{u} = \mathbf{P}^{1/2}\mathbf{d}$ reduces the problem to the Euclidean case.
2. code **Smoothness constants.** The smoothness constant L of Equation 35.3 bounds the curvature of f, and for a logistic loss it can be computed directly from the data.
   1. Let f be twice continuously differentiable and L-smooth. Show that $\|\nabla^2 f(\mathbf{x})\,\mathbf{v}\| \le L\,\|\mathbf{v}\|$ for all $\mathbf{x}$ and $\mathbf{v}$, and conclude that every Hessian eigenvalue lies in $[-L, L]$. This establishes the converse direction of the equivalence stated after Equation 35.3.
   2. Consider

$$f(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n} \log\left(1 + \exp(-y_i\,\mathbf{x}_i^\top \mathbf{w})\right) + \frac{\nu}{2}\|\mathbf{w}\|^2$$

   with labels $y_i = \pm 1$ and $\nu = 0.1$—the `loss` function that we will use in the data cell of Section 35.2.1, where $\mathbf{x}_i^\top$ is the i-th row of $\mathbf{X} \in \mathbb{R}^{n\times d}$ ($n = 200$, $d = 5$, with independent standard Gaussian entries). Show that $L = \lambda_{\max}(\mathbf{X}^\top\mathbf{X})/(4n) + \nu$ is a valid smoothness constant and that the Hessian attains this bound at $\mathbf{w} = \mathbf{0}$.
   3. Compute this L for a 200×5 matrix $\mathbf{X}$ of standard Gaussian entries and labels $y_i \in \{\pm 1\}$ drawn with $P(y_i = 1) = 1/(1 + \exp(-\mathbf{x}_i^\top \mathbf{w}_{\text{true}}))$ for $\mathbf{w}_{\text{true}} = (1, -2, 1.5, 0, 2)$. It comes out near 0.4 for every draw; explain why using $\lambda_{\max}(\mathbf{X}^\top\mathbf{X})/n \approx (1 + \sqrt{d/n})^2$ for standard Gaussian entries. Count how many steps $\eta = 0.5$ (the reference step) and $\eta = 1/L$ each need from $\mathbf{w} = \mathbf{0}$ to reach $\|\nabla f(\mathbf{w})\| \le 10^{-10}$. Which is faster, and does $\eta = 0.5$ stay below $2/L$?
3. code **Clipping on the quartic.** Take $f(x) = \frac{1}{4}x^4$ and the clipped update $x \leftarrow x - \eta\,\min(1, c/|g|)\,g$ with $g = x^3$.
   1. Show that $3x^2 \le 4 + |x|^3$ for all x, with equality at $|x| = 2$, so that f satisfies Equation 35.8 with $L_0 = 4$ and $L_1 = 1$. Show also that no L_0 works with $L_1 = 0$.

2. Run the clipped update from $x_0 = 3$ with $\eta = 0.3$ and $c = 1$, the step at which the unclipped iteration reached 6.5×10^{103} in six steps. How many steps are clipped, and how far does each of them move x?
3. Show that the clipped update decreases $|x|$ at every step, from every starting point $x_0 \neq 0$, whenever $\eta\, c^{2/3} < 2$. Treat the clipped case $|x| > c^{1/3}$ and the unclipped case separately. Why can no fixed step without clipping have this property?

4. **Steepest descent in other norms.** ● Problem 1b measures length by a norm $\|\cdot\|_{\mathbf{A}}$ induced by an inner product. Norms that do not come from an inner product produce steepest directions of a very different form.

   1. Find the minimizers of $\nabla f(\mathbf{x})^\top \mathbf{d}$ over $\|\mathbf{d}\|_\infty = 1$ and over $\|\mathbf{d}\|_1 = 1$, together with their minimal slopes, and state when each minimizer is unique.
   2. Let the parameter be a matrix whose gradient $\mathbf{G}$ has reduced SVD $\mathbf{G} = \mathbf{U}\,\mathrm{diag}(\sigma_i)\,\mathbf{V}^\top$, let the directional slope of $\mathbf{D}$ be $\mathrm{tr}(\mathbf{G}^\top \mathbf{D})$, and measure length by the spectral norm $\|\mathbf{D}\|_2$. Show that the minimal slope over $\|\mathbf{D}\|_2 \le 1$ is $-\sum_i \sigma_i$, attained at $\mathbf{D} = -\mathbf{U}\mathbf{V}^\top$—the polar factor that Muon applies to its momentum matrix (Section 9.6). Writing $\mathrm{tr}(\mathbf{G}^\top \mathbf{D}) = \sum_i \sigma_i\, \mathbf{u}_i^\top \mathbf{D}\, \mathbf{v}_i$ is a helpful first step. This is the steepest-descent perspective of Bernstein and Newhouse (2024) .

5. code **Heavy ball: the double root and the factor k.** ● Apply heavy ball Equation 35.12 to $f(u) = \frac{\lambda}{2}u^2$.

   1. Write one iteration as a linear map on the state (u_k, u_{k-1}) and show that the characteristic polynomial of its 2×2 matrix is Equation 35.13.
   2. At the tuning Equation 35.14, the two extreme modes have double eigenvalues $\sqrt{\beta^\star}$ and $-\sqrt{\beta^\star}$. Show that the matrix then has only a single eigenvector, and that the k-th power of the Jordan block $\left(\begin{smallmatrix} \rho & 1 \\ 0 & \rho \end{smallmatrix}\right)$ has entry $k\rho^{k-1}$ above the diagonal.
   3. With $\|\mathbf{e}_k\| \approx k\rho^k$ and $\log(1/\rho) \approx 2/\sqrt{\kappa}$, solve $k = \frac{\sqrt{\kappa}}{2}\big(\log\frac{1}{\epsilon} + \log k\big)$ numerically for $\epsilon = 10^{-6}$ and $\kappa \in \{10, 100, 1000\}$, and compare $k/\sqrt{\kappa}$ with the momentum cell's column 8.5, 9.3, 10.0.
   4. Run the momentum cell's heavy ball at $\kappa = 10^4$ and compare its step count with your prediction. Explain why the ratio $k/\sqrt{\kappa}$ never settles to a constant as κ grows.

6. code **Heavy ball beyond quadratics.** ● Lessard et al. (2016) define a one-dimensional function through its gradient:

$$\nabla f(x) = \begin{cases} 25x, & x < 1, \\ x + 24, & 1 \le x < 2, \\ 25x - 24, & x \ge 2. \end{cases}$$

   1. Show that f is μ-strongly convex with $\mu = 1$ and L-smooth with $L = 25$.
   2. Run heavy ball with the tuning Equation 35.14 for 300 steps from $x_0 = 3.3$ with zero initial velocity, and run Nesterov's method Equation 35.16 with $\eta = 1/L$ and $\beta = (\sqrt{\kappa} - 1)/(\sqrt{\kappa} + 1)$ from the same start, where $\kappa = L/\mu$. Plot x_k against k for both methods.
   3. The heavy-ball iterates approach a period-three cycle that visits the piece $x < 1$ twice and the piece $x \ge 2$ once. Express the heavy-ball update across these

pieces as a linear system for the three cycle points, solve it, and compare your solution with the simulation.

4. Repeat the heavy-ball run from a grid of starting points on $[-6, 6]$ and report the fraction that land on the cycle. Every piece of f is a quadratic with curvature in $[\mu, L]$; why does the per-mode analysis behind Equation 35.15 fail to apply to f?

7. code **A lower bound for first-order methods.** ● This problem derives the $O(1/k^2)$ part of the bound of Nemirovski and Yudin (1983) quoted in Section 35.1.3 (the smooth convex case), following the construction of Nesterov (2018) . Here, a first-order method is any algorithm whose iterates satisfy $\mathbf{x}_k \in \mathbf{x}_0 + \operatorname{span}\{\nabla f(\mathbf{x}_0), \dots, \nabla f(\mathbf{x}_{k-1})\}$; take $\mathbf{x}_0 = \mathbf{0}$. For $p \le d$, let $\mathbf{A}_p \in \mathbb{R}^{d\times d}$ be zero except for its leading $p \times p$ block, which is tridiagonal with 2 on the diagonal and -1 on both off-diagonals, and define

$$f_p(\mathbf{x}) = \frac{L}{4}\left(\tfrac{1}{2}\,\mathbf{x}^\top \mathbf{A}_p\,\mathbf{x} - x_1\right).$$

1. Show that f_p is convex and L-smooth.

2. Show that if $\mathbf{x}$ is supported on the first j coordinates, then $\nabla f_p(\mathbf{x})$ is supported on the first $j+1$. Conclude that the k-th iterate of every first-order method is supported on the first k coordinates.

3. Show that the minimizer of f_p has entries $x_i^\star = 1 - i/(p+1)$ for $i \le p$ and zero beyond, that the minimum value is $-\frac{L}{8}(1 - \frac{1}{p+1})$, and that $\|\mathbf{x}^\star\|^2 \le (p+1)/3$.

4. Let $2k+1 \le d$. The functions f_{2k+1} and f_k agree on all vectors supported on the first k coordinates. Use this fact to show that every first-order method applied to f_{2k+1} satisfies, with $\mathbf{x}^\star$ the minimizer of f_{2k+1},

$$f_{2k+1}(\mathbf{x}_k) - \min f_{2k+1} \;\ge\; \frac{3L\,\|\mathbf{x}_0 - \mathbf{x}^\star\|^2}{32\,(k+1)^2}.$$

Because the construction requires $d \ge 2k+1$, the bound governs roughly the first $d/2$ iterations. In fixed dimension, long runs can eventually do better asymptotically, but for neural networks with d in the millions this restriction never bites.

5. For $k \in \{5, 10, 25, 50, 100, 200, 400\}$, run k steps of gradient descent with $\eta = 1/L$ and of Nesterov's update Equation 35.16 with $\eta = 1/L$ and $\beta_j = (j-1)/(j+2)$ at step $j = 1, 2, \dots$ on f_{2k+1} with $d = 2k+1$, and divide each suboptimality gap by the lower bound. Which ratio stays bounded as k grows?

8. code **Edge of stability with momentum.** ●● Cohen et al. (2021) report that when momentum is added, the sharpness settles at a threshold different from $2/\eta$.

1. Use the iteration matrix from problem 5a to show that heavy ball Equation 35.12 on $f(x) = \frac{\lambda}{2}x^2$ with $0 \le \beta < 1$ converges if and only if $0 < \eta\lambda < 2 + 2\beta$. Derive the corresponding stability interval for Nesterov's update Equation 35.16.
2. Modify the edge-of-stability experiment of Section 9.1.4 to train with heavy ball and with Nesterov's update for $\beta \in \{0.25, 0.5\}$, choosing for each run a step size whose predicted threshold lies between 5 and 12. Track the sharpness against both $2/\eta$ and your threshold from (a).

3. At which threshold does the sharpness settle? What does this imply for choosing η from a sharpness measured at initialization when training with momentum?

35.2 Curvature Methods: Newton, Quasi-Newton, and Trust Regions

Gradient descent never measures the curvature of the function it minimizes; it only endures it. The steepest direction caps the step size, the flattest direction then crawls, and the number of iterations grows with the condition number κ, or with $\sqrt{\kappa}$ once momentum is added (Section 35.1.3). On our running example $\frac{1}{2}(x_1^2 + 1000\,x_2^2)$, that came to $6{,}908$ steps of gradient descent, or 315 with momentum. Yet anyone can solve this problem by inspection: divide each coordinate of the gradient by its curvature (1 or 1000) and a single step lands directly on the minimum. What can an optimizer achieve when it accounts for curvature instead of merely surviving it?

The classical answer is Newton's method, which predates gradient descent by a century and a half. At each iterate, it fits a quadratic approximation to the objective and jumps straight to the minimizer of that quadratic. In this section we examine how fast that step converges, what it costs, where it breaks down, and how much of its curvature information can be recovered from gradients alone when second derivatives are unavailable. Near a minimum, Newton's method is remarkably fast; for a neural network, however, forming the Hessian is far too expensive, and on nonconvex functions the method is drawn to saddle points just as readily as to minima. We then turn to two remedies: quasi-Newton methods, which rebuild curvature from successive gradient differences, and trust-region methods, which restrict each step to a neighborhood where the quadratic model actually agrees with the function.

For training deep networks, these classical methods are used much less often than first-order optimizers, and it is worth being candid about why. A full Hessian is too large to form, and minibatch noise spoils curvature estimates built from differences of gradients. Even so, the underlying ideas reappear throughout deep learning: the per-coordinate scaling of Adam acts as a diagonal stand-in for Newton's inverse Hessian (Section 35.5), and the damping added to curvature matrices plays the same stabilizing role as a trust region. The central result to keep in mind is the quadratic convergence of Newton's method Equation 35.19; the subsection on quasi-Newton methods may be skipped on a first reading. From Section 35.1, we carry over two ingredients: the Armijo condition for backtracking line search and the test function $\frac{1}{2}x^2 + \cos y$, whose saddle point stalled gradient descent.

35.2.1 Newton's Method

Gradient descent builds a linear approximation to f at the current point. Because a linear function has no minimum, the algorithm cannot choose its own step length and must rely on an external learning rate. A quadratic approximation, by contrast, has a unique minimum whenever it curves upward, giving us a step whose length is set automatically by the local geometry.

35.2.1.1 The Newton Step

Newton's method lets the local curvature determine both the direction and the length of the step (Section 9.1.5.1). Writing $\mathbf{g}_k = \nabla f(\mathbf{x}_k)$, we approximate f near $\mathbf{x}_k$ by its second-order Taylor expansion in the displacement $\mathbf{p}$:

$$m_k(\mathbf{p}) = f(\mathbf{x}_k) + \mathbf{g}_k^\top \mathbf{p} + \tfrac{1}{2}\,\mathbf{p}^\top \nabla^2 f(\mathbf{x}_k)\,\mathbf{p}. \tag{35.17}$$

When the Hessian $\nabla^2 f(\mathbf{x}_k)$ is positive definite, the model m_k is a strictly convex bowl with a unique minimizer, found by setting $\nabla m_k(\mathbf{p}) = \mathbf{g}_k + \nabla^2 f(\mathbf{x}_k)\,\mathbf{p} = \mathbf{0}$. Solving for $\mathbf{p}$ and moving there gives **Newton's method**:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \left(\nabla^2 f(\mathbf{x}_k)\right)^{-1} \nabla f(\mathbf{x}_k). \tag{35.18}$$

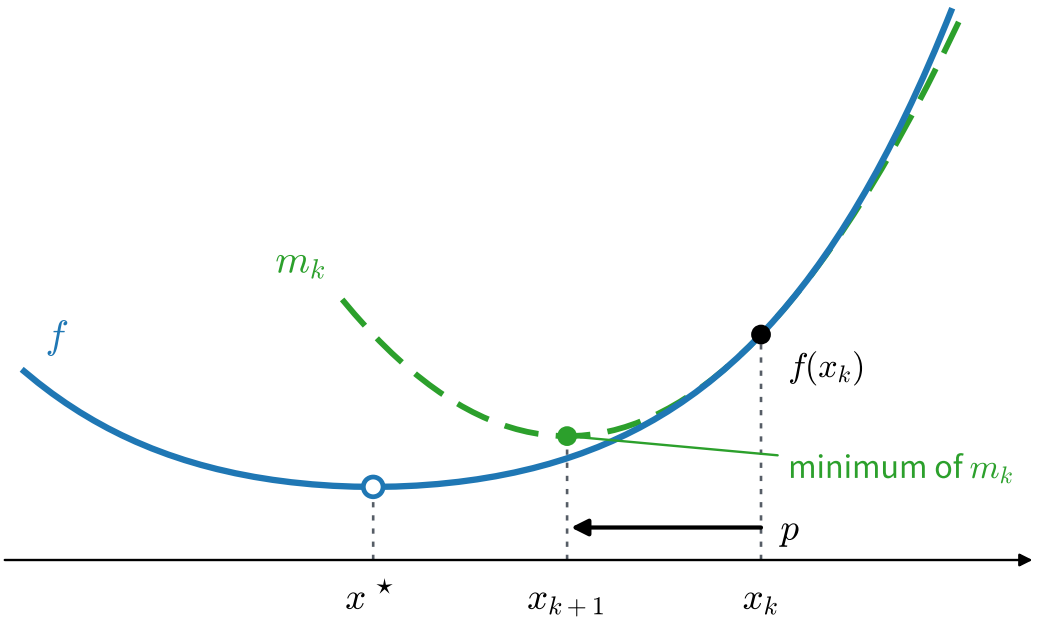


Figure 35.6: One Newton step on $f(x) = \frac{1}{4}x^4 + \frac{1}{2}x^2$ from $x_k = 1$. The model m_k (dashed) shares the value, slope and curvature of f at x_k, and the step p moves to the minimizer of the model, $x_{k+1} = \frac{1}{2}$. The minimizer of f is $x^\star = 0$.

Figure 35.6 shows one such step in one dimension. Notice that no step size appears in Equation 35.18: the inverse Hessian rescales each direction according to its curvature. More broadly, Newton's direction belongs to the family of preconditioned steps $\mathbf{d} = -\mathbf{P}^{-1}\nabla f$ with a positive definite matrix $\mathbf{P}$ (here $\mathbf{P} = \nabla^2 f$). Every such direction is a descent direction, since $\nabla f^\top \mathbf{d} = -\nabla f^\top \mathbf{P}^{-1} \nabla f < 0$. In fact, Newton's direction is simply the direction of steepest descent when step lengths are measured in the norm induced by the Hessian rather than the Euclidean norm (exercise 1).

On a positive definite quadratic $\frac{1}{2}\mathbf{x}^\top \mathbf{A}\mathbf{x}$, the Taylor model is exact, so a single Newton step lands immediately on the minimizer: $\mathbf{x}_1 = \mathbf{x}_0 - \mathbf{A}^{-1}\mathbf{A}\mathbf{x}_0 = \mathbf{0}$, regardless of the condition number κ. The reason goes deeper than matrix cancellation. Newton's method is **affine-invariant**, meaning that its trajectory does not depend on the choice of coordinates. Under any invertible linear change of variables $\mathbf{x} = \mathbf{T}\mathbf{y}$, running Newton's method on $f(\mathbf{T}\mathbf{y})$ produces the exact images of Newton's iterates for f. Gradient descent, by contrast, changes its

path under every non-orthogonal $\mathbf{T}$, while a scalar multiple of an orthogonal matrix merely rescales the effective step size (exercise 1). Because some linear transformation turns any positive definite quadratic into a perfect sphere, the condition number cannot affect Newton's method. A long valley is, to Newton's method, a round bowl drawn on stretched paper.

35.2.1.2 Quadratic Convergence

For a general non-quadratic objective, the Taylor model is only a local approximation, so one step is no longer enough. What survives is remarkable speed. Once the iterate enters a neighborhood of a minimizer, Newton's method converges much faster than geometrically, and the proof requires little more than the fundamental theorem of calculus. Let $\mathbf{e}_k = \mathbf{x}_k - \mathbf{x}^\star$ denote the error at iteration k.

Proposition (quadratic convergence). *Let $\nabla f(\mathbf{x}^\star) = \mathbf{0}$, and suppose that on a ball around $\mathbf{x}^\star$ the Hessian satisfies $\nabla^2 f(\mathbf{x}) \succeq \mu \mathbf{I}$ and $\|\nabla^2 f(\mathbf{x}) - \nabla^2 f(\mathbf{y})\| \le L_2 \|\mathbf{x} - \mathbf{y}\|$. If $\mathbf{x}_k$ lies in the ball, then*

$$\|\mathbf{e}_{k+1}\| \;\le\; \frac{L_2}{2\mu}\,\|\mathbf{e}_k\|^2. \tag{35.19}$$

Proof. Abbreviate $\nabla^2 f_k = \nabla^2 f(\mathbf{x}_k)$. Because $\nabla f(\mathbf{x}^\star) = \mathbf{0}$, subtracting $\mathbf{x}^\star$ from the Newton update yields $\mathbf{e}_{k+1} = \mathbf{e}_k - (\nabla^2 f_k)^{-1}\mathbf{g}_k = (\nabla^2 f_k)^{-1}\big[\nabla^2 f_k\,\mathbf{e}_k - (\nabla f(\mathbf{x}_k) - \nabla f(\mathbf{x}^\star))\big]$. Applying the fundamental theorem of calculus along the line segment from $\mathbf{x}^\star$ to $\mathbf{x}_k$ expresses the gradient difference as $\int_0^1 \nabla^2 f(\mathbf{x}^\star + t\mathbf{e}_k)\,\mathbf{e}_k\,dt$, so the bracketed term becomes $\int_0^1 \big[\nabla^2 f_k - \nabla^2 f(\mathbf{x}^\star + t\mathbf{e}_k)\big]\mathbf{e}_k\,dt$. Since $\mathbf{x}_k$ and $\mathbf{x}^\star + t\mathbf{e}_k$ are separated by $(1-t)\mathbf{e}_k$, the Lipschitz bound on the Hessian gives this integral norm at most $\int_0^1 L_2(1-t)\|\mathbf{e}_k\|^2\,dt = \tfrac{L_2}{2}\|\mathbf{e}_k\|^2$. Combining this with $\|(\nabla^2 f_k)^{-1}\| \le 1/\mu$ completes the proof. $\blacksquare$

To read what Equation 35.19 implies, multiply both sides by $L_2/(2\mu)$: $\tfrac{L_2}{2\mu}\|\mathbf{e}_{k+1}\| \le \big(\tfrac{L_2}{2\mu}\|\mathbf{e}_k\|\big)^2$. The scaled error squares at every step. If it starts below 1, it shrinks monotonically and every subsequent iterate remains in the ball. Once it drops below 0.1, it plunges to 10^{-2}, 10^{-4}, 10^{-8}, roughly doubling the number of accurate digits at each iteration. Compare that with gradient descent on our running example, which needs about $1{,}150$ steps to gain a single digit. The catch is written directly into the hypotheses: this quadratic rate is strictly local and guarantees nothing until the iterate is already close to $\mathbf{x}^\star$.

Far from $\mathbf{x}^\star$, the pure Newton step can overshoot even when the Hessian is positive definite. The standard safeguard is **damping**: update via $\mathbf{x}_k + \alpha_k \mathbf{p}_k$ along the Newton direction $\mathbf{p}_k = -(\nabla^2 f(\mathbf{x}_k))^{-1}\mathbf{g}_k$, choosing $\alpha_k \le 1$ by backtracking until the Armijo condition $f(\mathbf{x}_k + \alpha \mathbf{p}_k) \le f(\mathbf{x}_k) + c\,\alpha\,\mathbf{g}_k^\top \mathbf{p}_k$ holds (the general-direction form of the Armijo rule in Section 35.1). Near a minimizer with a positive definite Hessian, the full step $\alpha = 1$ automatically passes this test whenever $c < \tfrac{1}{2}$, so damping adds no overhead once quadratic convergence kicks in.

To test these predictions in code, we use ℓ_2-regularized logistic regression on $n = 200$ synthetic examples in $d = 5$ dimensions,

$$f(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n} \log\left(1 + \exp(-y_i\, \mathbf{x}_i^\top \mathbf{w})\right) + \frac{\nu}{2}\|\mathbf{w}\|^2, \qquad \nu = 0.1,$$

with binary labels $y_i = \pm 1$ sampled from a logistic model (we write ν for the weight-decay coefficient to reserve λ for eigenvalues). The setup cell below also computes a reference optimum $\mathbf{w}^\star$ using $4{,}000$ gradient steps of size $\eta = 0.5$, which is safely below $2/L$ because $L = \lambda_{\max}(\mathbf{X}^\top\mathbf{X})/(4n) + \nu$ is roughly 0.4 here (exercise 2).

```
rng = np.random.default_rng()
n, d, nu = 200, 5, 0.1
w_true = np.array([1.0, -2.0, 1.5, 0.0, 2.0])
X = rng.normal(size=(n, d))
y = np.where(rng.random(n) < 1 / (1 + np.exp(-X @ w_true)), 1.0, -1.0)

def loss(w, X, y, nu):
    return np.logaddexp(0, -y * (X @ w)).mean() + nu / 2 * (w @ w)

def grad_rows(w, X, y, nu):                    # row i: gradient of example i
    s = 1 / (1 + np.exp(y * (X @ w)))          # sigmoid(-y_i x_i^T w)
    return -(y * s)[:, None] * X + nu * w

w_star = np.zeros(d)
for _ in range(4000):                          # full-batch reference optimum
    w_star = w_star - 0.5 * grad_rows(w_star, X, y, nu).mean(axis=0)
```

We can now test both predictions directly. On our running quadratic $\mathrm{diag}(1, 1000)$, a single Newton step should replace the $\log 10^6 / \log(1001/999) \approx 6{,}908$ steps required by gradient descent at its optimal fixed step size (Section 35.1.3). On the non-quadratic logistic loss, the exponent of the gradient norm should roughly double at every iteration once the local phase begins.

```
A2 = np.diag([1.0, 1000.0])                    # the running example, kappa = 1000
x = np.array([1.0, 1.0])
x_new = x - np.linalg.solve(A2, A2 @ x)        # one Newton step
rate = (1000 - 1) / (1000 + 1)
print(f'Newton on the quadratic: |x_1| = {np.linalg.norm(x_new):.1e} after one step')
print(f'GD at the optimal step contracts by {rate:.4f}; it needs '
      f'{int(np.ceil(np.log(1e-6) / np.log(rate)))} steps for |x_k| <= 1e-6 |x_0|')
w = np.zeros(d)                                # Newton on the logistic loss
for it in range(6):
    s = 1 / (1 + np.exp(y * (X @ w)))
    g = grad_rows(w, X, y, nu).mean(axis=0)
    hess = (X * (s * (1 - s))[:, None]).T @ X / n + nu * np.eye(d)
    w = w - np.linalg.solve(hess, g)
    print(f'Newton iteration {it + 1}:  |grad| = {np.linalg.norm(g):.1e}')
print(f'max |w - w_star| = {np.abs(w - w_star).max():.1e}')
```

```
Newton on the quadratic: |x_1| = 0.0e+00 after one step
GD at the optimal step contracts by 0.9980; it needs 6908 steps for |x_k| <= 1e-6 |x_0|
Newton iteration 1:  |grad| = 3.5e-01
Newton iteration 2:  |grad| = 4.1e-02
```

```
Newton iteration 3:  |grad| = 1.7e-03
Newton iteration 4:  |grad| = 3.0e-06
Newton iteration 5:  |grad| = 9.9e-12
Newton iteration 6:  |grad| = 1.6e-17
max |w - w_star| = 4.4e-16
```

Both predictions hold. On the quadratic, a single Newton step drives the error to zero, whereas optimal fixed-step gradient descent contracts by only 0.998 per step and requires $6{,}908$ iterations. On the logistic loss, the gradient norm starts at a few tenths and drops below 10^{-8} by the fifth iteration, its exponent roughly doubling once it becomes small. After just six iterations, Newton's iterate agrees with the reference optimum to roughly 10^{-15}, matching the precision of the reference itself. Six linear solves have accomplished what took $4{,}000$ gradient steps.

35.2.1.3 The Cost of a Step

Unfortunately, comparing iteration counts alone flatters Newton's method. The Hessian contains d^2 entries, and solving the linear system in Equation 35.18 generally requires $O(d^3)$ operations per step. For $d = 5$ this cost is negligible, but for a neural network with even a modest 10^8 parameters, the Hessian has 10^{16} entries, ruling out exact Newton's method before the first step is ever taken (Section 9.1.5.1). Practical alternatives either approximate the Hessian from gradients, as in the quasi-Newton methods below, or use cheap diagonal and structured preconditioners (Section 35.5).

35.2.1.4 Saddle Points

Computational cost is not the only drawback of Newton's method, nor is it the most instructive one. Because the Newton step solves $\nabla m_k = \mathbf{0}$, it jumps to the stationary point of the quadratic model whether that point is a minimum, a maximum, or a saddle. Return to the function $f(x, y) = \frac{1}{2}x^2 + \cos y$ from Section 35.1, which has a saddle at the origin and a minimum of -1 at $(0, \pi)$, and start at $(0.5, 0.3)$. Here the gradient is $(0.5, -\sin 0.3) = (0.5, -0.296)$, and the Hessian $\operatorname{diag}(1, -\cos 0.3) = \operatorname{diag}(1, -0.955)$ is indefinite: the surface curves upward along x and downward along y. Which way does a Newton step move y from this point, compared with a plain gradient step of size $\eta = 1$?

```
def f_s(v):
    return 0.5 * v[0]**2 + np.cos(v[1])

def grad_s(v):
    return np.array([v[0], -np.sin(v[1])])

def hess_s(v):
    return np.diag([1.0, -np.cos(v[1])])

x_n = x_g = np.array([0.5, 0.3])
ys_n, ys_g = [], []
for _ in range(6):
    x_n = x_n - np.linalg.solve(hess_s(x_n), grad_s(x_n))    # Newton
    x_g = x_g - 1.0 * grad_s(x_g)                            # GD, eta = 1
    ys_n.append(x_n[1])
    ys_g.append(x_g[1])
```

```
print('Newton y:', np.array(ys_n[:3]), f'  f = {f_s(x_n):.4f}')
print('GD     y:', np.array(ys_g).round(4), f'  f = {f_s(x_g):.4f}')
```

```
Newton y: [-9.33624961e-03  2.71275924e-07 -6.61744490e-21]   f = 1.0000
GD     y: [0.5955 1.1565 2.0718 2.9489 3.1404 3.1416]   f = -1.0000
```

Newton's first step sends x to 0 and y to $0.3 - \tan 0.3 = -0.0093$; the second sends y to 2.7×10^{-7}. We are seeing Newton's rapid local convergence (here even cubic, since $y - \tan y \approx -y^3/3$), driving the iterate straight into the saddle $(0, 0)$ with $f = 1$. Gradient descent moves y in the opposite direction, through 0.60, 1.16, 2.07, 2.95, 3.140, and 3.1416, reaching the minimum $(0, \pi)$ with $f = -1$. The richer model has led to the worse point. Because the curvature along y is negative, the stationary point of the quadratic model is a saddle, and Newton's method is blind to the difference between a saddle and a minimum. A trust region (Section 35.2.3) fixes this flaw.

35.2.2 Quasi-Newton Methods

Newton's method requires explicit second derivatives. Can we instead infer curvature from the gradients we are already computing? When the gradient changes rapidly along a step, the objective must have large curvature in that direction. An optimizer that records these changes can gradually assemble a model of the Hessian without ever evaluating a second derivative.

35.2.2.1 The Secant Method

In one dimension the idea is immediate: the slope of the chord connecting the last two gradients estimates f'',

$$h_k = \frac{f'(x_k) - f'(x_{k-1})}{x_k - x_{k-1}}, \qquad x_{k+1} = x_k - \frac{f'(x_k)}{h_k}.$$

This is the **secant method**, which is simply Newton's step with $f''(x_k)$ replaced by a finite difference of gradients. Consider $f(x) = e^x - 2x$, whose minimizer is $\log 2 \approx 0.6931$, starting from $x_0 = 0$ and $x_1 = 1$. The first secant slope is $h_1 = (e-2) - (1-2) = e - 1 \approx 1.718$, giving $x_2 = 1 - (e-2)/1.718 \approx 0.582$. From there the errors shrink as 1.1×10^{-1}, 1.6×10^{-2}, 9.3×10^{-4}, 7.7×10^{-6}, and 3.6×10^{-9}. Each error is roughly half the product of the previous two (the factor of one-half comes from $f'''/(2f'')$ at the minimizer), so the number of correct digits grows by a factor approaching 1.6 per step. That is slightly slower than Newton's factor of 2, yet each iteration requires only a single gradient evaluation and no second derivative. On a one-dimensional quadratic $\frac{\lambda}{2}x^2$, the secant slope equals λ exactly, and the step from x_1 lands directly on the minimizer.

In d dimensions, the analogous secant condition is a single vector equation, which is far from enough to pin down a $d \times d$ matrix: one step reveals curvature along one direction only. We need a principled rule to fill in the remaining directions.

35.2.2.2 BFGS

A **quasi-Newton** method maintains an approximation $\mathbf{M}_k$ to the inverse Hessian and updates it after each step using the observed change in the gradient. After moving from $\mathbf{x}_k$ to

$\mathbf{x}_{k+1}$, we record the secant pair

$$\mathbf{s}_k = \mathbf{x}_{k+1} - \mathbf{x}_k, \qquad \mathbf{y}_k = \nabla f(\mathbf{x}_{k+1}) - \nabla f(\mathbf{x}_k).$$

On a quadratic $\frac{1}{2}\mathbf{x}^\top \mathbf{A}\mathbf{x}$, the gradient difference satisfies $\mathbf{y}_k = \mathbf{A}\mathbf{s}_k$, so the true inverse Hessian maps $\mathbf{y}_k$ back to $\mathbf{s}_k$. We require the updated approximation $\mathbf{M}_{k+1}$ to obey the same **secant equation**:

$$\mathbf{M}_{k+1}\,\mathbf{y}_k = \mathbf{s}_k. \tag{35.20}$$

Among all symmetric matrices satisfying Equation 35.20, **BFGS** (named after Broyden, Fletcher, Goldfarb, and Shanno) chooses the one closest to the previous estimate $\mathbf{M}_k$ in a suitably weighted Frobenius norm:

$$\mathbf{M}_{k+1} = (\mathbf{I} - \gamma_k \mathbf{s}_k \mathbf{y}_k^\top)\,\mathbf{M}_k\,(\mathbf{I} - \gamma_k \mathbf{y}_k \mathbf{s}_k^\top) + \gamma_k \mathbf{s}_k \mathbf{s}_k^\top, \qquad \gamma_k = \frac{1}{\mathbf{y}_k^\top \mathbf{s}_k}. \tag{35.21}$$

Even without deriving Equation 35.21, we can read off how it works (dropping the index k on the pair for clarity). The right factor $\mathbf{I} - \gamma \mathbf{y}\mathbf{s}^\top$ sends $\mathbf{y}$ to zero, because $\gamma\,\mathbf{s}^\top\mathbf{y} = 1$, while leaving every vector orthogonal to $\mathbf{s}$ untouched. Thus the first term erases what $\mathbf{M}_k$ previously predicted along $\mathbf{y}$, and the second term writes in the new observation: $\mathbf{M}_{k+1}\mathbf{y} = \gamma\,\mathbf{s}\,(\mathbf{s}^\top\mathbf{y}) = \mathbf{s}$, satisfying the secant equation.

Just as importantly, the update preserves positive definiteness, ensuring that the next search direction $-\mathbf{M}_{k+1}\nabla f(\mathbf{x}_{k+1})$ remains a descent direction. For any nonzero vector $\mathbf{z}$, set $\mathbf{w} = (\mathbf{I} - \gamma \mathbf{y}\mathbf{s}^\top)\mathbf{z}$, so that $\mathbf{z}^\top \mathbf{M}_{k+1}\mathbf{z} = \mathbf{w}^\top \mathbf{M}_k \mathbf{w} + \gamma(\mathbf{s}^\top\mathbf{z})^2$. When $\mathbf{M}_k \succ 0$ and $\gamma > 0$, both terms are nonnegative and cannot vanish simultaneously: $\mathbf{w} = \mathbf{0}$ forces $\mathbf{z} = \gamma(\mathbf{s}^\top\mathbf{z})\,\mathbf{y}$, which requires $\mathbf{s}^\top\mathbf{z} \neq 0$. The requirement $\gamma > 0$ is the **curvature condition** $\mathbf{y}_k^\top \mathbf{s}_k > 0$, meaning that the directional slope increased along the step. It holds automatically on any strictly convex objective; on general objectives, the line search enforces it via a second check known as the Wolfe condition (Nocedal and Wright 2006).

How quickly do secant pairs reconstruct the true curvature? Let us run BFGS from $(1, 1)$ on the two-dimensional quadratic $\mathrm{diag}(1, 10)$, starting with $\mathbf{M}_0 = \mathbf{I}$ and using the exact line search $\alpha = -\mathbf{g}^\top\mathbf{p}/(\mathbf{p}^\top\mathbf{A}\mathbf{p})$, and inspect both $\mathbf{x}$ and $\mathbf{M}$ after each step.

```
def bfgs_update(M, s, y):
    gamma = 1.0 / (y @ s)
    V = np.eye(len(s)) - gamma * np.outer(s, y)
    return V @ M @ V.T + gamma * np.outer(s, s)

A = np.diag([1.0, 10.0])
x, M = np.array([1.0, 1.0]), np.eye(2)
for k in range(2):
    g = A @ x
    p = -M @ g
    alpha = -(g @ p) / (p @ A @ p)           # exact line search
    s = alpha * p
    M = bfgs_update(M, s, A @ s)             # y = A s on a quadratic
    x = x + s
```

```
        print(f'step {k + 1}: x = {x.round(4)},  M = {M.round(4).tolist()}')
    print(f'A^-1 = {np.linalg.inv(A).tolist()},  '
          f'max |M_2 - A^-1| = {np.abs(M - np.linalg.inv(A)).max():.0e}')
```

```
step 1: x = [ 0.8991 -0.009 ],  M = [[1.009, -0.0001], [-0.0001, 0.1]]
step 2: x = [0. 0.],  M = [[1.0, -0.0], [-0.0, 0.1]]
A^-1 = [[1.0, 0.0], [0.0, 0.1]],  max |M_2 - A^-1| = 1e-16
```

After a single step, $\mathbf{M}_1 \approx \operatorname{diag}(1.009, 0.100)$ is already close to $\mathbf{A}^{-1}$ along the steep direction it has just probed. After the second step, $\mathbf{M}_2 = \mathbf{A}^{-1}$ up to floating-point rounding and $\mathbf{x}_2 = \mathbf{0}$. This two-dimensional behavior generalizes cleanly: on any d-dimensional positive definite quadratic with exact line searches, each secant equation remains satisfied after subsequent updates, and the steps are mutually $\mathbf{A}$-conjugate ($\mathbf{s}_i^\top \mathbf{A}\mathbf{s}_j = 0$ for $i \neq j$). After d steps, $\mathbf{M}_d\mathbf{A} = \mathbf{I}$ and BFGS reaches the exact minimizer (exercise 2).

Finite termination, however, holds specifically for quadratics with exact line searches and should not be taken as a general guarantee. On non-quadratic objectives, line searches are inexact, the true Hessian varies from step to step, and stochastic gradients corrupt $\mathbf{y}_k$: the difference between two minibatch gradients is often dominated by sampling noise rather than curvature. In addition, full BFGS stores a dense $d \times d$ matrix. **L-BFGS** avoids that memory cost by retaining only the m most recent secant pairs and applying the resulting inverse-Hessian approximation (a scaled identity plus a correction of rank at most $2m$) in $O(md)$ time and memory (Liu and Nocedal 1989). It works very well on deterministic, medium-scale problems, though it remains awkward when fresh minibatch noise makes gradient differences unreliable.

35.2.3 Trust Regions

Newton's method went astray at $(0.5, 0.3)$ because it trusted its local quadratic model across the entire space. That model was built from derivatives at a single point and accurately reflects f only nearby. A trust-region method takes this limitation literally: it keeps the full curvature information, including directions of negative curvature, and instead bounds the step to a region where the model can be trusted.

35.2.3.1 The Subproblem

At $(0.5, 0.3)$, the quadratic model of $\frac{1}{2}x^2 + \cos y$ has curvature -0.955 along y. Because it curves downward without bound, asking for its unconstrained minimum makes no sense. A **trust-region method** instead minimizes the quadratic model inside a ball of radius Δ_k, which represents how far we believe the approximation holds:

$$\min_{\|\mathbf{p}\| \le \Delta_k} m_k(\mathbf{p}) = f(\mathbf{x}_k) + \nabla f(\mathbf{x}_k)^\top \mathbf{p} + \tfrac{1}{2}\mathbf{p}^\top \mathbf{B}_k \mathbf{p}. \tag{35.22}$$

Here $\mathbf{B}_k$ is either the exact Hessian or a symmetric approximation to it. Restricting $\mathbf{p}$ to a closed ball guarantees that a minimizer exists even when $\mathbf{B}_k$ has negative eigenvalues, because any continuous function achieves a minimum on a compact set. Better still, the exact solutions of this constrained problem have a clean characterization.

Proposition (solution of the subproblem). *A step $\mathbf{p}$ with $\|\mathbf{p}\| \le \Delta$ solves Equation 35.22 if and only if there is a $\tau \ge 0$ with*

$$(\mathbf{B} + \tau \mathbf{I})\,\mathbf{p} = -\mathbf{g}, \qquad \mathbf{B} + \tau \mathbf{I} \succeq \mathbf{0}, \qquad \tau\,(\Delta - \|\mathbf{p}\|) = 0.$$

This result is Theorem 4.1 of Nocedal and Wright (2006); exercise 3 proves the sufficiency of these conditions. Geometrically, the trust region adds a uniform shift τ to every eigenvalue of $\mathbf{B}$, just enough to make the shifted quadratic convex. Two cases arise. When $\mathbf{B}$ is positive definite and the unconstrained Newton step already lies inside the ball, we have $\tau = 0$ and simply take the Newton step. Otherwise the solution lies on the boundary $\|\mathbf{p}\| = \Delta$, and τ is the root of the scalar equation $\|(\mathbf{B} + \tau \mathbf{I})^{-1}\mathbf{g}\| = \Delta$ over $\tau > -\lambda_{\min}(\mathbf{B})$, where the left-hand side decreases strictly as τ grows.

This scalar equation fails to have a root only in the degenerate *hard case*, when $\mathbf{g}$ is orthogonal to the eigenvectors associated with the smallest eigenvalue of $\mathbf{B}$ (explored in part d of exercise 3). As we will see in Section 35.6, the shift τ is simply the Lagrange multiplier enforcing the constraint $\tfrac{1}{2}\|\mathbf{p}\|^2 \le \tfrac{1}{2}\Delta^2$.

Between iterations, the radius Δ_k adapts according to how accurately the model predicted the actual drop in f. After solving Equation 35.22, we form the ratio of actual decrease to predicted decrease:

$$r_k = \frac{f(\mathbf{x}_k) - f(\mathbf{x}_k + \mathbf{p}_k)}{m_k(\mathbf{0}) - m_k(\mathbf{p}_k)}. \tag{35.23}$$

A ratio near 1 indicates that the quadratic model matched the objective well. In the implementation below, we accept the step whenever $r_k > 0.1$. Independently, we shrink the radius to $\Delta \leftarrow \Delta/4$ whenever $r_k < \tfrac{1}{4}$, and we double it when $r_k > \tfrac{3}{4}$ and the step hit the boundary $\|\mathbf{p}_k\| = \Delta_k$.

35.2.3.2 The Trust-Region Step at a Saddle

Let us work out one trust-region step at $(0.5, 0.3)$ by hand with $\Delta = 0.5$. At this point $\mathbf{g} = (0.5, -0.296)$ and $\mathbf{B} = \mathrm{diag}(1, -0.955)$. Making $\mathbf{B} + \tau\mathbf{I}$ positive semidefinite requires $\tau \ge 0.955$, so $\tau = 0$ is impossible and the step must lie on the boundary of the ball. The optimality system gives $(1+\tau)p_1 = -0.5$ and $(\tau - 0.955)p_2 = 0.296$ subject to $p_1^2 + p_2^2 = 0.25$, whose solution is $\tau \approx 1.60$ and $\mathbf{p} \approx (-0.19, 0.46)$. Moving to $(0.31, 0.76)$ lowers f from 1.080 to 0.771. Because the model predicted a decrease of 0.316 and the actual decrease is 0.309, the ratio is $r = 0.98$, so the step is accepted and the radius doubles. Crucially, the trust-region step moves *away* from the saddle along the negative-curvature direction where the model drops, whereas the unconstrained Newton step $(-0.5, -0.31)$ marched straight toward it.

To see where this subproblem solution leads over several iterations, we first implement `tr_step(g, B, Delta)`, which returns the unconstrained Newton step when $\mathbf{B} \succ \mathbf{0}$ and $\|\mathbf{B}^{-1}\mathbf{g}\| \le \Delta$, and otherwise bisects on τ until $\|(\mathbf{B} + \tau\mathbf{I})^{-1}\mathbf{g}\| = \Delta$:

```
def tr_step(g, B, Delta):
    """Minimize g p + p B p / 2 over |p| <= Delta; return (p, tau)."""
    eig_min = np.linalg.eigvalsh(B).min()
    if eig_min > 0 and np.linalg.norm(np.linalg.solve(B, -g)) <= Delta:
        return np.linalg.solve(B, -g), 0.0     # Newton step fits: tau = 0
    lo = hi = max(0.0, -eig_min)
```

```
    hi += 1e3
    for _ in range(100):                        # bisection for |p(tau)| = Delta
        tau = (lo + hi) / 2
        p = np.linalg.solve(B + tau * np.eye(len(g)), -g)
        lo, hi = (tau, hi) if np.linalg.norm(p) > Delta else (lo, tau)
    return p, tau
```

Starting from $\mathbf{x} = (0.5, 0.3)$ with $\Delta = 0.5$, we now run six trust-region iterations on f_s, printing the iterate, objective value, step norm $\|\mathbf{p}\|$, shift τ, agreement ratio r, and radius Δ at each step:

```
x, Delta = np.array([0.5, 0.3]), 0.5
print(' k        x            f      |p|    tau      r  Delta')
for k in range(6):
    g, B = grad_s(x), hess_s(x)
    p, tau = tr_step(g, B, Delta)
    r = (f_s(x) - f_s(x + p)) / -(g @ p + 0.5 * p @ B @ p)
    print(f'{k:2d} ({x[0]:6.3f},{x[1]:6.3f}) {f_s(x):7.4f} '
          f'{np.linalg.norm(p):6.3f} {tau:6.3f} {r:6.2f} {Delta:5.2f}')
    if r > 0.1:                                 # accept the step
        x = x + p
    if r < 0.25:
        Delta /= 4                              # poor prediction: shrink
    elif r > 0.75 and np.linalg.norm(p) > 0.99 * Delta:
        Delta *= 2                              # good, and on the boundary
print(f'final x = ({x[0]:.4f}, {x[1]:.4f}),  f = {f_s(x):.4f}')
```

```
 k        x            f      |p|    tau      r  Delta
 0 ( 0.500, 0.300)  1.0803  0.500  1.596   0.98  0.50
 1 ( 0.307, 0.761)  0.7711  1.000  1.419   0.87  1.00
 2 ( 0.180, 1.753) -0.1652  2.000  0.311   0.41  2.00
 3 ( 0.043, 3.749) -0.8205  0.696  0.000   0.88  2.00
 4 ( 0.000, 3.054) -0.9962  0.088  0.000   1.00  2.00
 5 ( 0.000, 3.142) -1.0000  0.000  0.000   1.00  2.00
final x = (0.0000, 3.1416),  f = -1.0000
```

The first row matches our hand calculation. The radius doubles from 0.5 to 1 and then to 2, while the third step (with ratio 0.41) is accepted without changing the radius. By the fourth iterate, $(0.04, 3.75)$, the algorithm has entered the basin where the Hessian is positive definite. From that point on, the unconstrained Newton step fits inside the trust-region ball with $\tau = 0$, so the method seamlessly turns into Newton's method and converges quadratically to the minimum $(0, \pi)$ with $f = -1$. Six iterations are enough; Figure 35.7 contrasts both trajectories.

Solving the trust-region subproblem exactly requires factoring $\mathbf{B}_k + \tau \mathbf{I}$ for each trial value of τ. Large-scale implementations instead solve it approximately using Hessian–vector products, each costing roughly the same as a single gradient evaluation (Section 34.5). Even with that speedup, neither BFGS nor trust-region methods replace stochastic gradient descent on large, noisy deep learning objectives.

The saddle of $\frac{1}{2}x^2 + \cos y$ has now tripped up two optimizers: gradient descent stalled at it from a start on the line $y = 0$, and Newton's method ran straight into it from a generic one. A trust region escapes, but only by checking each local model against the true objective. Yet

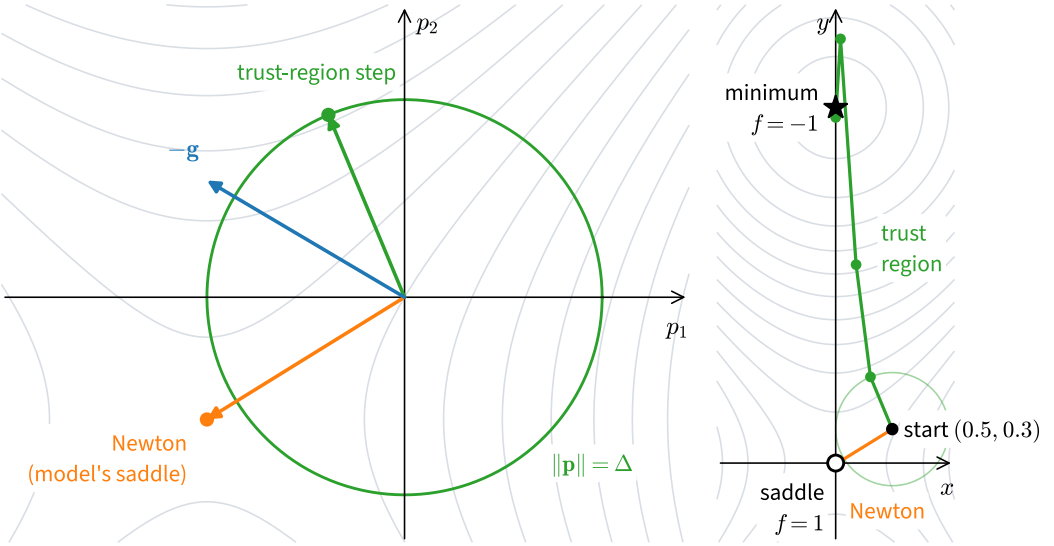


Figure 35.7: Newton's method and a trust-region method from $(0.5, 0.3)$ on $\frac{1}{2}x^2 + \cos y$, where the Hessian is $\mathrm{diag}(1, -0.96)$. (a) The local quadratic model forms a saddle surface: Newton's step moves to its stationary point, whereas minimizing the model over the ball of radius $\Delta = 0.5$ follows the negative-curvature direction away from the saddle. (b) Newton's method converges quadratically to the saddle $(0, 0)$ with $f = 1$, while the trust-region method (shown with its initial radius) reaches the minimum $(0, \pi)$ with $f = -1$ in six steps as its radius expands through $0.5, 1, 2$.

there is a large class of functions where none of these precautions are needed: functions with no saddles, no spurious basins, and where every stationary point is a global minimum. Identifying those functions is the subject of Section 35.3.

35.2.4 Summary

Newton's method minimizes a local quadratic model of f at each iterate. It solves any positive definite quadratic in a single step regardless of the condition number, and near a minimizer with a positive definite, Lipschitz-continuous Hessian it squares the error at every iteration. That speed comes with two obstacles: each step requires $O(d^2)$ memory and $O(d^3)$ operations, far beyond what a neural network can afford, and the unconstrained step is attracted to saddle points just as strongly as to minima. BFGS avoids explicit second derivatives by reconstructing the inverse Hessian from past step and gradient differences, recovering it completely in d steps on a d-dimensional quadratic. Trust-region methods make any quadratic model safe to use by confining each step to a ball, comparing the predicted and actual decreases in f, and expanding or shrinking the ball accordingly.

35.2.5 Exercises

1. **Reparametrization and preconditioning.** Let $\mathbf{T}$ be an invertible matrix and $h(\mathbf{y}) = f(\mathbf{T}\mathbf{y})$, so that $\mathbf{x} = \mathbf{T}\mathbf{y}$ is a change of coordinates.
   1. Express ∇h and $\nabla^2 h$ through the derivatives of f. Show that Newton's iterates Equation 35.18 for h, started at $\mathbf{y}_0 = \mathbf{T}^{-1}\mathbf{x}_0$, satisfy $\mathbf{y}_k = \mathbf{T}^{-1}\mathbf{x}_k$, where $\mathbf{x}_k$ are Newton's iterates for f.
   2. Run gradient descent with step η on h and map every iterate back by $\mathbf{x}_k = \mathbf{T}\mathbf{y}_k$. Show that the mapped iterates follow $\mathbf{x}_{k+1} = \mathbf{x}_k - \eta\,\mathbf{P}^{-1}\nabla f(\mathbf{x}_k)$ for a positive definite matrix $\mathbf{P}$, and find $\mathbf{P}$. By problem 1b this direction is the steepest one in the norm $\|\cdot\|_{\mathbf{P}}$. What is $\|\mathbf{x}\|_{\mathbf{P}}$ in the coordinates $\mathbf{y}$?
   3. For $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{A}\,\mathbf{x}$ with $\mathbf{A} = \operatorname{diag}(1, 1000)$, the $\kappa = 1000$ quadratic of the Newton subsection, find a diagonal $\mathbf{T}$ for which gradient descent on h with $\eta = 1$ reaches the minimizer in one step. What does constructing such a $\mathbf{T}$ require for a general positive definite $\mathbf{A}$?
2. code **BFGS on a quadratic.** ● Let $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{A}\mathbf{x} - \mathbf{b}^\top \mathbf{x}$ with $\mathbf{A}$ positive definite, and run BFGS Equation 35.21 with exact line searches from $\mathbf{M}_0 = \mathbf{I}$.
   1. Show that an exact line search along $\mathbf{p}_k$ gives $\nabla f(\mathbf{x}_{k+1})^\top \mathbf{p}_k = 0$, and that $\mathbf{y}_k = \mathbf{A}\mathbf{s}_k$.
   2. Show the hereditary property: if $\mathbf{M}_k \mathbf{y}_j = \mathbf{s}_j$ for all $j < k$ and the steps $\mathbf{s}_0, \dots, \mathbf{s}_{k-1}$, $\mathbf{s}_k$ are $\mathbf{A}$-conjugate, then $\mathbf{M}_{k+1}\mathbf{y}_j = \mathbf{s}_j$ for all $j \le k$. (Hint: $\mathbf{s}_j^\top \mathbf{y}_k = \mathbf{s}_j^\top \mathbf{A}\mathbf{s}_k = 0$ for $j < k$.) Conclude that after d linearly independent steps $\mathbf{M}_d\mathbf{A} = \mathbf{I}$.
   3. Run the BFGS loop of the text in $d = 6$ dimensions, on a randomly rotated quadratic with eigenvalues from 1 to 1000. Print $\max_{j \le k} \|\mathbf{M}_{k+1}\mathbf{y}_j - \mathbf{s}_j\|$ after each step, and confirm that it stays at rounding level and that $\mathbf{M}_6 = \mathbf{A}^{-1}$. How far from the minimizer is gradient descent at its best fixed step after the same six gradients?
   4. The symmetric rank-one update $\mathbf{M}_{k+1} = \mathbf{M}_k + \frac{(\mathbf{s} - \mathbf{M}_k\mathbf{y})(\mathbf{s} - \mathbf{M}_k\mathbf{y})^\top}{(\mathbf{s} - \mathbf{M}_k\mathbf{y})^\top \mathbf{y}}$ also satisfies the secant equation. Show by a 2×2 example that it need not stay positive definite even when $\mathbf{y}^\top \mathbf{s} > 0$, and say which of the two checks in the text fails for it.
3. code **Trust region at a saddle.** ● Consider the model $m(\mathbf{p}) = \mathbf{g}^\top \mathbf{p} + \frac{1}{2}\mathbf{p}^\top \mathbf{B}\mathbf{p}$ with

$\mathbf{B} = \operatorname{diag}(1, -1)$ and $\mathbf{g} = (\frac{1}{2}, -\frac{3}{10})$.

1. Show that m is unbounded below on $\mathbb{R}^2$ and that the Newton point $-\mathbf{B}^{-1}\mathbf{g} = (-\frac{1}{2}, -\frac{3}{10})$ is a saddle of m, not a minimum.
2. Prove that the conditions of the subproblem proposition are sufficient: if $(\mathbf{B} + \tau\mathbf{I})\mathbf{p} = -\mathbf{g}$ with $\tau \geq 0$, $\mathbf{B} + \tau\mathbf{I} \succeq \mathbf{0}$, and $\tau(\Delta - \|\mathbf{p}\|) = 0$, then every $\|\mathbf{q}\| \leq \Delta$ satisfies $m(\mathbf{q}) \geq m(\mathbf{p}) + \frac{\tau}{2}(\|\mathbf{p}\|^2 - \|\mathbf{q}\|^2) \geq m(\mathbf{p})$. (Hint: expand $m(\mathbf{q}) - m(\mathbf{p})$ around $\mathbf{p}$ and use $\mathbf{g} = -(\mathbf{B} + \tau\mathbf{I})\mathbf{p}$.)
3. For $\Delta = \frac{1}{2}$ find $\tau \geq 1$ by bisection on $\|(\mathbf{B} + \tau\mathbf{I})^{-1}\mathbf{g}\| = \Delta$ (the answer is $\tau \approx 1.65$, $\mathbf{p} \approx (-0.19, 0.46)$), and check that the step increases the coordinate with negative curvature while the Newton step decreases it.
4. Replace $\mathbf{g}$ by $(\frac{1}{2}, 0)$. Show that $\|(\mathbf{B}+\tau\mathbf{I})^{-1}\mathbf{g}\| = \frac{1}{2}/(1+\tau) \leq \frac{1}{4}$ for all $\tau > 1$, so the boundary equation has no solution with $\tau > 1$. Find the minimizers on the ball directly (they have $\tau = 1$ and $p_2 = \pm\sqrt{\Delta^2 - p_1^2}$), and explain in one sentence what the ball does with a direction of negative curvature that the gradient does not see.

35.3 Convexity

When a training run finishes, we usually know only one thing about the point where it stopped: the loss no longer decreases there. Gradient descent with a safe step size reliably reaches such a point, where the gradient is nearly zero (Section 35.1), and Newton's method gets there faster, even when the stationary point turns out to be a saddle (Section 35.2). Neither guarantee tells us whether a much better point is hiding somewhere else. In this section we ask when a stationary point is actually the best point available, and how quickly gradient descent converges to it.

A simple one-dimensional experiment shows that this is not an idle question. Suppose we run the exact same gradient-descent loop from 500 random starting points, first on the bowl $f(x) = (x - \frac{1}{2})^2$ and then on the tilted double well $\phi(x) = (x^2 - 1)^2 + x/2$. On the bowl, every single run lands at the same minimizer. On the double well, nearly half of the runs get trapped in a shallower valley whose loss sits almost a full unit above the global minimum. Those stranded runs still have zero gradient—which is all that a stationarity guarantee promises—and looking at the gradient alone gives no hint that a deeper valley exists nearby.

The difference between the two functions comes down to a single geometric property. Every chord drawn between two points on the bowl lies above its graph, whereas the double well has chords that cut below the curve. Functions whose chords always lie on or above their graph are called *convex*, and for them our opening question has the cleanest possible answer: every local minimum is a global minimum. The proof takes just three lines (Section 35.3.2.2). Underneath it lies an even more useful fact: any tangent line to a convex function stays below the graph across the entire domain, so local information at a single point constrains the function everywhere. Summing this tangent bound along the iterates of gradient descent yields convergence rates that hold from any starting point. Averaging it over a random variable yields Jensen's inequality, a workhorse of the probability and information-theory chapters.

We cover a fair amount of ground here, and not every result carries the same weight. Three ideas are worth keeping at your fingertips: the tangent lower bound, the fact that local minima are global, and the linear convergence rate of gradient descent—whose proof relies on

a single inequality that survives even when convexity fails. We begin with convex sets and three equivalent tests for convex functions, derive their main consequences, and assemble a short calculus for recognizing convex losses. Since deep-network losses are not convex, we close by asking which parts of the theory carry over to them. If you are in a hurry, you can read the chord and tangent conditions in Section 35.3.2 and jump straight to Section 35.3.3. Two examples will accompany us throughout: the chapter's running quadratic $\frac{1}{2}(x_1^2 + 1000\, x_2^2)$ for measuring convergence rates, and the double well ϕ, which fails every test of convexity we throw at it.

35.3.1 Convex Sets

Because convex functions are defined along line segments between points of their domain, we start with the sets those points live in. The domain of a loss, the feasible region of a constrained problem, and the set of all minimizers are all sets. Fortunately, we need only a short toolkit: one definition, three standard families of examples, and one rule for building new convex sets from old ones.

35.3.1.1 Definition and Examples

Geometrically, a set is convex if the straight line segment connecting any two of its points stays entirely inside the set. Formally, a set $C \subseteq \mathbb{R}^d$ of d-dimensional points is **convex** if

$$\theta \mathbf{x} + (1-\theta)\,\mathbf{y} \in C \qquad \text{for all } \mathbf{x}, \mathbf{y} \in C \text{ and all } \theta \in [0,1]. \tag{35.24}$$

As θ moves from 1 to 0, the combination $\theta\mathbf{x}+(1-\theta)\mathbf{y}$ sweeps out the line segment from $\mathbf{x}$ to $\mathbf{y}$. Figure 35.8 contrasts a convex set with a nonconvex crescent and illustrates the two convex sets that appear most often in deep learning: the probability simplex and a half-space.

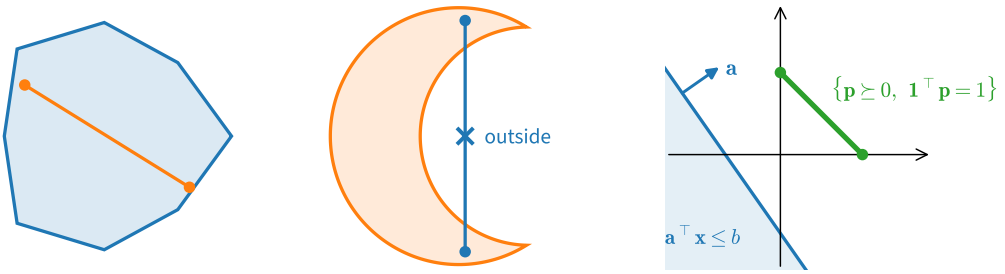


Figure 35.8: Convex versus nonconvex sets. Left: a convex set, where the segment between any two points stays inside. Middle: a nonconvex crescent, where a chord between two of its points passes outside the set. Right: the probability simplex $\{\mathbf{p} \succeq 0,\ \mathbf{1}^\top \mathbf{p} = 1\}$ and a half-space $\{\mathbf{x} : \mathbf{a}^\top \mathbf{x} \le b\}$, both convex.

To see what the definition rules out, consider the annulus $\{\mathbf{x} : 1 \le \|\mathbf{x}\| \le 2\}$. It contains both $(\frac{3}{2}, 0)$ and $(-\frac{3}{2}, 0)$, yet their midpoint is the origin, which lies in the hole. A single missing midpoint is enough to violate Equation 35.24.

Three families of convex sets cover almost everything we meet in this chapter. First, any hyperplane $\{\mathbf{x} : \mathbf{a}^\top \mathbf{x} = b\}$ and any half-space $\{\mathbf{x} : \mathbf{a}^\top \mathbf{x} \le b\}$ are convex because $\mathbf{a}^\top \mathbf{x}$

is linear: a weighted average of two numbers equal to b (or at most b) is again equal to b (or at most b). Second, any norm ball $\{\mathbf{x} : \|\mathbf{x}\| \le r\}$ is convex by the triangle inequality and homogeneity, $\|\theta\mathbf{x} + (1-\theta)\mathbf{y}\| \le \theta\|\mathbf{x}\| + (1-\theta)\|\mathbf{y}\| \le r$. This single check covers both the ℓ_2 balls of weight decay and gradient clipping and the ℓ_1 balls of sparsity-inducing regularization. Third is the probability simplex $\Delta = \{\mathbf{p} \in \mathbb{R}^d : \mathbf{p} \succeq 0,\ \mathbf{1}^\top \mathbf{p} = 1\}$, where $\mathbf{p} \succeq 0$ means every entry is nonnegative. Attention weights, class probabilities, and mixture weights all live on Δ, and its convexity follows immediately from our next rule.

35.3.1.2 New Convex Sets from Old

The intersection of any collection of convex sets, finite or infinite, is always convex. If two points lie in the intersection, then they lie in every set of the collection, so the segment between them stays inside every set and therefore inside the intersection. Intuitively, a convex set has no dents, and carving a dent-free set with another dent-free set cannot create one. Unions, by contrast, have no such guarantee: $[0,1] \cup [2,3]$ is a union of two convex intervals, yet the segment from 1 to 2 leaves the set.

This single intersection rule certifies two central sets at once. The probability simplex is the intersection of one hyperplane ($\mathbf{1}^\top \mathbf{p} = 1$) with d half-spaces ($-p_i \le 0$), so it is convex. Likewise, the set of symmetric positive semidefinite (PSD) matrices is convex because it is an intersection of infinitely many half-spaces: for each fixed vector $\mathbf{z}$, the constraint $\mathbf{z}^\top \mathbf{A}\mathbf{z} \ge 0$ is a single linear inequality in the entries of $\mathbf{A}$.

We can also build a convex set from the inside out. The **convex hull** of a set S is the collection of all weighted averages $\sum_i \theta_i \mathbf{x}_i$ of points $\mathbf{x}_i \in S$ with $\theta_i \ge 0$ and $\sum_i \theta_i = 1$. It is convex, and in fact it is the smallest convex set containing S (Exercise 2). Seen this way, the probability simplex is simply the convex hull of the standard basis vectors $\mathbf{e}_1, \dots, \mathbf{e}_d$: every probability distribution is a weighted average of certainties.

35.3.2 Convex Functions

Just as a convex set has no dents, a convex function is one whose graph curves upward without ever dipping into a second valley. We now develop three equivalent tests for convexity—requiring zero, one, and two derivatives respectively—followed by two refinements: strong convexity, which governs how fast gradient descent converges, and subgradients, which handle losses with corners. If you remember only one of the three tests, make it the second: the tangent lower bound is the engine behind almost every proof in this section.

35.3.2.1 The Chord Condition

The first test needs no derivatives at all, which means it applies just as easily to the absolute value and the hinge loss as to smooth objectives. A function $f : C \to \mathbb{R}$ on a convex domain C is **convex** if its graph lies on or below every chord joining two of its points:

$$f\left(\theta\mathbf{x} + (1-\theta)\,\mathbf{y}\right) \;\le\; \theta f(\mathbf{x}) + (1-\theta) f(\mathbf{y}) \qquad \text{for all } \mathbf{x}, \mathbf{y} \in C,\ \theta \in [0,1]. \tag{35.25}$$

Here the left side evaluates f at a point on the segment between $\mathbf{x}$ and $\mathbf{y}$, while the right side gives the height of the straight chord joining $(\mathbf{x}, f(\mathbf{x}))$ to $(\mathbf{y}, f(\mathbf{y}))$ directly above that point. Let us check a concrete case: for $f(x) = x^2$ with $x = 0$, $y = 2$, and $\theta = \tfrac{1}{2}$, the graph sits at $f(1) = 1$ while the chord sits at $\tfrac{1}{2}\cdot 0 + \tfrac{1}{2}\cdot 4 = 2$, and indeed $1 \le 2$. For arbitrary x, y, and θ, expanding the squares shows that the height gap between chord and graph is

$$\theta x^2 + (1-\theta)y^2 - \big(\theta x + (1-\theta)y\big)^2 = \theta(1-\theta)(x-y)^2 \;\geq\; 0.$$

Thus x^2 is convex, with a gap that grows quadratically with the separation between the two points. We will meet this exact expression again shortly as the variance in Jensen's inequality.

Two nearby notions are worth naming right away. If the inequality Equation 35.25 is strict whenever $\mathbf{x} \neq \mathbf{y}$ and $\theta \in (0,1)$, we call f **strictly convex**; if $-f$ is convex, we call f **concave**. Notice that a linear function $f(x) = x$ is convex (and concave) but not strictly convex, because every chord lies directly on its graph. Notice also why the domain C must be convex: without that requirement, the intermediate point $\theta\mathbf{x} + (1-\theta)\mathbf{y}$ might not even belong to the domain where f is defined.

Convex functions and convex sets are two sides of the same geometric coin. The **epigraph** of f is the region on and above its graph, $\{(\mathbf{x}, t) : t \geq f(\mathbf{x})\}$. A function f is convex if and only if its epigraph is a convex set, because a chord between two points of the epigraph stays above the graph precisely when Equation 35.25 holds. Convexity also passes from functions to sets via their **sublevel sets** $\{\mathbf{x} \in C : f(\mathbf{x}) \leq c\}$: if $f(\mathbf{x}) \leq c$ and $f(\mathbf{y}) \leq c$, the chord inequality immediately gives $f(\theta\mathbf{x} + (1-\theta)\mathbf{y}) \leq \theta c + (1-\theta)c = c$. This is why any inequality constraint $g(\mathbf{x}) \leq 0$ with a convex g automatically carves out a convex feasible region. Figure 35.9(a) shows both sets for a parabola.

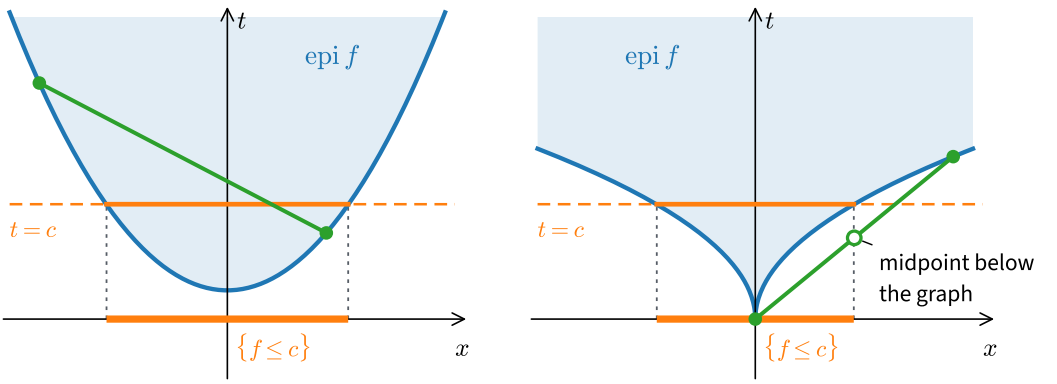


Figure 35.9: Epigraphs and sublevel sets. (a) The epigraph of the convex function $f(x) = \frac{1}{2}x^2 + \frac{1}{4}$ (shaded, continuing upward without bound) is a convex set, so the chord between any two of its points stays inside it. Cutting the epigraph at height c and projecting onto the x-axis gives the sublevel set $\{f \leq c\}$. (b) Every sublevel set of $f(x) = \sqrt{|x|}$ is an interval, yet the epigraph is not convex: the midpoint of the chord lies below the graph.

Be careful, though: the converse does not hold. Every sublevel set of $f(x) = \sqrt{|x|}$ is a convex interval $\{|x| \leq c^2\}$, yet f itself is not convex. At $x = 0$, $y = 1$, and $\theta = \frac{1}{2}$, the graph sits at $\sqrt{1/2} \approx 0.71$ while the chord sits below it at 0.5 (Figure 35.9(b)).

35.3.2.2 Local Minima Are Global

Remarkably, the chord condition alone is already enough to answer the question we posed at the opening. Figure 35.10 contrasts the two landscapes: a single basin on a convex objective

versus two competing basins on the double well.

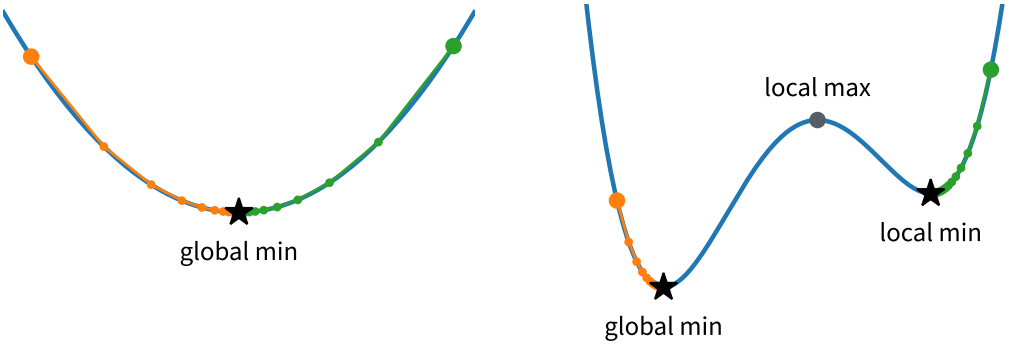


Figure 35.10: (a) On a convex objective, gradient descent with a small fixed step reaches the global minimizer from either start. (b) On a double well, the same update from different starts reaches different minima; the basins meet at the local maximum.

Proposition (local minima are global). *If f is convex on a convex set C, every local minimizer of f on C is a global minimizer.*

Proof. Let $\mathbf{x}^\star$ be a local minimizer and let $\mathbf{y} \in C$ be any other point. For sufficiently small $\theta > 0$, the point $\mathbf{z}_\theta = (1-\theta)\mathbf{x}^\star + \theta\mathbf{y}$ along the segment toward $\mathbf{y}$ falls inside the neighborhood where $\mathbf{x}^\star$ is minimal. Combining local minimality with the chord inequality gives

$$f(\mathbf{x}^\star) \;\le\; f(\mathbf{z}_\theta) \;\le\; (1-\theta) f(\mathbf{x}^\star) + \theta f(\mathbf{y}).$$

Subtract $(1-\theta) f(\mathbf{x}^\star)$ from both sides and divide by $\theta > 0$ to obtain $f(\mathbf{x}^\star) \le f(\mathbf{y})$. ■

The geometric picture is just as crisp: if any strictly better point $\mathbf{y}$ existed anywhere in C, the chord from $\mathbf{x}^\star$ toward $\mathbf{y}$ would slope downward immediately, forcing the graph below $f(\mathbf{x}^\star)$ arbitrarily close to $\mathbf{x}^\star$ and contradicting local minimality.

Let us now run the experiment from the opening and see how the proposition plays out numerically. We apply the same gradient-descent loop (step size 0.05, 600 steps) to both the bowl $(x - \tfrac{1}{2})^2$ and the double well ϕ, starting from the same 500 points drawn uniformly on $[-1.5, 1.5]$. Because the step is small, each iterate rolls downhill without hopping over the central hump, so a run on ϕ should land in whichever well lies on its own side of the local maximum. We compute that local maximum as a root of $\phi'(x) = 4x^3 - 4x + \tfrac{1}{2}$ and check whether the fraction of starts to its right matches the fraction of runs that end in the shallower right well.

```
def gd(grad_fn, x, eta=0.05, steps=600):
    for _ in range(steps):
        x = x - eta * grad_fn(x)
    return x

phi = lambda x: (x**2 - 1)**2 + 0.5 * x        # tilted double well
dphi = lambda x: 4 * x * (x**2 - 1) + 0.5      # its derivative
rng = np.random.default_rng()
inits = rng.uniform(-1.5, 1.5, size=500)       # 500 random starting points
```

```
bowl = gd(lambda x: 2 * (x - 0.5), inits)      # convex: (x - 1/2)^2
well = gd(dphi, inits)
left, right = well[well < 0], well[well > 0]
print(f'bowl: all {bowl.size} runs end at x = {bowl.mean():.6f}'
      f' (spread {np.ptp(bowl):.0e})')
for name, w in (('left ', left), ('right', right)):
    print(f'well, {name}: {w.size} runs, x = {w.mean():.4f}, phi = {phi(w.mean()):.4f}')
x_max = np.sort(np.roots([4, 0, -4, 0.5]).real)[1]   # local max of phi
print(f'local max at {x_max:.4f}: predicted share right {(1.5 - x_max) / 3:.3f},'
      f' observed {right.size / inits.size:.3f}')
```

```
bowl: all 500 runs end at x = 0.500000 (spread 7e-16)
well, left : 263 runs, x = -1.0575, phi = -0.5148
well, right: 237 runs, x = 0.9304, phi = 0.4833
local max at 0.1271: predicted share right 0.458, observed 0.474
```

On the bowl, all 500 runs converge to $x = 0.5$ to within numerical rounding. On the double well, the runs split cleanly into two camps: one at $x \approx -1.0575$ with $\phi \approx -0.515$, and another at $x \approx 0.9304$ with $\phi \approx 0.483$—nearly a full unit worse. The local maximum separating the two basins sits at $x \approx 0.127$, so a fraction $(1.5 - 0.127)/3 \approx 0.458$ of uniform starts fall to its right. Across random draws, the observed share hovers within a few percentage points of 46%, matching the prediction up to the sampling noise of 500 starts.

Both experiments ran the exact same optimizer; only the objective changed. Convexity eliminates the second basin, though it does not by itself guarantee that gradient descent will reach the minimizer—that still requires smoothness and a safe step size, as in Section 35.1.1.

35.3.2.3 The First-Order Condition

The chord condition has the virtue of needing no derivatives, but optimization proofs are built out of gradients. When f is differentiable, convexity takes an equivalent first-order form: the tangent hyperplane at any point is a *global under-estimator* of the function,

$$f(\mathbf{y}) \;\geq\; f(\mathbf{x}) + \nabla f(\mathbf{x})^{\top}(\mathbf{y} - \mathbf{x}) \qquad \text{for all } \mathbf{x}, \mathbf{y} \in C. \tag{35.26}$$

Check this on $f(x) = x^2$ at $x = 1$: the tangent line is $1 + 2(y - 1) = 2y - 1$, and the vertical gap between graph and tangent is $y^2 - (2y - 1) = (y - 1)^2 \geq 0$ for every y. Pause to appreciate how strong a statement Equation 35.26 is: a single gradient, computed at a single point $\mathbf{x}$, bounds f from below at every other point in the domain. Almost every proof in the rest of this section rests on this one inequality. Figure 35.11 places the first two tests side by side: chords lie above the graph, while tangents lie below it.

35.3.2.4 The Second-Order Condition

For smooth objectives, the quickest way to verify convexity is usually a matrix test. When f is twice differentiable on an **open** convex domain C, f is convex if and only if its Hessian is positive semidefinite everywhere:

$$\nabla^2 f(\mathbf{x}) \succeq 0 \qquad \text{for all } \mathbf{x} \in C.$$

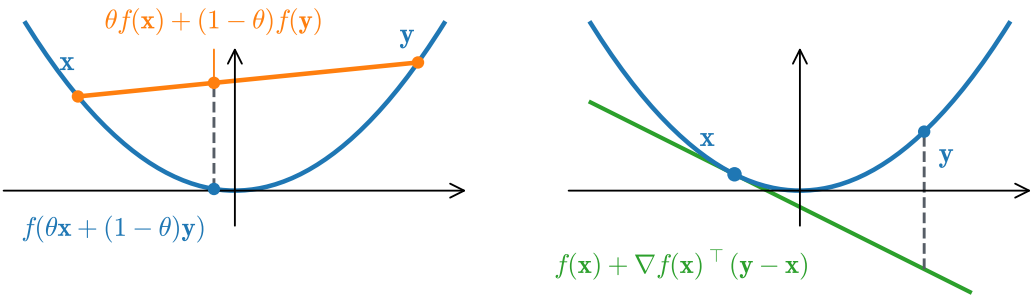


Figure 35.11: Two equivalent conditions. (a) The chord joining two points of the graph lies above the graph: $f(\theta\mathbf{x}+(1-\theta)\mathbf{y}) \le \theta f(\mathbf{x}) + (1-\theta) f(\mathbf{y})$. (b) The tangent at any point lies below the graph: $f(\mathbf{y}) \ge f(\mathbf{x}) + \nabla f(\mathbf{x})^\top (\mathbf{y} - \mathbf{x})$.

In one dimension this reduces to the familiar calculus rule $f''(x) \ge 0$: the curve bends upward. In d dimensions it says that every one-dimensional slice through $\mathbf{x}$ bends upward, since $\mathbf{v}^\top \nabla^2 f(\mathbf{x})\, \mathbf{v}$ is the second derivative of f along the line in direction $\mathbf{v}$ (Section 34.2).

This matrix test settles three workhorse examples immediately. First, a quadratic $\frac{1}{2}\mathbf{x}^\top \mathbf{A}\mathbf{x} + \mathbf{b}^\top \mathbf{x}$ with symmetric $\mathbf{A}$ has constant Hessian $\mathbf{A}$, so it is convex if and only if $\mathbf{A} \succeq 0$. Second, the least-squares loss $\frac{1}{2}\|\mathbf{X}\mathbf{w} - \mathbf{y}\|^2$ for a design matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$ (n examples, d weights) has Hessian $\mathbf{X}^\top \mathbf{X}$; because $\mathbf{v}^\top \mathbf{X}^\top \mathbf{X} \mathbf{v} = \|\mathbf{X}\mathbf{v}\|^2 \ge 0$ for every $\mathbf{v}$, least squares is always convex (Section 33.3.2.2). Third, the scalar functions e^x, $-\log x$, and $x \log x$ are all convex on their domains, since their second derivatives e^x, $1/x^2$, and $1/x$ are positive everywhere.

Why did we insist that the domain C be open? On a lower-dimensional domain, only directions that stay inside C matter: for instance, $x^2 - y^2$ is convex when restricted to the line $y = 0$, even though its full two-dimensional Hessian is indefinite.

We now verify that all three tests single out the exact same class of functions. The proof has four short parts, and the second is the one worth remembering: applying the tangent bound at the *intermediate* point of a chord makes the gradient terms cancel—the same trick we will use to prove Jensen's inequality.

Proposition (the three characterizations agree). *Let f be differentiable on an open convex domain C. Then the chord condition Equation 35.25 and the first-order condition Equation 35.26 are equivalent. If f is twice differentiable, both are equivalent to $\nabla^2 f(\mathbf{x}) \succeq 0$ on C.*

Proof. *(Chord $\Rightarrow$ first-order.)* Fix $\mathbf{x}, \mathbf{y} \in C$ and $\theta \in (0, 1]$. Apply the chord inequality at the intermediate point $\mathbf{x} + \theta(\mathbf{y} - \mathbf{x})$, subtract $f(\mathbf{x})$ from both sides, and divide by θ:

$$\frac{f(\mathbf{x} + \theta(\mathbf{y} - \mathbf{x})) - f(\mathbf{x})}{\theta} \;\le\; f(\mathbf{y}) - f(\mathbf{x}).$$

Letting $\theta \downarrow 0$, the left side approaches the directional derivative $\nabla f(\mathbf{x})^\top (\mathbf{y} - \mathbf{x})$, yielding Equation 35.26.

(First-order $\Rightarrow$ chord.) Set $\mathbf{z} = \theta\mathbf{x} + (1-\theta)\mathbf{y}$ and apply the tangent lower bound at $\mathbf{z}$ twice, once toward each endpoint:

$$f(\mathbf{x}) \geq f(\mathbf{z}) + \nabla f(\mathbf{z})^\top (\mathbf{x} - \mathbf{z}), \qquad f(\mathbf{y}) \geq f(\mathbf{z}) + \nabla f(\mathbf{z})^\top (\mathbf{y} - \mathbf{z}).$$

Multiply the first inequality by θ, the second by $1 - \theta$, and add. The linear terms combine to $\nabla f(\mathbf{z})^\top (\theta \mathbf{x} + (1-\theta)\mathbf{y} - \mathbf{z}) = 0$, leaving Equation 35.25.

(Second-order $\Rightarrow$ first-order.) Restrict f to the line segment between $\mathbf{x}$ and $\mathbf{y}$ by defining $g(t) = f(\mathbf{x} + t(\mathbf{y} - \mathbf{x}))$. Its second derivative is $g''(t) = (\mathbf{y} - \mathbf{x})^\top \nabla^2 f(\mathbf{x} + t(\mathbf{y} - \mathbf{x}))\,(\mathbf{y} - \mathbf{x}) \geq 0$. By Taylor's theorem with the Lagrange remainder (Section 34.1), $g(1) = g(0) + g'(0) + \frac{1}{2} g''(\tau)$ for some $\tau \in (0, 1)$. Dropping the nonnegative remainder yields Equation 35.26.

(First-order $\Rightarrow$ second-order.) Suppose for contradiction that $\mathbf{v}^\top \nabla^2 f(\mathbf{x})\, \mathbf{v} < 0$ for some $\mathbf{x} \in C$ and direction $\mathbf{v}$. Expanding f along $\mathbf{v}$ gives

$$f(\mathbf{x} + t\mathbf{v}) = f(\mathbf{x}) + t\, \nabla f(\mathbf{x})^\top \mathbf{v} + \frac{t^2}{2}\, \mathbf{v}^\top \nabla^2 f(\mathbf{x})\, \mathbf{v} + o(t^2).$$

For sufficiently small $t > 0$, the negative quadratic term dominates the $o(t^2)$ remainder, so $f(\mathbf{x} + t\mathbf{v})$ drops strictly below the tangent value $f(\mathbf{x}) + t \nabla f(\mathbf{x})^\top \mathbf{v}$, contradicting Equation 35.26. ■

35.3.2.5 A Numerical Check of the Three Tests

Let us put all three tests to work numerically on two losses: the least-squares objective $f(\mathbf{w}) = \frac{1}{2}\|\mathbf{X}\mathbf{w} - \mathbf{y}\|^2$ with $n = 20$ random examples and $d = 2$ weights, and the double well ϕ from our opening experiment. For least squares we already know the exact formulas for the violations: expanding the squares as we did for x^2 gives the chord gap (graph minus chord) as $-\frac{1}{2}\theta(1-\theta)\|\mathbf{X}(\mathbf{u} - \mathbf{v})\|^2$ and the tangent gap (tangent minus graph) as $-\frac{1}{2}\|\mathbf{X}(\mathbf{v} - \mathbf{u})\|^2$, both nonpositive. For each loss, the code below samples $1{,}000$ random chords and tangents in $[-1.5, 1.5]^d$, reports the largest violation of each kind, and checks the smallest Hessian eigenvalue across the sampled points.

```
X, y = rng.normal(size=(20, 2)), rng.normal(size=20)  # least squares, d = 2
losses = {'least squares': (2, lambda w: 0.5 * ((X @ w - y) ** 2).sum(),
                            lambda w: X.T @ (X @ w - y), lambda w: X.T @ X),
          'double well': (1, lambda w: phi(w[0]), dphi,  # phi'' = 12 x^2 - 4
                          lambda w: np.diag(12 * w**2 - 4))}
print('               worst chord gap  worst tangent gap  smallest Hessian eig')
for name, (d, f, grad, hess) in losses.items():
    U, V = rng.uniform(-1.5, 1.5, size=(2, 1000, d))  # 1000 random pairs
    T = rng.uniform(size=1000)
    chord = [f(t * u + (1 - t) * v) - (t * f(u) + (1 - t) * f(v))
             for u, v, t in zip(U, V, T)]             # graph minus chord
    tangent = [f(u) + grad(u) @ (v - u) - f(v)        # tangent minus graph
               for u, v in zip(U, V)]
    eig = min(np.linalg.eigvalsh(hess(u))[0] for u in U)
    print(f'{name:15s}{max(chord):15.3f}{max(tangent):19.3f}{eig:22.3f}')
i = int(np.argmax(chord))                             # the double well's worst
print(f'worst chord of the double well: from {U[i, 0]:.2f} to {V[i, 0]:.2f}')
```

```
                worst chord gap  worst tangent gap  smallest Hessian eig
least squares           -0.000             -0.003                24.375
double well              0.972              2.997                -4.000
worst chord of the double well: from -0.96 to 1.10
```

For least squares, not a single sampled chord dips below the graph, no tangent rises above it, and the Hessian eigenvalues stay positive, just as the algebra predicted. The double well fails all three checks: its worst chord cuts nearly one unit below the graph between points in the two valleys, its tangents slice through the curve, and its second derivative drops near -4 at the central hump, where $\phi''(0) = -4$. Notice, however, an asymmetry between the tests. Random sampling can *refute* convexity—a single bad chord is a complete counterexample for ϕ—but it can never *prove* convexity; for least squares it only failed to find a violation. What proves convexity for least squares is the Hessian test, because $\mathbf{X}^\top\mathbf{X}$ is the same PSD matrix at every $\mathbf{w}$.

35.3.2.6 Strict and Strong Convexity

Plain convexity still permits arbitrarily flat stretches, and on a flat stretch gradient descent can crawl arbitrarily slowly. To guarantee fast convergence, we need curvature that stays bounded away from zero everywhere. Three simple functions illustrate the hierarchy, shown in Figure 35.12(a). The line $f(x) = x$ is convex but not strictly convex, because its chords sit right on its graph. The quartic x^4 is strictly convex—its second derivative $12x^2$ vanishes only at 0, so its slope increases strictly everywhere—yet $f''(0) = 0$ leaves it completely flat at the minimizer. The quadratic x^2, by contrast, has constant curvature $f''(x) = 2$ at every point.

Strong convexity turns this observation into a quantitative bound. For $\mu > 0$, a function f is μ-**strongly convex** if subtracting a quadratic of curvature μ leaves a convex function—that is, if $h(\mathbf{x}) = f(\mathbf{x}) - \frac{\mu}{2}\|\mathbf{x}\|^2$ is convex. When f is differentiable, applying the first-order condition to h (whose gradient is $\nabla f(\mathbf{x}) - \mu\mathbf{x}$) and using the identity $\|\mathbf{y}\|^2 - \|\mathbf{x}\|^2 - 2\mathbf{x}^\top(\mathbf{y} - \mathbf{x}) = \|\mathbf{y} - \mathbf{x}\|^2$ yields the equivalent inequality

$$f(\mathbf{y}) \;\geq\; f(\mathbf{x}) + \nabla f(\mathbf{x})^\top(\mathbf{y} - \mathbf{x}) + \frac{\mu}{2}\,\|\mathbf{y} - \mathbf{x}\|^2 \qquad \text{for all } \mathbf{x}, \mathbf{y}. \tag{35.27}$$

Geometrically, the graph of f must sit not merely above its tangent line at $\mathbf{x}$, but above a parabola of curvature μ tangent to it there. Applying the second-order condition to h gives the matrix test: a twice-differentiable f is μ-strongly convex if and only if $\nabla^2 f(\mathbf{x}) \succeq \mu\mathbf{I}$ everywhere, where $\mathbf{I}$ is the identity matrix. Thus x^2 is 2-strongly convex, whereas x^4 fails to be strongly convex on any neighborhood of 0 because $f''(0) = 0 < \mu$ for every $\mu > 0$. Notice also a practical bonus: adding the ridge penalty $\frac{\lambda}{2}\|\mathbf{w}\|^2$ to any convex loss immediately makes the sum λ-strongly convex, because subtracting $\frac{\lambda}{2}\|\mathbf{w}\|^2$ recovers the original convex loss. Weight decay does not just shrink parameters; it guarantees strong convexity.

Strong convexity is the natural counterpart to smoothness. Recall from Section 35.1.1 that f is L-**smooth** when its gradient is L-Lipschitz, $\|\nabla f(\mathbf{x}) - \nabla f(\mathbf{y})\| \leq L\|\mathbf{x} - \mathbf{y}\|$; for a convex twice-differentiable function, this means $\nabla^2 f \preceq L\mathbf{I}$. Smoothness places a quadratic ceiling above the graph, $f(\mathbf{y}) \leq f(\mathbf{x}) + \nabla f(\mathbf{x})^\top(\mathbf{y} - \mathbf{x}) + \frac{L}{2}\|\mathbf{y} - \mathbf{x}\|^2$, while strong convexity places a quadratic floor Equation 35.27 beneath it. Together they sandwich f between two tangent parabolas of curvatures μ and L, as illustrated in Figure 35.12(b) for $f(x) = \frac{1}{2}x^2 +$

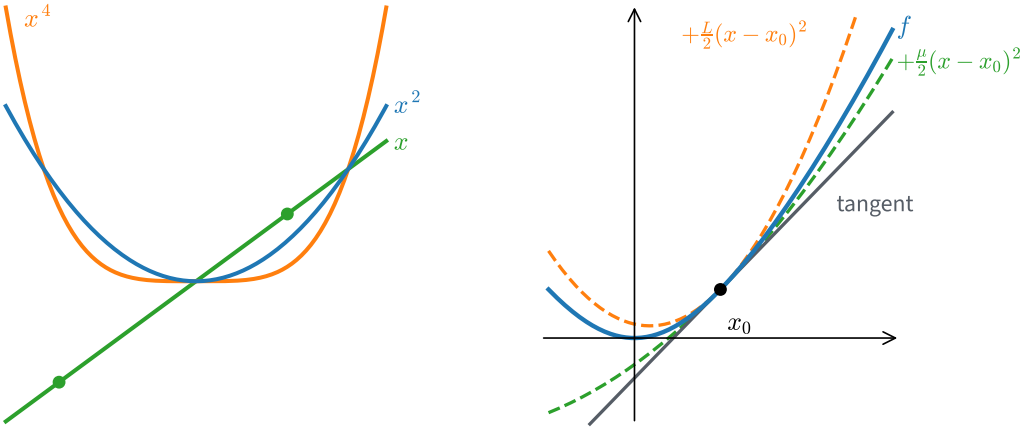


Figure 35.12: (a) Convex, strictly convex, strongly convex: x has chords on its graph, x^4 is flat at the origin, x^2 curves everywhere. (b) A μ-strongly convex, L-smooth function, here $f(x) = \frac{1}{2}x^2 + 3\log\cosh x$, lies between two parabolas tangent at $x_0 = 1$, with curvatures $\mu = 1$ and $L = 4$; their ratio is κ.

$3\log\cosh x$, whose second derivative $f''(x) = 1 + 3/\cosh^2 x$ ranges between $\mu = 1$ and $L = 4$.

The ratio $\kappa = L/\mu$ is the **condition number**, generalizing the eigenvalue ratio we encountered for quadratics in Section 35.1.2. On our running quadratic $\frac{1}{2}(x_1^2 + 1000\,x_2^2)$, the extreme curvatures are $\mu = 1$ and $L = 1000$, giving $\kappa = 10^3$.

35.3.2.7 Subgradients

Important losses such as the ℓ_1 norm $\|\mathbf{x}\|_1$ and the hinge loss $\max(0, 1 - z)$ have sharp corners, where no gradient exists and Equation 35.26 cannot be written as stated. Rather than give up on the tangent bound, we keep the *job* the gradient was doing—producing a global linear under-estimator—and turn that job into a definition. A vector $\mathbf{g}$ is a **subgradient** of f at $\mathbf{x}$ if

$$f(\mathbf{y}) \;\geq\; f(\mathbf{x}) + \mathbf{g}^\top(\mathbf{y} - \mathbf{x}) \qquad \text{for all } \mathbf{y}. \tag{35.28}$$

The set of all subgradients at $\mathbf{x}$ is called the **subdifferential**, written $\partial f(\mathbf{x})$. Wherever a convex function f is differentiable, the subdifferential collapses to a single vector, $\{\nabla f(\mathbf{x})\}$. At a corner, however, it fans out into a whole set of supporting slopes (Figure 35.13). For $f(x) = |x|$ at $x = 0$, every slope $g \in [-1, 1]$ satisfies $|y| \geq gy$ for all y, so $\partial f(0) = [-1, 1]$; similarly, for the hinge loss $\max(0, 1 - z)$ at the kink $z = 1$, we get $\partial f(1) = [-1, 0]$.

This immediately gives a derivative-free optimality test: $\mathbf{x}^\star$ minimizes f if and only if $\mathbf{0} \in \partial f(\mathbf{x}^\star)$. The proof is a one-line tautology: setting $\mathbf{g} = \mathbf{0}$ in Equation 35.28 reads $f(\mathbf{y}) \geq f(\mathbf{x}^\star)$ for all $\mathbf{y}$, which is literally the definition of a global minimizer.

Does a convex function always possess a subgradient? At every interior point of its domain, yes—a fact we will rely on to prove Jensen's inequality. In one dimension the reason is elementary. For any three points $a < b < c$ in an open interval where f is convex, rearranging

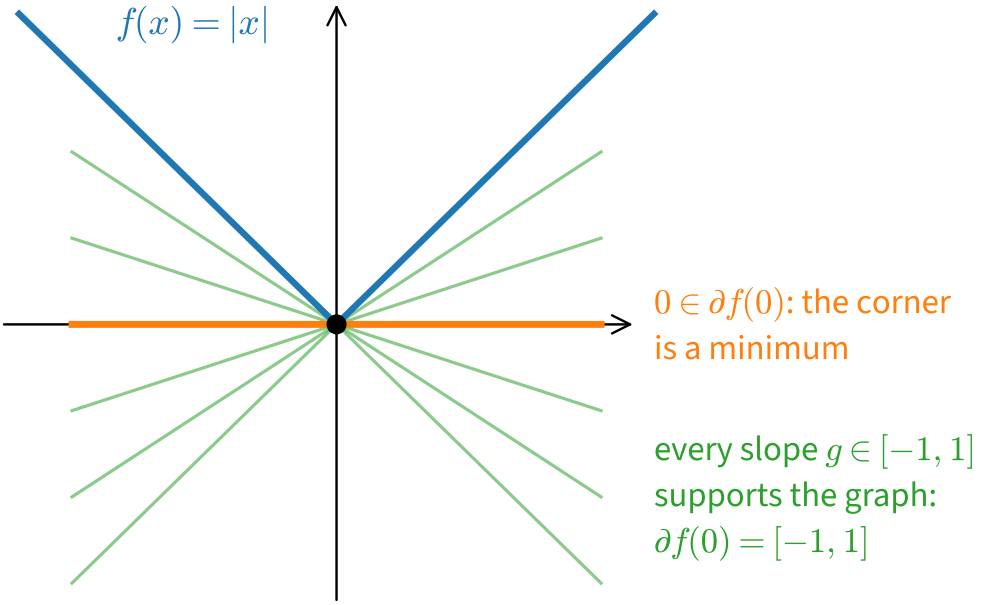


Figure 35.13: At the kink of $f(x) = |x|$ the gradient does not exist, but the subdifferential is a set: every slope $g \in (-1, 1)$ defines a supporting line below the V, and the two extreme slopes ± 1 lie along the branches themselves, so $\partial f(0)$ is the whole interval from -1 to 1. The zero-slope member (orange) is the optimality certificate $0 \in \partial f(0)$: the corner is a minimum certified without differentiability.

the chord inequality at b gives

$$\frac{f(b)-f(a)}{b-a} \;\le\; \frac{f(c)-f(a)}{c-a} \;\le\; \frac{f(c)-f(b)}{c-b},$$

meaning that chord slopes are nondecreasing from left to right. Consequently, the difference quotient $q(x) = (f(x) - f(x_0))/(x - x_0)$ at any interior point x_0 is nondecreasing in x, so its one-sided limits—the left and right derivatives $f'_-(x_0) \le f'_+(x_0)$—both exist and are finite. Every slope $g \in [f'_-(x_0), f'_+(x_0)]$ is a subgradient: for $y > x_0$ we have $q(y) \ge f'_+(x_0) \ge g$, which rearranges to $f(y) \ge f(x_0) + g(y - x_0)$, and the case $y < x_0$ is symmetric.

The same guarantee extends to d dimensions: at every interior point of its domain, a convex function has at least one subgradient, so a supporting hyperplane always fits beneath its graph (Rockafellar (1970); Boyd and Vandenberghe (2004), section 2.5). We will use this result without proof. Keep in mind, though, that as a search direction a subgradient is much weaker than a true gradient: stepping along $-\mathbf{g}$ need not decrease f at all (at the minimum $x = 0$ of $|x|$, the valid subgradient $g = \frac{1}{2}$ steps to $-\eta/2$, raising the loss). For that reason, the subgradient method requires diminishing step sizes and converges slowly, with error falling only as $1/\sqrt{k}$.

35.3.3 Consequences of Convexity

The rest of this book draws on three major consequences of convexity, and all three spring from the same geometric source: a supporting line or hyperplane that bounds the function from below everywhere. Averaged over a random variable, that bound yields Jensen's inequality. Evaluated at a stationary point, it certifies global optimality—sharpening what Section 35.3.2.2 deduced from the chord condition alone. Summed along the steps of gradient descent, it turns local progress into global convergence rates. Jensen's inequality stands on its own, so if you are reading purely for optimization you can skip directly to the next subsection.

35.3.3.1 Jensen's Inequality

Expected losses, average log-likelihoods, and KL divergences all confront the same basic question: how does a function of an average compare with the average of the function? The simplest two-point example shows which way the inequality points. Let Z take the values 0 and 2 with probability $\frac{1}{2}$ each, and let $f(x) = x^2$. Evaluating f at the mean gives $f(\mathbb{E}[Z]) = f(1) = 1$, whereas averaging f over the outcomes gives $\mathbb{E}[f(Z)] = \frac{1}{2}\cdot 0 + \frac{1}{2}\cdot 4 = 2$. The gap of 1 is simply the variance of Z; in fact, for $f(x) = x^2$ the difference $\mathbb{E}[Z^2] - (\mathbb{E}[Z])^2$ is always the variance. Look back at Figure 35.11(a): the chord inequality is making this exact comparison for a two-point distribution with weights θ and $1-\theta$, and the gap $\theta(1-\theta)(x-y)^2$ we computed earlier for x^2 is that distribution's variance.

Nothing about this argument is restricted to two points. Applying Equation 35.25 inductively (Exercise 2) gives the finite form

$$f\left(\sum_i \theta_i \mathbf{x}_i\right) \;\le\; \sum_i \theta_i\, f(\mathbf{x}_i), \qquad \theta_i \ge 0,\ \sum_i \theta_i = 1. \tag{35.29}$$

Because nonnegative weights summing to one define a discrete probability distribution on the points $\mathbf{x}_i$, the natural generalization replaces the finite average with an arbitrary expectation.

Proposition (Jensen's inequality). *Let f be convex on a convex set $C \subseteq \mathbb{R}^d$, and let $\mathbf{Z}$ be a random vector taking values in C, with $\mathbb{E}\|\mathbf{Z}\| < \infty$ and mean $\mathbf{m} = \mathbb{E}[\mathbf{Z}]$ in the interior of C. Then*

$$f(\mathbf{m}) \;\le\; \mathbb{E}\left[f(\mathbf{Z})\right]. \tag{35.30}$$

If f is strictly convex, equality holds if and only if $\mathbf{Z} = \mathbf{m}$ with probability one.

Proof. Because $\mathbf{m}$ lies in the interior of C, f has at least one subgradient $\mathbf{g} \in \partial f(\mathbf{m})$. The supporting hyperplane at $\mathbf{m}$ bounds f for every outcome at once: by Equation 35.28,

$$f(\mathbf{Z}) \;\ge\; f(\mathbf{m}) + \mathbf{g}^{\top}(\mathbf{Z} - \mathbf{m}).$$

Now take expectations of both sides. Because $\mathbb{E}[\mathbf{Z} - \mathbf{m}] = \mathbf{0}$, the linear term vanishes, leaving $\mathbb{E}[f(\mathbf{Z})] \ge f(\mathbf{m})$. When f is strictly convex, this supporting hyperplane touches the graph *only* at $\mathbf{m}$: if it also touched at some $\mathbf{x} \neq \mathbf{m}$, strict convexity would force $f(\frac{\mathbf{x}+\mathbf{m}}{2}) < \frac{1}{2}f(\mathbf{x}) + \frac{1}{2}f(\mathbf{m})$, placing the graph strictly below the supporting hyperplane at the midpoint, a contradiction. Hence the pointwise inequality is strict whenever $\mathbf{Z} \neq \mathbf{m}$, so equality in expectation forces $\mathbf{Z} = \mathbf{m}$ with probability one. ■

As a bonus, the same pointwise bound shows that $f(\mathbf{Z})$ is bounded below by an integrable random variable, so $\mathbb{E}[f(\mathbf{Z})]$ is always well-defined (possibly $+\infty$, in which case the inequality holds trivially). In plain terms, spreading a random variable out while keeping its mean fixed can only increase the expected value of a convex function. For a concave function f, the inequality reverses: $\mathbb{E}[f(\mathbf{Z})] \le f(\mathbb{E}[\mathbf{Z}])$.

In machine learning, the most frequent application of Jensen's inequality is proving that KL divergence is never negative; Section 37.1 meets this same result as Gibbs' inequality.

Corollary (nonnegativity of KL divergence). *For probability distributions p and q on a common finite alphabet, with $q(x) > 0$ wherever $p(x) > 0$,*

$$D_{\mathrm{KL}}(p \,\|\, q) \;=\; \sum_{x} p(x) \log \frac{p(x)}{q(x)} \;\ge\; 0,$$

with equality if and only if $p = q$.

Proof. Draw $Z \sim p$ and apply Jensen's inequality to the strictly convex function $-\log$ and the positive random variable $R = q(Z)/p(Z)$:

$$D_{\mathrm{KL}}(p \,\|\, q) = \mathbb{E}_p\left[-\log R\right] \;\ge\; -\log \mathbb{E}_p[R] = -\log \sum_{x : p(x) > 0} p(x)\, \frac{q(x)}{p(x)} \;\ge\; -\log 1 = 0,$$

where the last step uses $\sum_{x:p(x)>0} q(x) \le 1$. By the equality condition of Jensen's inequality, equality holds if and only if R is constant with probability one and q places no mass outside the support of p—that is, $q = c\,p$ with $c = 1$. ■

Applying the same idea to the concave function log yields the classical **arithmetic–geometric mean inequality**: for any positive numbers $x_1, \dots, x_N$, concavity gives $\log(\frac{1}{N}\sum_i x_i) \geq \frac{1}{N}\sum_i \log x_i$, and exponentiating both sides produces

$$\frac{1}{N}\sum_{i=1}^{N} x_i \;\geq\; \left(\prod_{i=1}^{N} x_i\right)^{1/N},$$

with equality if and only if all x_i are equal, since log is strictly concave. Likewise, the evidence lower bound (ELBO) of Section 36.7, which bounds the logarithm of an expectation from below, is another direct application of Jensen's inequality to log.

Let us test both Jensen's gap and the KL equality case numerically. First, for $Z \sim \mathcal{N}(0,1)$ and $f(x) = e^x$, we can compute both sides in closed form—$\mathbb{E}[e^Z] = \sqrt{e} \approx 1.649$ versus $e^{\mathbb{E}[Z]} = 1$—and see how closely Monte Carlo matches them. Second, as we blend q into p via $(1-\varepsilon)p + \varepsilon q$, how quickly does the KL divergence vanish: linearly in ε, or faster?

```
rng = np.random.default_rng()
z = rng.normal(size=1_000_000)                # Z ~ N(0, 1)
print(f'E[exp(Z)] = {np.exp(z).mean():.2f} vs exp(E[Z]) = {np.exp(z.mean()):.2f}'
      f'  (population: {np.exp(0.5):.4f} vs 1)')
kl = lambda p, q: (p * np.log(p / q)).sum(axis=1)
p = rng.dirichlet(np.ones(10), size=1000)     # 1000 random pairs, 10 symbols
q = rng.dirichlet(np.ones(10), size=1000)
print('median KL(p || (1-eps) p + eps q), eps = 1e-2, 1e-3, 1e-4:',
      *[f'{np.median(kl(p, (1 - e) * p + e * q)):.1e}' for e in (1e-2, 1e-3, 1e-4)])
```

```
E[exp(Z)] = 1.65 vs exp(E[Z]) = 1.00  (population: 1.6487 vs 1)
median KL(p || (1-eps) p + eps q), eps = 1e-2, 1e-3, 1e-4: 1.6e-04 1.6e-06 1.6e-08
```

The Monte Carlo averages land right around 1.65 versus 1.00; drawing more samples shrinks the estimation noise, but the structural Jensen gap remains fixed. In the KL experiment, the divergence vanishes much faster than ε: every tenfold reduction in ε cuts the median divergence roughly a hundredfold. Expanding the logarithm explains why: $D_{\mathrm{KL}}(p \,\|\, (1-\varepsilon)p + \varepsilon q) \approx \frac{\varepsilon^2}{2}\sum_x (q(x)-p(x))^2/p(x)$ once $\varepsilon\, q(x)/p(x)$ is small for every x. (Pairs where p assigns tiny probability to some symbol enter this quadratic regime last, which is why we report the median rather than the maximum.) Equality is approached quadratically, and attained only in the limit $q \to p$.

35.3.3.2 Local and Global Minima

We already proved that every local minimum of a convex function is global (Section 35.3.2.2). Three companion facts follow just as effortlessly, each in a single line. First, the set of all global minimizers is convex: if both $\mathbf{x}$ and $\mathbf{y}$ attain the minimum value $f^\star$, the chord inequality forces every point between them to lie at or below $f^\star$. Second, for a differentiable convex f on an open set, $\nabla f(\mathbf{x}) = \mathbf{0}$ immediately implies $f(\mathbf{x}) = f^\star$, because the first-order condition with a zero gradient reads $f(\mathbf{y}) \geq f(\mathbf{x})$ for all $\mathbf{y}$. Third, a small gradient guarantees a small suboptimality gap: pointing the first-order condition at a minimizer $\mathbf{x}^\star$ gives $f(\mathbf{x}) - f^\star \leq \nabla f(\mathbf{x})^\top(\mathbf{x} - \mathbf{x}^\star) \leq \|\nabla f(\mathbf{x})\|\,\|\mathbf{x} - \mathbf{x}^\star\|$. Thus the stationarity result of Section 35.1.1—that gradient descent with $\eta = 1/L$ drives the smallest gradient norm to zero—upgrades for convex objectives into a guarantee on the *loss value* itself, provided the

iterates stay within a bounded distance of $\mathbf{x}^\star$. In Section 35.3.3.3 we will prove that they do, and pin down the exact rate.

Be aware of two things that plain convexity does *not* guarantee: a minimizer need not be unique (a convex function can be flat along an entire segment), and it need not even exist (e^x approaches 0 as $x \to -\infty$ without ever attaining it). Strict convexity rules out multiple minimizers, and strong convexity guarantees both existence and uniqueness.

Proposition (uniqueness). *A strictly convex f has at most one minimizer. A continuous μ-strongly convex f on a nonempty closed convex set C with nonempty interior has exactly one.*

Proof. Suppose $\mathbf{x} \neq \mathbf{y}$ were both global minimizers with value $f^\star$. By the strict chord inequality at $\theta = \frac{1}{2}$, their midpoint would satisfy $f(\frac{\mathbf{x}+\mathbf{y}}{2}) < f^\star$, a contradiction. Because strong convexity implies strict convexity, it remains only to prove existence. We prove it for a differentiable f on $C = \mathbb{R}^d$, which covers the unconstrained losses in this book; the general case follows the same steps using a subgradient of $f - \frac{\mu}{2}\|\cdot\|^2$ at an interior point of C. Fix any starting point $\mathbf{x}_0$. In Equation 35.27, the linear term is at least $-\|\nabla f(\mathbf{x}_0)\|\,\|\mathbf{y} - \mathbf{x}_0\|$, so $f(\mathbf{y}) > f(\mathbf{x}_0)$ whenever $\|\mathbf{y} - \mathbf{x}_0\| > 2\|\nabla f(\mathbf{x}_0)\|/\mu$. Hence the sublevel set $\{\mathbf{y} : f(\mathbf{y}) \leq f(\mathbf{x}_0)\}$ is bounded, and continuity of f ensures that it is closed. Every continuous function on a nonempty closed and bounded set attains its minimum, and the minimum over this sublevel set is automatically the global minimum over $\mathbb{R}^d$. ■

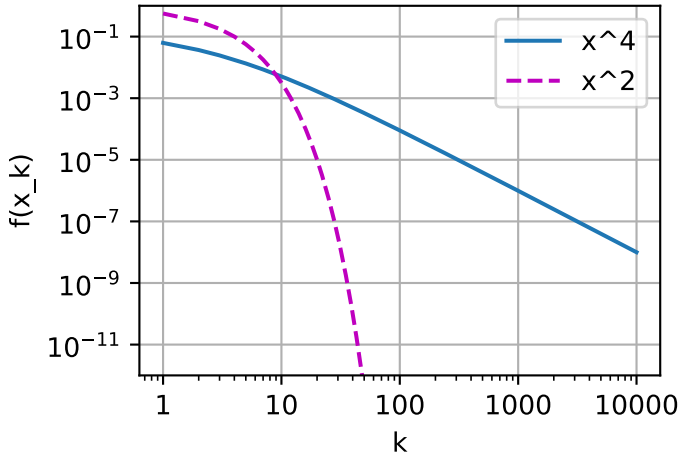
The technical hypotheses on C and continuity merely exclude pathological functions that jump upward at the boundary of their domain, which never occur for the losses in this book. By contrast, the missing minimizer of e^x shows up in real machine learning: Exercise 3 traces it directly to logistic regression on linearly separable data.

35.3.3.3 Convergence Rates

Convexity tells us *where* gradient descent stops, but not *how fast* it gets there—and on speed, plain convexity promises surprisingly little. Compare $f(x) = x^2$ and $f(x) = x^4$: both are strictly convex with unique minimizer 0, yet only x^2 is strongly convex. Let us run the exact same gradient-descent loop ($\eta = \frac{1}{8}$, starting from $x_0 = 1$) on both functions for 10^4 steps and watch how differently they behave.

```
eta, steps = 1 / 8, 10_000
x2, x4 = np.ones(steps + 1), np.ones(steps + 1)  # iterates from x_0 = 1
for j in range(steps):                           # same start, same step
    x2[j + 1] = x2[j] - eta * 2 * x2[j]          # f(x) = x^2
    x4[j + 1] = x4[j] - eta * 4 * x4[j]**3       # f(x) = x^4
k = np.arange(1, steps + 1)
print('x^2: first k with x_k < 1e-6:', k[x2[1:] < 1e-6][0])
print('x^4: x_k * sqrt(k) at k = 1e2, 1e3, 1e4:',
      (x4[1:] * np.sqrt(k))[[99, 999, 9999]].round(3))
d2l.plot(k, [x4[1:]**4, x2[1:]**2], 'k', 'f(x_k)', xscale='log',
         yscale='log', ylim=(1e-12, 1), legend=['x^4', 'x^2'])
```

```
x^2: first k with x_k < 1e-6: 49
x^4: x_k * sqrt(k) at k = 1e2, 1e3, 1e4: [0.973 0.996 1.   ]
```



The contrast is dramatic. On x^2, every step multiplies the iterate by a constant factor $1 - 2\eta = 0.75$, driving it below 10^{-6} in just 49 steps; the loss drops off the bottom of the plot almost immediately. On x^4, by contrast, each step multiplies x_k by $1 - 4\eta x_k^2$, a contraction factor that creeps closer and closer to 1 as the iterate approaches the flat valley at 0. The printed products $x_k\sqrt{k}$ settle near 1, showing that x_k decays only like $1/\sqrt{k}$ and still sits at 0.01 after $10{,}000$ steps. On a log–log plot, the loss gap x_k^4 therefore traces a straight line of slope -2. Strict convexity rules out multiple minimizers, but it cannot guarantee fast convergence; a geometric rate requires $\mu > 0$.

The next two propositions make this contrast completely general: every smooth strongly convex function converges geometrically like x^2, whereas without $\mu > 0$ we can guarantee only an $O(1/k)$ decay. Both proofs begin with the **descent lemma** from Section 35.1.1: for an L-smooth function f and the gradient step $\mathbf{x}_{k+1} = \mathbf{x}_k - \frac{1}{L}\nabla f(\mathbf{x}_k)$,

$$f(\mathbf{x}_{k+1}) \;\le\; f(\mathbf{x}_k) - \frac{1}{2L}\,\|\nabla f(\mathbf{x}_k)\|^2.$$

Each step reduces the objective by at least $\|\nabla f(\mathbf{x}_k)\|^2/(2L)$. On a general nonconvex function, that only proves the gradient shrinks to zero. Convexity supplies the missing link between the gradient and the suboptimality gap: whenever $f(\mathbf{x}_k) - f^\star$ is still large, the gradient must be large enough to keep making progress, so the iterates cannot stall far from the optimum.

Proposition (gradient descent on smooth convex functions). *Let f be convex and L-smooth with a minimizer $\mathbf{x}^\star$. Gradient descent with $\eta = 1/L$ satisfies, for every $k \ge 1$,*

$$f(\mathbf{x}_k) - f(\mathbf{x}^\star) \;\le\; \frac{L\,\|\mathbf{x}_0 - \mathbf{x}^\star\|^2}{2k}. \tag{35.31}$$

Thus the loss gap decays at least as fast as $1/k$, scaled by the smoothness constant and the squared initial distance to the minimizer. To put numbers on it, if $L = 1$ and $\|\mathbf{x}_0 - \mathbf{x}^\star\| = 10$, the bound is $50/k$, guaranteeing a gap below 10^{-2} within $5{,}000$ steps.

Proof. Combine the descent lemma with the first-order condition Equation 35.26 evaluated at $\mathbf{x}_k$ toward the optimum, $f(\mathbf{x}_k) - f(\mathbf{x}^\star) \le \nabla f(\mathbf{x}_k)^\top(\mathbf{x}_k - \mathbf{x}^\star)$:

$$f(\mathbf{x}_{k+1}) - f(\mathbf{x}^\star) \;\le\; \nabla f(\mathbf{x}_k)^\top(\mathbf{x}_k - \mathbf{x}^\star) - \frac{1}{2L}\|\nabla f(\mathbf{x}_k)\|^2.$$

Now comes the clever step: complete the square so that the right side turns into a difference of two squared distances to $\mathbf{x}^\star$,

$$\begin{aligned}\nabla f(\mathbf{x}_k)^\top(\mathbf{x}_k - \mathbf{x}^\star) - \frac{1}{2L}\|\nabla f(\mathbf{x}_k)\|^2 \\ = \frac{L}{2}\left(\|\mathbf{x}_k - \mathbf{x}^\star\|^2 - \left\|\mathbf{x}_k - \mathbf{x}^\star - \tfrac{1}{L}\nabla f(\mathbf{x}_k)\right\|^2\right).\end{aligned}$$

Notice that the second norm is simply $\|\mathbf{x}_{k+1} - \mathbf{x}^\star\|^2$. Summing over $j = 0, \dots, k-1$, the right side telescopes cleanly:

$$\begin{aligned}\sum_{j=0}^{k-1}\left(f(\mathbf{x}_{j+1}) - f(\mathbf{x}^\star)\right) &\le \frac{L}{2}\left(\|\mathbf{x}_0 - \mathbf{x}^\star\|^2 - \|\mathbf{x}_k - \mathbf{x}^\star\|^2\right) \\ &\le \frac{L}{2}\|\mathbf{x}_0 - \mathbf{x}^\star\|^2.\end{aligned}$$

Because the descent lemma ensures that $f(\mathbf{x}_j)$ never increases, the final term $f(\mathbf{x}_k) - f(\mathbf{x}^\star)$ is the smallest term in the sum, so $k\,(f(\mathbf{x}_k) - f(\mathbf{x}^\star))$ is bounded by the total sum. Dividing by k gives Equation 35.31. ■

Two features of this proof are worth noting. First, the dimension d never appears: a model with a billion weights and the same L and initial distance enjoys the exact same worst-case guarantee. Second, before summing, the per-step inequality has a nonnegative left side $f(\mathbf{x}_{k+1}) - f^\star \ge 0$, which forces $\|\mathbf{x}_{k+1} - \mathbf{x}^\star\| \le \|\mathbf{x}_k - \mathbf{x}^\star\|$: every single step moves monotonically closer to every minimizer.

Adding strong convexity transforms this $1/k$ decay into a geometric rate.

Proposition (linear rate under strong convexity). *Let f be μ-strongly convex and L-smooth, with minimizer $\mathbf{x}^\star$. Gradient descent with $\eta = 1/L$ satisfies*

$$f(\mathbf{x}_k) - f(\mathbf{x}^\star) \;\le\; \left(1 - \frac{\mu}{L}\right)^k \left(f(\mathbf{x}_0) - f(\mathbf{x}^\star)\right). \tag{35.32}$$

In words, every iteration shaves off at least a fraction $\mu/L = 1/\kappa$ of the remaining loss gap.

Proof. The proof splits into two short steps. First, strong convexity bounds the squared gradient norm from below in terms of the suboptimality gap. Minimize both sides of Equation 35.27 over $\mathbf{y}$: the left side attains its minimum $f(\mathbf{x}^\star)$, while the right side is a simple quadratic in $\mathbf{y}$ minimized at $\mathbf{y} = \mathbf{x} - \frac{1}{\mu}\nabla f(\mathbf{x})$, where its value is $f(\mathbf{x}) - \frac{1}{2\mu}\|\nabla f(\mathbf{x})\|^2$. Rearranging gives

$$\frac{1}{2}\|\nabla f(\mathbf{x})\|^2 \;\ge\; \mu\left(f(\mathbf{x}) - f(\mathbf{x}^\star)\right) \qquad \text{for all } \mathbf{x}. \tag{35.33}$$

Second, substitute Equation 35.33 directly into the descent lemma:

$$f(\mathbf{x}_{k+1}) - f(\mathbf{x}^\star) \le f(\mathbf{x}_k) - f(\mathbf{x}^\star) - \frac{1}{2L}\|\nabla f(\mathbf{x}_k)\|^2$$
$$\le \left(1 - \frac{\mu}{L}\right)\left(f(\mathbf{x}_k) - f(\mathbf{x}^\star)\right).$$

Applying this contraction k times completes the proof. ■

Look closely at the second half of that proof: it never invoked convexity at all, only the inequality Equation 35.33. We will return to that surprising fact in Section 35.3.5. Notice also the enormous speedup over the non-strongly-convex case: reaching relative accuracy ϵ now takes roughly $\kappa \log(1/\epsilon)$ steps instead of $O(1/\epsilon)$. On our running quadratic with $\kappa = 10^3$, gaining six digits of accuracy, $(1 - 10^{-3})^k \le 10^{-6}$, takes about $13{,}800$ steps by this bound.

How tight is Equation 35.32 in practice? Let us measure it on a least-squares problem whose Hessian $\mathbf{X}^\top\mathbf{X}$ has eigenvalues 1 and 10 by construction ($\mu = 1$, $L = 10$, $\kappa = 10$). We run gradient descent with $\eta = 1/L$ and track the actual per-step ratio of successive loss gaps, $(f(\mathbf{w}_{k+1}) - f^\star)/(f(\mathbf{w}_k) - f^\star)$.

```
rng = np.random.default_rng()
Q, _ = np.linalg.qr(rng.normal(size=(20, 2)))  # orthonormal columns
R, _ = np.linalg.qr(rng.normal(size=(2, 2)))   # random rotation
X = Q @ np.diag([1.0, np.sqrt(10.0)]) @ R      # X^T X has eigenvalues 1, 10
y = rng.normal(size=20)
f = lambda w: 0.5 * ((X @ w - y) ** 2).sum()
mu, L = np.linalg.eigvalsh(X.T @ X)[[0, -1]]
w_star = np.linalg.lstsq(X, y, rcond=None)[0]
w, gaps = np.zeros(2), [f(np.zeros(2)) - f(w_star)]
for _ in range(60):                            # gradient descent, eta = 1/L
    w = w - X.T @ (X @ w - y) / L
    gaps.append(f(w) - f(w_star))
r = np.array(gaps[1:]) / np.array(gaps[:-1])   # per-step contraction of f - f*
print(f'mu = {mu:.4f}, L = {L:.4f}: bound 1 - mu/L = {1 - mu / L:.4f},'
      f' its square {(1 - mu / L)**2:.4f}')
print('first three ratios:', r[:3].round(4), ' largest ratio:', r.max().round(4))
```

```
mu = 1.0000, L = 10.0000: bound 1 - mu/L = 0.9000, its square 0.8100
first three ratios: [0.6084 0.81   0.81  ]  largest ratio: 0.81
```

Our bound promises a contraction factor of 0.9 per step, yet from the second step onward the measured ratio is 0.81—which is $(1 - \mu/L)^2$. (The very first ratio depends on the random starting direction and varies from run to run.) Viewing the update in the eigenvector coordinates of $\mathbf{X}^\top\mathbf{X}$ explains what happened. With $\eta = 1/L$, the first step multiplies the stiff eigen-component of $\mathbf{w} - \mathbf{w}^\star$ by $1 - L/L = 0$, wiping it out completely. In two dimensions, only the slow component remains; its distance to $\mathbf{w}^\star$ shrinks by $1 - \mu/L$ per step, so the loss gap—which is quadratic in that distance—shrinks by $(1 - \mu/L)^2$. In higher dimensions, the ratio climbs toward $(1 - \mu/L)^2$ as the faster eigen-components decay.

Our short proof gave away a square in the contraction factor. A sharper analysis shows that gradient descent with $\eta = 1/L$ in fact contracts $f - f^\star$ by $(1 - \mu/L)^2$ per step on *every* L-smooth μ-strongly convex function (Taylor et al. 2018), with quadratics achieving equality; Exercise 5 proves the closely related bound $\kappa(1-\mu/L)^{2k}$. Returning to our $\kappa = 10^3$ running

quadratic, six digits of accuracy in the loss gap (three digits in the distance to the minimizer) cost roughly $13{,}800$ steps by our simple bound, about $6{,}900$ steps at the true $(1-\mu/L)^2$ rate, and half as many, $3{,}454$ steps, at the optimal fixed step $\eta = 2/(\mu+L)$. This is half of the $6{,}908$ steps that Section 35.1 counted at the same step for six digits in the distance. One factor of two came from the looseness of our proof, and the second from doubling the step size.

35.3.4 Recognizing Convexity

Global optimality and clean convergence rates are useful only if we can tell whether a given loss function is convex in the first place. Computing Hessians by hand quickly becomes tedious (and fails altogether at kinks), so in practice we rarely check convexity from scratch. Instead, we recognize convex objectives by assembling them from simple convex building blocks.

35.3.4.1 A Calculus of Convex Functions

Four elementary operations preserve convexity, and almost every convex loss in machine learning is built from basic convex atoms using just these four rules.

1. **Nonnegative weighted sums.** If $f_1, \dots, f_m$ are convex and $w_i \ge 0$, then $\sum_i w_i f_i$ is convex: simply multiply the chord inequality for each f_i by w_i and add. In particular, averaging a convex per-example loss over a training set always yields a convex objective.

2. **Affine pre-composition.** If f is convex, then so is $\mathbf{x} \mapsto f(\mathbf{A}\mathbf{x}+\mathbf{b})$: an affine map carries the line segment between $\mathbf{x}$ and $\mathbf{y}$ onto the segment between their images with the exact same weight θ, so the chord inequality transfers directly. As a result, any loss that is convex in a model's *predictions* is automatically convex in the *weights of a linear model* producing those predictions.

3. **Pointwise maximum and supremum.** If every f_i is convex, then so is $\max_i f_i$. The quickest proof uses epigraphs: the epigraph of a pointwise maximum is the intersection of the individual epigraphs, and intersections of convex sets are convex. Consequently, the upper envelope of any collection of affine functions is convex, no matter how large the collection.

4. **Monotone composition.** If g is convex and h is both convex *and nondecreasing*, then the composition $h \circ g$ is convex. Two lines prove it:

$$h(g(\theta\mathbf{x}+(1-\theta)\mathbf{y})) \;\le\; h(\theta g(\mathbf{x})+(1-\theta)g(\mathbf{y})) \;\le\; \theta\, h(g(\mathbf{x})) + (1-\theta)\, h(g(\mathbf{y})),$$

   where the first inequality uses convexity of g together with monotonicity of h, and the second uses convexity of h. Do not forget the monotonicity requirement on h: $h(t)=t^2$ is convex but not nondecreasing, and composing it with the convex function $g(x)=x^2-1$ produces the nonconvex double well $(x^2-1)^2$ (which becomes ϕ after adding $x/2$).

These four rules certify nearly all the standard convex losses in one sweep. The hinge loss $\max(0, 1-y\,\mathbf{w}^\top\mathbf{x})$ is the maximum of two affine functions of $\mathbf{w}$ (rules 3 and 2). The ℓ_1 norm $\|\mathbf{w}\|_1 = \sum_i \max(w_i, -w_i)$ is a sum of maxima of affine functions (rules 1, 3, and 2).

The logistic loss $\log(1 + e^{-y\,\mathbf{w}^\top \mathbf{x}})$ starts from the scalar function $t \mapsto \log(1 + e^t)$—convex because its second derivative $e^t/(1 + e^t)^2$ is strictly positive—pre-composed with an affine map (rule 2) and summed over training examples (rule 1). Adding the ridge penalty $\frac{\lambda}{2}\|\mathbf{w}\|^2$ to any of these produces a strongly convex objective. Notice, however, that convexity does not automatically upgrade to strong convexity under rule 3: $\max(x^2, 1 - x)$ is convex, yet it equals the flat affine line $1 - x$ across the entire interval $(-1.618, 0.618)$ between the two roots of $x^2 = 1 - x$, so it is not strongly convex.

Notice what is conspicuously absent from our list: composition with an inner map that is neither affine nor convex. A two-layer neural network $\mathbf{W}_2\,\mathrm{relu}(\mathbf{W}_1\mathbf{x})$ is precisely such a composition in its weights, which is the exact point where convexity breaks down (Section 35.3.5).

35.3.4.2 Log-Sum-Exp and the Softmax Covariance

One fundamental loss sits just outside the four rules above: the log-sum-exp function $\mathrm{lse}(\mathbf{x}) = \log \sum_{i=1}^{d} e^{x_i}$ at the heart of every softmax cross-entropy loss. Geometrically, $\mathrm{lse}(\mathbf{x})$ is a smooth upper approximation to $\max_i x_i$ that never overshoots it by more than $\log d$. Rather than force it into the composition rules, we can certify its convexity directly through its Hessian—which turns out to have a remarkably clean probabilistic interpretation.

Proposition (log-sum-exp is convex). *The Hessian of* lse *at* $\mathbf{x}$ *is*

$$\nabla^2 \mathrm{lse}(\mathbf{x}) = \mathrm{diag}(\mathbf{s}) - \mathbf{s}\mathbf{s}^\top, \qquad \mathbf{s} = \mathrm{softmax}(\mathbf{x}), \tag{35.34}$$

which is the covariance matrix of the one-hot encoding of a class drawn from $\mathbf{s}$*. Since every covariance matrix is PSD,* lse *is convex.*

Proof. Differentiating once gives $\partial\,\mathrm{lse}/\partial x_i = e^{x_i}/\sum_k e^{x_k} = s_i$, so the gradient of lse is simply the softmax vector $\mathbf{s}$. Differentiating $s_i = e^{x_i}/\sum_k e^{x_k}$ a second time by the quotient rule yields

$$\frac{\partial s_i}{\partial x_j} = \frac{\delta_{ij}\, e^{x_i}}{\sum_k e^{x_k}} - \frac{e^{x_i}\, e^{x_j}}{\left(\sum_k e^{x_k}\right)^2} = s_i\,(\delta_{ij} - s_j),$$

which is the (i, j) entry of $\mathrm{diag}(\mathbf{s}) - \mathbf{s}\mathbf{s}^\top$ (the same softmax Jacobian derived in Section 34.4). Now let Y be a random class index drawn with probabilities $P(Y = i) = s_i$, and let $\mathbf{e}_Y$ be its one-hot vector. Then $\mathbb{E}[\mathbf{e}_Y] = \mathbf{s}$ and $\mathbb{E}[\mathbf{e}_Y \mathbf{e}_Y^\top] = \mathrm{diag}(\mathbf{s})$, so the Hessian is literally $\mathrm{Cov}(\mathbf{e}_Y) = \mathrm{diag}(\mathbf{s}) - \mathbf{s}\mathbf{s}^\top$. Every covariance matrix is PSD; explicitly, for any direction $\mathbf{v}$,

$$\mathbf{v}^\top \nabla^2 \mathrm{lse}(\mathbf{x})\,\mathbf{v} = \sum_i s_i v_i^2 - \left(\sum_i s_i v_i\right)^2 = \mathbb{E}\left[v_Y^2\right] - \left(\mathbb{E}[v_Y]\right)^2 = \mathrm{Var}(v_Y) \;\geq\; 0. \;\blacksquare$$

In other words, the curvature of log-sum-exp along a direction $\mathbf{v}$ is nothing more than the variance of the coordinate v_Y under the softmax distribution.

Three practical consequences follow immediately from this formula. First, the cross-entropy loss as a function of the logits, $\ell(\mathbf{z}) = \mathrm{lse}(\mathbf{z}) - z_y$ for true class y, is log-sum-exp minus a linear term, hence convex. Second, softmax regression (Section 3.1) feeds affine logits $\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}$ into $\ell(\mathbf{z})$, so by rule 2 the loss is convex in the weights: training a linear classifier on fixed features is always a convex problem. Third, the variance formula immediately predicts a *zero* eigenvalue along the all-ones vector $\mathbf{1}$, because $\mathrm{Var}(v_Y) = 0$ whenever all entries v_i are equal. That flat direction reflects the shift invariance $\mathrm{lse}(\mathbf{x} + c\mathbf{1}) = \mathrm{lse}(\mathbf{x}) + c$ that makes the numerically stable softmax possible. The very same Jacobian $\mathrm{diag}(\mathbf{s}) - \mathbf{s}\mathbf{s}^\top$ also governs the backward pass of every attention head, whose attention weights are a softmax of query–key scores (Section 10.2).

How large can the curvature of log-sum-exp get? As Exercise 6 proves, no eigenvalue of its Hessian ever exceeds $\frac{1}{2}$, regardless of the logits, so $\frac{1}{2}$ is a global smoothness constant. Where is that ceiling approached, and how curved is the loss at typical logits? Below, we draw $10{,}000$ random six-class logit vectors, rescale them by a temperature multiplier c, and report both the largest Hessian eigenvalue across all draws and the median.

```
z = rng.normal(size=(10_000, 6))              # 10,000 draws of six logits
print('  c   largest eigenvalue   median')
for c in (0.1, 1.0, 3.0, 10.0, 30.0):
    s = np.exp(c * z - (c * z).max(axis=1, keepdims=True))
    s /= s.sum(axis=1, keepdims=True)         # softmax of each row
    hess = s[:, :, None] * np.eye(6) - s[:, :, None] * s[:, None, :]
    top = np.linalg.eigvalsh(hess)[:, -1]     # largest eigenvalue per draw
    print(f'{c:5.1f}  {top.max():14.4f}  {np.median(top):12.4f}')
```

```
  c   largest eigenvalue   median
  0.1          0.2193        0.1843
  1.0          0.4546        0.3071
  3.0          0.4983        0.2995
 10.0          0.5000        0.0165
 30.0          0.5000        0.0000
```

As predicted, the largest eigenvalue never exceeds $\frac{1}{2}$, confirming that softmax cross-entropy is $\frac{1}{2}$-smooth in the logits at every temperature. When the logits are small ($c = 0.1$), the softmax is nearly uniform across the six classes and the curvature hovers just above $\frac{1}{6}$. At moderate scales, the typical curvature peaks with a median near 0.3. Push c larger, however, and a split emerges: a typical draw becomes sharply confident (nearly one-hot), so its median curvature collapses toward zero—the loss is almost completely flat around a confident prediction. Meanwhile, the *maximum* over the $10{,}000$ draws climbs toward $\frac{1}{2}$, coming from the rare draws where the top two logits nearly tie and the softmax splits its probability evenly between two classes.

35.3.5 Beyond Convexity

Deep neural networks do not have convex loss landscapes. In this final subsection we show why with just two scalar weights, identify the weaker condition that preserves gradient descent's linear convergence rate without convexity, and ask how an optimizer chooses a single solution when infinitely many global minimizers exist.

35.3.5.1 Nonconvex Networks

Consider $f(a, b) = (ab-1)^2$: a linear model with two scalar weights stacked in sequence under a squared loss—the simplest possible "deep" network. As Figure 35.14 shows, its global minimizers form the hyperbola $\{ab = 1\}$. We proved in Section 35.3.3 that the minimizer set of any convex function must be convex, yet a hyperbola is manifestly not convex: the midpoint of the two minimizers $(1, 1)$ and $(-1, -1)$ is the origin $(0, 0)$, where $f(0, 0) = 1$ sits well above the minimum value 0. The origin is a stationary point, not a minimum—its Hessian $\nabla^2 f(0, 0)$ has eigenvalues ± 2, making it a saddle point.

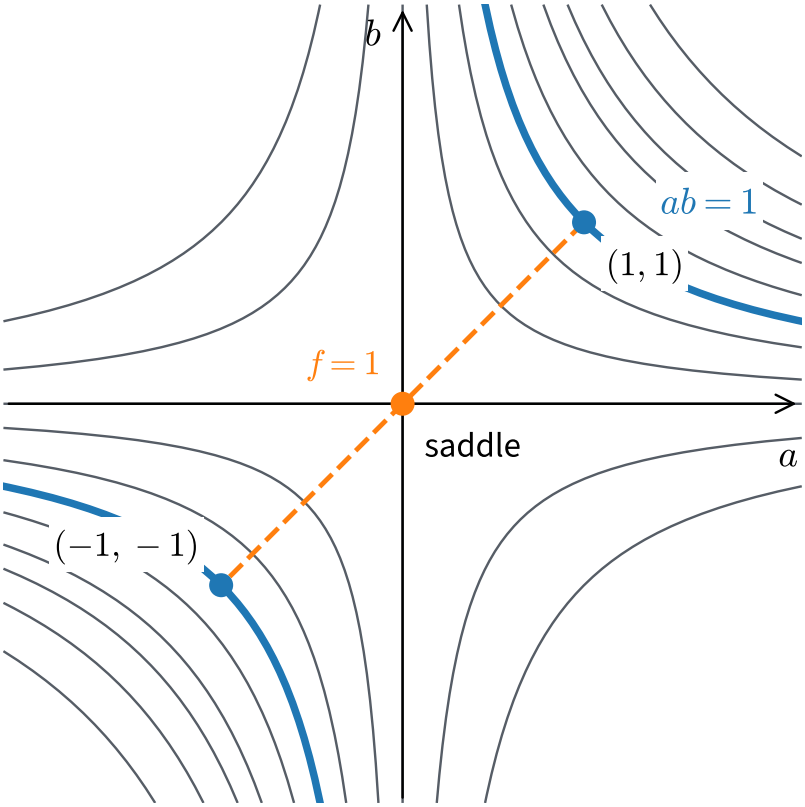


Figure 35.14: The two-weight model $(ab - 1)^2$. Its minimizers form the hyperbola $ab = 1$, which is not convex: the midpoint of $(1, 1)$ and $(-1, -1)$ is the origin, where $f = 1$ and the Hessian has eigenvalues ± 2.

Real neural networks lose convexity for a closely related structural reason. Permuting the hidden units of any layer—together with their incoming and outgoing weights—leaves the network's input–output function unchanged, so every minimizer automatically comes with a huge number of symmetric copies scattered across parameter space. Generically these copies are distinct, and averaging two of them alters the function and raises the loss, just as averaging $(1, 1)$ and $(-1, -1)$ landed on the saddle at the origin. A minimizer set that contains distinct points without containing their midpoints cannot belong to a convex objective.

Even a tiny network with two ReLU units already exhibits this failure. The parameter vector

$(w_1, v_1, w_2, v_2) = (1, 1, -1, 1)$ in $v_1 \operatorname{relu}(w_1 x) + v_2 \operatorname{relu}(w_2 x)$ computes $|x|$, and swapping the two units computes $|x|$ identically. Yet averaging the two parameter vectors cancels the input weights to zero and collapses the network to the zero function.

35.3.5.2 The Polyak–Łojasiewicz Condition

Remember how our proof of the linear convergence rate under strong convexity worked? It used convexity *only* in the first step, to derive Equation 35.33; the second step never touched convexity at all. That intermediate inequality has a name of its own. A function f with infimum $f^\star$ satisfies the **Polyak–Łojasiewicz (PL) inequality** with constant $\mu > 0$ if $\frac{1}{2}\|\nabla f(\mathbf{x})\|^2 \geq \mu\,(f(\mathbf{x}) - f^\star)$ for all $\mathbf{x}$, a condition introduced independently by Polyak and Łojasiewicz in 1963 (Polyak 1963; Łojasiewicz 1963). Intuitively, it says that the gradient is allowed to be small only where the loss is already close to optimal.

Proposition (linear rate under PL). *Let f be L-smooth, bounded below with $f^\star = \inf f$, and satisfy the PL inequality with constant μ. Then gradient descent with $\eta = 1/L$ satisfies $f(\mathbf{x}_k) - f^\star \leq (1 - \mu/L)^k\,(f(\mathbf{x}_0) - f^\star)$.*

Proof. Repeat the second half of the strong-convexity proof verbatim: combining the descent lemma with the PL inequality yields $f(\mathbf{x}_{k+1}) - f^\star \leq (1 - \mu/L)(f(\mathbf{x}_k) - f^\star)$ (Karimi et al. 2016), and iterating k times finishes the proof. ■

What makes the PL condition so appealing is how much nonconvexity it tolerates. A PL function can have multiple minimizers, flat manifolds of minimizers, and regions of negative curvature. Two things, however, are non-negotiable: every stationary point must be a global minimizer (since $\nabla f(\mathbf{x}) = \mathbf{0}$ forces $f(\mathbf{x}) = f^\star$), *and* $f - f^\star$ must grow at least quadratically away from the minimizer set. The quartic x^4 shows why the second requirement matters: it has only one stationary point, its global minimum, yet it fails PL because $\frac{1}{2}f'(x)^2 = 8x^6$ cannot stay above μx^4 as $x \to 0$ for any $\mu > 0$. That failure is precisely why gradient descent crawled on x^4 in Section 35.3.3.3. In short, strong convexity implies PL (as we proved earlier), but the converse is false.

The classic example of a nonconvex PL function is $f(x) = x^2 + 3\sin^2 x$, whose second derivative $2 + 6\cos 2x$ dips all the way to -4 and peaks at $L = 8$. Let us estimate its PL constant numerically on $[-5, 5]$ and run gradient descent from $x_0 = 4.5$ using step size $\eta = \frac{1}{16} = 1/(2L)$. (We choose $1/(2L)$ rather than $1/L$ because at $\eta = 1/L$ the local contraction factor $1 - f''(0)\,\eta$ vanishes at the origin, hiding the asymptotic rate.) At $\eta = 1/(2L)$, the quadratic ceiling guarantees a per-step decrease of $\frac{3}{8L}f'(x_k)^2$, which the PL inequality turns into the contraction factor $1 - \frac{3\mu}{4L}$.

```
f_pl = lambda x: x**2 + 3 * np.sin(x)**2       # nonconvex; f* = 0 at x = 0
df_pl = lambda x: 2 * x + 3 * np.sin(2 * x)
xs = np.linspace(-5, 5, 100001)
xs = xs[np.abs(xs) > 1e-3]                     # avoid 0/0 at the minimizer
mu = (0.5 * df_pl(xs)**2 / f_pl(xs)).min()
print(f'PL constant on [-5, 5]: mu = {mu:.4f}')
print(f"min f'' = {(2 + 6 * np.cos(2 * xs)).min():.2f}  (f is not convex)")
path = [4.5]
for _ in range(80):                            # gradient descent, eta = 1/16
    path.append(path[-1] - df_pl(path[-1]) / 16.0)
path = np.array(path)
```

```
    ratios = f_pl(path[1:]) / f_pl(path[:-1])      # per-step contraction of f - f*
    j = ratios.argmax()
    print(f'guaranteed {1 - 3 * mu / 32:.4f}; largest ratio {ratios[j]:.4f}'
          f' from x = {path[j]:.2f}; last ratio {ratios[-1]:.4f}')
```

```
PL constant on [-5, 5]: mu = 0.1755
min f'' = -4.00  (f is not convex)
guaranteed 0.9835; largest ratio 0.9778 from x = 2.19; last ratio 0.2500
```

The PL constant on $[-5, 5]$ is $\mu \approx 0.1755$, and the minimum of f'' is -4: the function is nonconvex yet PL. The window suffices, because outside it the ratio $\frac{1}{2}f'(x)^2/f(x)$ stays well above this value and tends to 2 as $|x| \to \infty$; Exercise 4 proves the bound.

With this μ the guaranteed factor is about 0.98 per step. The largest measured ratio, about 0.978, comes from $x \approx 2.2$, where $\frac{1}{2}f'(x)^2/f(x)$ attains its minimum μ, so there the guarantee is nearly tight. Near the minimizer the ratio settles at 0.25. That limit comes from the local quadratic: near 0, $f \approx \frac{1}{2}f''(0)\,x^2$ with $f''(0) = 8$, each step multiplies x by $1 - 8/16 = \frac{1}{2}$, and the gap, quadratic in x, by $\frac{1}{4}$.

PL can also hold on part of the space. For $(ab-1)^2$ the gradient satisfies $\frac{1}{2}\|\nabla f\|^2 = 2(a^2 + b^2)f$, so PL holds with constant $2r^2$ on the region $\{a^2 + b^2 \geq r^2\}$ and with no constant near the saddle at the origin. Analyses of wide networks establish such local PL inequalities near interpolating solutions (C. Liu et al. 2022), which gives a rate for the training loss and says nothing about generalization.

35.3.5.3 Implicit Bias

When a loss has many global minimizers, convex or not, the optimizer decides which one it reaches: it has an **implicit bias**. For deep learning this is not a curiosity. A network with more weights than training examples can fit its data in many ways, all with the same training loss and with different predictions elsewhere, so the choice among them decides how the model generalizes. The simplest case is underdetermined least squares, with fewer examples than weights and $\mathbf{X}$ of full row rank. Every gradient $\mathbf{X}^\top(\mathbf{X}\mathbf{w} - \mathbf{y})$ lies in the row space of $\mathbf{X}$, so iterates started from $\mathbf{w}_0 = \mathbf{0}$ never leave it. The limit is therefore the one interpolant in the row space, the *minimum-norm* interpolant that the pseudoinverse computes (Exercise 4 of Section 33.5). For logistic regression on separable data the loss has no minimizer at all, yet the direction of the iterates converges to the maximum-margin separator, the solution of the hard-margin support vector machine (Soudry et al. 2018) (Exercise 8). Which minimizer is selected depends on the model, the loss, the start, and the update rule. A fixed diagonal preconditioner $\mathbf{D}$, for instance, leads to the interpolant that minimizes $\mathbf{w}^\top\mathbf{D}^{-1}\mathbf{w}$ instead of $\|\mathbf{w}\|^2$ (Exercise 7), so the per-coordinate steps of Adam (Section 9.5) need not select the solution that gradient descent selects.

Every result of this section assumes the exact gradient and weights free to range over all of $\mathbb{R}^d$. Training uses minibatch gradients that are right only on average (Section 35.4), and a weight vector confined to a ball, a simplex, or a half-space needs the projections and multipliers of Section 35.6.

35.3.6 Summary

A convex function lies below its chords. For differentiable functions this is equivalent to lying above every tangent, and for twice-differentiable functions to a PSD Hessian; at a corner a subgradient takes the place of the gradient. Because one gradient bounds the function everywhere, a stationary point is a global minimizer, and the descent lemma turns into global rates for gradient descent: $L\|\mathbf{x}_0 - \mathbf{x}^\star\|^2/(2k)$ for smooth convex f, and a factor $1 - \mu/L$ per step with strong convexity, hence about $\kappa \log(1/\epsilon)$ steps. Jensen's inequality is the chord inequality for expectations and gives $D_{\mathrm{KL}} \geq 0$. Four closure rules certify the hinge, ℓ_1, and logistic losses; log-sum-exp, and with it the softmax cross-entropy, is convex because its Hessian is a covariance. Network losses are generically nonconvex. Where a PL inequality holds, the linear rate survives without convexity, and among many minimizers the optimizer's dynamics pick one.

35.3.7 Exercises

1. **Subdifferentials of norms and maxima.** Compute each subdifferential from the definition Equation 35.28.
    1. Compute $\partial\|\mathbf{x}\|_2$ at every $\mathbf{x} \in \mathbb{R}^d$, including $\mathbf{x} = \mathbf{0}$.
    2. Let $f(\mathbf{x}) = \max_i x_i$, and let $\mathcal{A}(\mathbf{x}) = \{i : x_i = f(\mathbf{x})\}$ be its set of active coordinates. Show that $\partial f(\mathbf{x})$ is the convex hull of $\{\mathbf{e}_i : i \in \mathcal{A}(\mathbf{x})\}$. Backpropagation through max pooling passes the incoming gradient to a single maximal input, so it uses one vertex $\mathbf{e}_i$ of this hull. Suppose that two or more inputs tie. Show that a small step along $-\mathbf{e}_i$ does not decrease f. For which $\mathbf{g} \in \partial f(\mathbf{x})$ is $-\mathbf{g}$ a descent direction, and which of them makes the directional derivative of f along $-\mathbf{g}/\|\mathbf{g}\|$ most negative?
    3. Use (a) and the optimality criterion $\mathbf{0} \in \partial(\cdot)$ to minimize $\lambda\|\mathbf{w}\|_2 + \frac{1}{2}\|\mathbf{w} - \mathbf{z}\|^2$ over $\mathbf{w} \in \mathbb{R}^d$, which gives the proximal operator of $\lambda\|\cdot\|_2$. Compare the zero pattern of the solution with that of coordinatewise soft-thresholding, $w_i = \operatorname{sign}(z_i)\max(|z_i| - \lambda, 0)$. What does this penalty do when $\mathbf{w}$ collects the outgoing weights of one hidden unit?

2. **Convex combinations and Jensen's inequality.** Write $\sum_{i=1}^N \theta_i \mathbf{x}_i$ with $\theta_i \geq 0$ and $\sum_i \theta_i = 1$ for a convex combination of N points.
    1. Show by induction on N that a convex set C contains every convex combination of points of C. This is the inclusion that completes the description of the convex hull in Section 35.3.1.
    2. Apply (a) to the epigraph of a convex function f to derive the finite form Equation 35.29 of Jensen's inequality.
    3. Let f be twice differentiable with $\alpha \leq f'' \leq \beta$ on an interval that contains the values of a random variable Z with mean m and variance s^2. Show that the Jensen gap $\mathbb{E}[f(Z)] - f(m)$ lies between $\alpha s^2/2$ and $\beta s^2/2$. Check the bounds on the two-point example of the text, $Z \in \{0, 2\}$ with $f(x) = x^2$, where $\alpha = \beta = 2$. Then apply the result to $f = -\log$ to bound $\log \mathbb{E}[Z] - \mathbb{E}[\log Z]$ for Z with values in $[a, b] \subset (0, \infty)$.

3. **Existence of minimizers.** Let $\ell(\mathbf{w}) = \sum_i \log(1 + e^{-y_i \mathbf{w}^\top \mathbf{x}_i})$ be the logistic loss of labels $y_i \in \{\pm 1\}$, and let $\mathbf{X}$ be the matrix with rows $\mathbf{x}_i^\top$. The uniqueness proposition of Section 35.3.3 states that a strictly convex function has at most one minimizer, and

that a continuous, strongly convex function on a nonempty closed convex set with nonempty interior has exactly one.

1. Suppose that some $\mathbf{w}$ satisfies $y_i \mathbf{w}^\top \mathbf{x}_i > 0$ for every i, that is, the data are linearly separable by a hyperplane through the origin. Show that $\inf_{\mathbf{w}} \ell(\mathbf{w}) = 0$ and that ℓ has no minimizer.
2. Show that ℓ is strictly convex whenever $\mathbf{X}$ has full column rank. Why does this not contradict (a)?
3. Show that $\ell(\mathbf{w}) + \frac{\lambda}{2}\|\mathbf{w}\|^2$ has exactly one minimizer for every $\lambda > 0$ and every dataset. Which hypothesis of the uniqueness proposition does the penalty supply?

4. **The PL condition beyond convexity.** The cell that studies $f(x) = x^2 + 3\sin^2 x$ measures its PL constant $\mu \approx 0.1755$ on the window $[-5, 5]$.

   1. Bound $|f'(x)|$ from below and $f(x)$ from above for $|x| \geq 5$. Conclude that the PL inequality Equation 35.33 holds on all of $\mathbb{R}$ with the constant measured on the window.
   2. Let $\mathbf{X} \in \mathbb{R}^{n \times d}$ with $n < d$ have full row rank, and let $f(\mathbf{w}) = \frac{1}{2}\|\mathbf{X}\mathbf{w} - \mathbf{y}\|^2$. Show that f is not strongly convex and has infinitely many minimizers, but satisfies the PL inequality with $\mu = \lambda_{\min}(\mathbf{X}\mathbf{X}^\top)$, the smallest eigenvalue of $\mathbf{X}\mathbf{X}^\top$. Which linear rate does gradient descent with $\eta = 1/L$ then achieve, and how is L related to $\mathbf{X}\mathbf{X}^\top$?

5. code **Contraction of the distance.** ● The rate Equation 35.32 guarantees the per-step factor $1 - \mu/L$ for $f - f^\star$, but the rate-check cell measures $(1 - \mu/L)^2$. Let f be twice continuously differentiable with $\mu \mathbf{I} \preceq \nabla^2 f \preceq L\mathbf{I}$, and let $\mathbf{x}_{k+1} = \mathbf{x}_k - \eta \nabla f(\mathbf{x}_k)$. Two facts from Section 35.1 are used: the quadratic ceiling $f(\mathbf{y}) \leq f(\mathbf{x}) + \nabla f(\mathbf{x})^\top (\mathbf{y} - \mathbf{x}) + \frac{L}{2}\|\mathbf{y} - \mathbf{x}\|^2$ and the step $\eta^\star = 2/(\mu + L)$, the best fixed step on a quadratic.

   1. Write $\nabla f(\mathbf{x}_k) = \bar{\mathbf{H}}_k (\mathbf{x}_k - \mathbf{x}^\star)$ with $\bar{\mathbf{H}}_k = \int_0^1 \nabla^2 f(\mathbf{x}^\star + s(\mathbf{x}_k - \mathbf{x}^\star))\, ds$. Show that

   $$\|\mathbf{x}_{k+1} - \mathbf{x}^\star\| \leq \max(|1 - \eta\mu|,\ |1 - \eta L|)\, \|\mathbf{x}_k - \mathbf{x}^\star\|. \tag{35.35}$$

   2. Set $\eta = 1/L$ and $\kappa = L/\mu$. Combine (a) with the lower bound Equation 35.27 and the quadratic ceiling, both taken at $\mathbf{x}^\star$, to show that $f(\mathbf{x}_k) - f^\star \leq \kappa\,(1 - \mu/L)^{2k}\,(f(\mathbf{x}_0) - f^\star)$. From which k on is this bound stronger than Equation 35.32?
   3. Predict the per-step contraction of $f - f^\star$ on the $\kappa = 10$ design of the rate-check cell at $\eta^\star = 2/11$. (Hint: both eigen-components are multiplied by numbers of absolute value $9/11$.) Verify the prediction by changing the step in that cell, and explain why, unlike at $\eta = 1/L$, the first ratio is no longer special.

6. **Smoothing the maximum.** ● A maximum of affine functions, the shape of hinge-type and worst-case objectives, has a kink wherever two of its pieces tie. For a temperature $\tau > 0$, let $\mathrm{lse}_\tau(\mathbf{z}) = \tau \,\mathrm{lse}(\mathbf{z}/\tau)$ for $\mathbf{z} \in \mathbb{R}^m$. This smoothing technique is due to Nesterov (2005) .

   1. Show that $\max_j z_j \leq \mathrm{lse}(\mathbf{z}) \leq \max_j z_j + \log m$, and conclude that $\max_j z_j \leq \mathrm{lse}_\tau(\mathbf{z}) \leq \max_j z_j + \tau \log m$.

2. Use Equation 35.34 to show that the largest eigenvalue of $\nabla^2 \mathrm{lse}_\tau(\mathbf{z})$ is at most $1/(2\tau)$, by bounding $\mathrm{Var}(v_Y)$ by $\mathbb{E}[(v_Y - c)^2]$ for a well-chosen constant c. Is the bound attained?

3. Let $F(\mathbf{w}) = \max_j (\mathbf{A}\mathbf{w} + \mathbf{r})_j$ and $F_\tau(\mathbf{w}) = \mathrm{lse}_\tau(\mathbf{A}\mathbf{w} + \mathbf{r})$, and assume that F_τ has a minimizer within distance R of $\mathbf{w}_0$. Show with (b) that F_τ is L-smooth for $L = \|\mathbf{A}\|_2^2/(2\tau)$, and that k steps of gradient descent on F_τ with $\eta = 1/L$ give

$$F(\mathbf{w}_k) - \min F \le \tau \log m + \frac{\|\mathbf{A}\|_2^2 R^2}{4\tau k}. \tag{35.36}$$

Choose τ to minimize this bound, and give the number of steps that guarantees accuracy ϵ. How does the count change when an accelerated method with error $O(LR^2/k^2)$ replaces gradient descent?

7. code **Implicit bias and the optimizer.** ● Exercise 4 of Section 33.5 shows on paper that gradient descent on $\frac{1}{2}\|\mathbf{X}\mathbf{w} - \mathbf{y}\|^2$, for $\mathbf{X} \in \mathbb{R}^{n \times d}$ with $n < d$ and full row rank, started at $\mathbf{w}_0 = \mathbf{0}$, stays in the row space of $\mathbf{X}$ and converges to the minimum-norm interpolant $\mathbf{X}^+\mathbf{y}$. Here the start stays at $\mathbf{w}_0 = \mathbf{0}$ and the algorithm changes. Take a random 4×10 system, and compare the limit of each method and its norm with `np.linalg.pinv(X) @ y`.

   1. Run gradient descent with $\eta = 1/\lambda_{\max}(\mathbf{X}^\top\mathbf{X})$, and track the norm of the component of $\mathbf{w}_k$ orthogonal to the row space, computed with the projector $\mathbf{I} - \mathbf{X}^+\mathbf{X}$.
   2. Heavy-ball momentum updates $\mathbf{w}_{k+1} = \mathbf{w}_k - \eta\nabla f(\mathbf{w}_k) + \beta(\mathbf{w}_k - \mathbf{w}_{k-1})$. Show that its iterates stay in the row space, and predict their limit. Verify the prediction with the step of (a) and $\beta = 0.9$.
   3. Replace the update by the diagonally preconditioned step $\mathbf{w}_{k+1} = \mathbf{w}_k - \eta\,\mathbf{D}\mathbf{X}^\top(\mathbf{X}\mathbf{w}_k - \mathbf{y})$, where $\mathbf{D}$ is a fixed diagonal matrix with entries drawn uniformly from $[0.1, 10]$ and $\eta = 1/\lambda_{\max}(\mathbf{D}^{1/2}\mathbf{X}^\top\mathbf{X}\mathbf{D}^{1/2})$. Use the substitution $\mathbf{u} = \mathbf{D}^{-1/2}\mathbf{w}$ to find the limit in closed form, and verify it numerically. Is the limit still the minimum-norm interpolant? What does this suggest about the implicit bias of adaptive methods such as Adam (Section 9.5), whose per-coordinate step sizes act like $\mathbf{D}$?
   4. Run exact cyclic coordinate descent for 1000 sweeps, with $\mathbf{A} = \mathbf{X}^\top\mathbf{X}$, $\mathbf{b} = \mathbf{X}^\top\mathbf{y}$, and the update $w_j \leftarrow (b_j - \sum_{i \ne j} A_{ji} w_i)/A_{jj}$ for $j = 1, \dots, d$ in turn. Check that the limit interpolates the data. Is it the minimum-norm interpolant, and which step of the row-space argument fails? Repeat the run three times with coordinates drawn uniformly at random. Does the limit depend on the draw?

8. [extended] **Max-margin bias of gradient descent.** ●● On data that a hyperplane through the origin separates, the logistic loss has no minimizer (part (a) of Exercise 3). Soudry et al. (2018) prove that gradient descent on it nevertheless diverges in a definite direction: the iterates satisfy $\mathbf{w}_k = \hat{\mathbf{w}} \log k + \rho_k$, with ρ_k bounded for almost every dataset, where

$$\hat{\mathbf{w}} = \operatorname*{argmin}_{\mathbf{w}} \|\mathbf{w}\| \quad \text{subject to} \quad y_i \mathbf{w}^\top \mathbf{x}_i \ge 1 \text{ for all } i \tag{35.37}$$

is the hard-margin SVM solution, with margin $\gamma = 1/\|\hat{\mathbf{w}}\|$. The vector $\hat{\mathbf{w}}$ is characterized by $\hat{\mathbf{w}} = \sum_i \alpha_i y_i \mathbf{x}_i$ with $\alpha_i \ge 0$, where $\alpha_i = 0$ unless $y_i \hat{\mathbf{w}}^\top \mathbf{x}_i = 1$; the points

with equality are the support vectors.

1. Derive the direction heuristically. Replace the logistic loss by its tail e^{-q} in the margin $q = y_i \mathbf{w}^\top \mathbf{x}_i$, and suppose that $\mathbf{w}_k = g(k)\,\mathbf{u} + O(1)$ with $\|\mathbf{u}\| = 1$ and $g(k) \to \infty$. Show that the negative gradient is asymptotically a nonnegative combination of the points $y_i \mathbf{x}_i$ of smallest margin $y_i \mathbf{u}^\top \mathbf{x}_i$. Conclude that $\mathbf{u}$ is proportional to $\hat{\mathbf{w}}$.
2. Derive the growth rate. Keep only the support vectors in the gradient flow $\dot{\mathbf{w}} = -\nabla \ell(\mathbf{w})$, show that $\dot{g} \approx c\, e^{-\gamma g}$ for a constant $c > 0$, and conclude that $\|\mathbf{w}(t)\| \approx \|\hat{\mathbf{w}}\| \log t$.
3. Take the support vectors $\mathbf{x}_1 = (0.5, 1.5)$ and $\mathbf{x}_2 = (1.5, 0.5)$ with label $+1$, their negatives with label -1, and 12 random points, 6 per class, with $y_i \hat{\mathbf{w}}^\top \mathbf{x}_i > 1$ for $\hat{\mathbf{w}} = (\tfrac{1}{2}, \tfrac{1}{2})$. Let $\mathbf{X}$ be the 16×2 matrix with rows $\mathbf{x}_i^\top$. Run gradient descent on the summed logistic loss from $\mathbf{w}_0 = \mathbf{0}$ with $\eta = 1/\|\mathbf{X}\|_2^2$ for 10^6 steps, which takes seconds in vectorized NumPy. Plot, against k on a logarithmic axis, the norm $\|\mathbf{w}_k\|$, the training loss, the angle between $\mathbf{w}_k$ and $\hat{\mathbf{w}}$, and the margin gap $\gamma - \min_i y_i \mathbf{w}_k^\top \mathbf{x}_i / \|\mathbf{w}_k\|$. Compare the slope of $\|\mathbf{w}_k\|$ against $\log k$ with (b).
4. The training error reaches zero within the first few steps. By what factor do the margin gap and the training loss change between $k = 10^3$ and $k = 10^6$? What does the comparison imply for stopping training as soon as the training error reaches zero?

9. **Convexity of the log-partition function.** This exercise generalizes the log-sum-exp proposition to the log-partition functions that Section 36.4 builds on. For a finite set $\mathcal{X}$, weights $h(x) > 0$, statistics $\mathbf{T}(x) \in \mathbb{R}^p$, and natural parameters $\eta \in \mathbb{R}^p$ (not a step size here), let $A(\eta) = \log \sum_{x \in \mathcal{X}} h(x) \exp(\eta^\top \mathbf{T}(x))$ and $p_\eta(x) = h(x) \exp(\eta^\top \mathbf{T}(x) - A(\eta))$.

   1. Show that $\nabla A(\eta) = \mathbb{E}_\eta[\mathbf{T}]$ and $\nabla^2 A(\eta) = \mathrm{Cov}_\eta[\mathbf{T}]$, with expectation and covariance under p_η.
   2. Conclude that A is convex, and that it is strictly convex if and only if no nonzero $\mathbf{v}$ makes $\mathbf{v}^\top \mathbf{T}(x)$ constant on $\mathcal{X}$.
   3. Recover Equation 35.34 with $\mathcal{X} = \{1, \dots, d\}$, $\mathbf{T}(x) = \mathbf{e}_x$, and $h \equiv 1$, and name the direction in which A is flat.
   4. For observations $x_1, \dots, x_n$ with average statistic $\bar{\mathbf{T}} = \frac{1}{n} \sum_i \mathbf{T}(x_i)$, show that the average negative log-likelihood $A(\eta) - \eta^\top \bar{\mathbf{T}} - \frac{1}{n} \sum_i \log h(x_i)$ is convex in η, and that a minimizer satisfies $\mathbb{E}_\eta[\mathbf{T}] = \bar{\mathbf{T}}$ (moment matching). For the softmax case of (c), why does no minimizer exist when one class is never observed?

35.4 Stochastic Gradient Methods

Training a neural network almost never uses the exact gradient of its loss. Instead of sweeping through the entire dataset for every update, each step computes the gradient on a random minibatch of a few hundred examples. Every section so far assumed exact gradients, where curvature determines the number of steps and convexity turns any stationary point into a global minimizer. A minibatch gradient points in the right direction on average, but it is noisy, and that noise changes the central question of optimization from how many steps a run needs to how close to the minimizer it can get. This section asks what gradient noise costs and how we can overcome it.

Even the simplest possible loss reveals the cost. Consider $f(x) = \frac{1}{2}x^2$, whose minimizer is 0, and suppose we add independent noise of unit variance to every gradient while taking constant steps of size $\eta = 0.1$. With exact gradients, each step would shrink x by a tenth, driving the iterate as close to 0 as we like. With noisy gradients, the iterate never settles: its mean square converges to $0.1/1.9 \approx 0.053$, so it wanders roughly 0.23 away from the minimizer no matter how many steps we take. Three quantities determine how low the loss can get: the size of the noise, the step size, and the number of steps we can afford.

We begin by quantifying the noise of a minibatch gradient. On a quadratic objective, we then compute where a fixed step size stalls and how many steps it takes to get there. Next, we show how decaying step sizes eliminate that stall and what constant a $1/k$ schedule requires, before bounding the error that a constant step tuned to a fixed budget achieves on both convex and nonconvex losses. Finally, we show how to cancel the noise on a finite training set using a stored snapshot gradient. Three takeaways are worth keeping in mind throughout: the noise floor is proportional to the step size, a $1/k$ schedule needs a large enough constant to achieve its rate, and a fixed budget of K steps calls for a step size proportional to $1/\sqrt{K}$. The final subsection on variance reduction can be skipped on a first reading.

35.4.1 Minibatch Gradients

35.4.1.1 The Cost of a Full Gradient

A typical training objective averages the losses of n examples,

$$f(\mathbf{x}) = \frac{1}{n}\sum_{i=1}^{n} f_i(\mathbf{x}),$$

where f_i is the loss on example i and $\mathbf{x} \in \mathbb{R}^d$ holds the model parameters. Because its gradient $\nabla f = \frac{1}{n}\sum_i \nabla f_i$ is also an average, evaluating it touches all n examples. When $n = 10^9$, a single step of gradient descent requires a full pass over the dataset, and gradient descent typically needs thousands of steps. Even on this chapter's running example, $\frac{1}{2}(x_1^2 + 1000\,x_2^2)$, whose curvatures differ by a factor of 1000, the optimal fixed step requires about $7{,}000$ iterations to shrink the error a million-fold (Section 35.1). **Stochastic gradient descent** (SGD) avoids this cost by replacing the full average with the average over a random **minibatch** $\mathcal{B}$ of b indices,

$$\hat{\mathbf{g}}_b(\mathbf{x}) = \frac{1}{b}\sum_{i\in\mathcal{B}} \nabla f_i(\mathbf{x}), \qquad \mathbf{x}_{k+1} = \mathbf{x}_k - \eta_k\,\hat{\mathbf{g}}_b(\mathbf{x}_k). \tag{35.38}$$

Each step now inspects only b examples instead of n: a minibatch of $b = 256$ out of 10^9 examples is roughly four million times cheaper than a full pass. Throughout this section, k indexes optimizer steps, and the step size η_k may vary with k. For our analysis, we draw a fresh minibatch at every step whose b indices are independent and uniform on $\{1, \dots, n\}$ (sampling with replacement). Real training loops usually shuffle the dataset and sample without replacement, which multiplies the gradient variance derived below by $(n-b)/(n-1)$—a factor close to one whenever $b \ll n$ (Section 9.2).

35.4.1.2 Unbiasedness and the $1/b$ Law

How accurate is a minibatch gradient? Two quantities capture its quality: its mean, which tells us whether it points in the right direction on average, and its mean squared distance from the true gradient, which we call its *noise power*. That noise power depends on how widely individual example gradients scatter around their mean, as measured by their covariance matrix.

Proposition (minibatch gradients are unbiased with noise power $\operatorname{tr}\Sigma/b$). *Fix* $\mathbf{x}$ *and let*

$$\Sigma = \frac{1}{n}\sum_{i=1}^{n} \big(\nabla f_i(\mathbf{x}) - \nabla f(\mathbf{x})\big)\big(\nabla f_i(\mathbf{x}) - \nabla f(\mathbf{x})\big)^{\top}$$

be the covariance of a single uniformly drawn example gradient. If the b indices of $\mathcal{B}$ are independent and uniform on $\{1, \dots, n\}$, then

$$\mathbb{E}\big[\hat{\mathbf{g}}_b(\mathbf{x})\big] = \nabla f(\mathbf{x}), \qquad \mathbb{E}\big[\|\hat{\mathbf{g}}_b(\mathbf{x}) - \nabla f(\mathbf{x})\|^2\big] = \frac{\operatorname{tr}\Sigma}{b}. \tag{35.39}$$

Proof. For a single uniform index i, $\mathbb{E}[\nabla f_i(\mathbf{x})] = \frac{1}{n}\sum_i \nabla f_i(\mathbf{x}) = \nabla f(\mathbf{x})$. Every term in the minibatch therefore has mean $\nabla f(\mathbf{x})$, and so does their average. Because the b indices are independent and each term has covariance Σ, the covariance of their sum is $b\Sigma$, and dividing by b scales the covariance by $1/b^2$ to give Σ/b. Finally, the expected squared norm of any mean-zero random vector equals the trace of its covariance matrix. ∎

During optimization, $\mathbf{x}$ is the current iterate $\mathbf{x}_k$ and the minibatch is drawn independently of past steps, so both identities hold conditionally on $\mathbf{x}_k$. Note that unbiasedness depends on uniform sampling: if a sampler draws example i with probability p_i, each gradient must be reweighted by $1/(np_i)$ to keep the estimator unbiased.

Because the noise power decays like $1/b$, its root-mean-square amplitude decays only like $1/\sqrt{b}$: cutting the gradient noise in ten requires a hundred times as many examples per step. Meanwhile, the true gradient signal has squared norm $\|\nabla f\|^2$ regardless of batch size. Signal and noise have equal power at the batch size $b_{\text{noise}} = \operatorname{tr}\Sigma/\|\nabla f\|^2$, which the main text calls the *gradient-noise scale* (Section 9.8); exercise 1 derives its curvature-weighted counterpart.

To see how large this noise is on concrete data, we set up a small logistic-regression problem that we will reuse in the step-size experiment below. It has $n = 200$ examples $\mathbf{a}_i \in \mathbb{R}^5$ with standard normal features, labels $y_i \in \{-1, +1\}$ generated from a logistic model with true weights $(1, -2, 1.5, 0, 2)$, and the regularized loss

$$f(\mathbf{x}) = \frac{1}{n}\sum_{i=1}^{n} \log\big(1 + e^{-y_i \mathbf{a}_i^{\top}\mathbf{x}}\big) + \frac{0.1}{2}\|\mathbf{x}\|^2.$$

The quadratic penalty makes f strongly convex with $\mu \geq 0.1$. The gradient of example i is $\nabla f_i(\mathbf{x}) = -y_i s_i \mathbf{a}_i + 0.1\,\mathbf{x}$, where $s_i = 1/(1 + e^{y_i \mathbf{a}_i^{\top}\mathbf{x}})$ is the probability that the model assigns to the wrong label. The cell below generates the dataset and defines the loss together with a vectorized helper returning all n per-example gradients as the rows of an $n \times d$ matrix.

```
n, d, reg = 200, 5, 0.1                          # examples, parameters, l2 weight
x_true = np.array([1.0, -2.0, 1.5, 0.0, 2.0])
rng = np.random.default_rng()
A = rng.normal(size=(n, d))                       # row i: the features a_i
y = np.where(rng.random(n) < 1 / (1 + np.exp(-A @ x_true)), 1.0, -1.0)

def loss(x, A, y, reg):
    """The regularized logistic loss f(x)."""
    return np.mean(np.log1p(np.exp(-y * (A @ x)))) + 0.5 * reg * x @ x

def example_grads(x, A, y, reg):
    """The n x d matrix whose row i is the gradient of f_i at x."""
    s = 1 / (1 + np.exp(y * (A @ x)))             # probability of the wrong label
    return -(y * s)[:, None] * A + reg * x
```

Since the proposition is exact, simulation serves mainly as a sanity check and to show the actual scale of $\operatorname{tr}\Sigma$ on this problem. Below, we evaluate $10{,}000$ random minibatches of each size at $\mathbf{x} = \mathbf{0}$ and compare the empirical error against $\operatorname{tr}\Sigma/b$.

```
G = example_grads(np.zeros(d), A, y, reg)        # per-example gradients at x = 0
g = G.mean(axis=0)                                # the full gradient
tr_sigma = ((G - g)**2).sum(axis=1).mean()
print(f'tr Sigma = {tr_sigma:.3f}')
for b in [1, 4, 16, 64]:
    idx = rng.integers(0, n, size=(10000, b))    # minibatches, with replacement
    err = ((G[idx].mean(axis=1) - g)**2).sum(axis=1).mean()
    print(f'b = {b:2d}: measured {err:.4f}, tr Sigma / b = {tr_sigma / b:.4f}, '
          f'ratio {err * b / tr_sigma:.3f}')
```

```
tr Sigma = 1.131
b =  1: measured 1.1188, tr Sigma / b = 1.1311, ratio 0.989
b =  4: measured 0.2847, tr Sigma / b = 0.2828, ratio 1.007
b = 16: measured 0.0699, tr Sigma / b = 0.0707, ratio 0.989
b = 64: measured 0.0177, tr Sigma / b = 0.0177, ratio 1.003
```

The single-example noise power $\operatorname{tr}\Sigma$ is about 1.1, and across all four batch sizes the measured error matches $\operatorname{tr}\Sigma/b$ to within a few percent of sampling fluctuation. Knowing the noise of a single step, we can now ask what that noise does over an entire optimization run.

35.4.1.3 The Noise Floor

Does gradient noise average out over many steps, or does it accumulate? Section 9.2 measured the cloud in which SGD settles inside a valley and derived its radius. Here we solve the entire trajectory in closed form, which also tells us how many steps it takes to reach that cloud. The simplest case already contains the full mechanism: a single coordinate of a quadratic, $f(x) = \frac{1}{2}\lambda x^2$ with curvature $\lambda > 0$ and minimizer 0. Model the stochastic gradient as the true gradient λx_k plus random noise ξ_k with mean zero given x_k and variance σ^2. Here σ^2 plays the role of a diagonal entry of Σ/b, so every quantity proportional to σ^2 scales as $1/b$. One SGD step reads

$$x_{k+1} = x_k - \eta\,(\lambda x_k + \xi_k) = \rho\, x_k - \eta\,\xi_k, \qquad \rho = 1 - \eta\lambda.$$

At each step, the deterministic part pulls the iterate toward the minimizer by the factor ρ, just as exact gradient descent does, while the stochastic part delivers a random kick $-\eta\xi_k$. To track the iterate's size, we follow its second moment $e_k = \mathbb{E}[x_k^2]$. Squaring the update and taking expectations, the cross term $-2\rho\eta\,\mathbb{E}[x_k\xi_k]$ vanishes because ξ_k has mean zero conditioned on x_k, leaving

$$e_{k+1} = \rho^2 e_k + \eta^2\sigma^2.$$

Every step contracts the second moment by ρ^2 and injects fresh noise power $\eta^2\sigma^2$. Nearly every result in this section can be read off this one recursion. A linear recurrence with a constant driving term is solved by finding its fixed point V, where $V = \rho^2 V + \eta^2\sigma^2$. Subtracting that fixed-point equation from the recursion shows that the remainder $e_k - V$ shrinks by ρ^2 at every step, giving

$$e_k = V + \rho^{2k}\,(e_0 - V) \tag{35.40}$$

with the **noise floor**

$$V = \frac{\eta^2\sigma^2}{1-\rho^2} = \frac{\eta\,\sigma^2}{\lambda\,(2-\eta\lambda)} \;\approx\; \frac{\eta\,\sigma^2}{2\lambda} \qquad (\eta\lambda \ll 1). \tag{35.41}$$

The floor grows linearly with the step size and the noise variance, and shrinks inversely with the curvature. Both formulas require $0 < \eta < 2/\lambda$, the same stability ceiling as exact gradient descent on a quadratic; above it, the second moment diverges even without noise.

The trajectory splits cleanly into two phases. The transient $\rho^{2k}(e_0 - V)$ decays geometrically, just as in deterministic gradient descent, whereas the floor V is purely an effect of noise. Far from the minimizer, the transient dominates and SGD behaves much like gradient descent. Close to the minimizer, the inward pull $\eta\lambda x$ weakens while the random kicks keep the same variance $\eta^2\sigma^2$, until the two balance at $e = V$. From then on the iterate never converges; instead, its distribution settles into a stationary **noise ball** around the minimizer whose root-mean-square radius, roughly $\sqrt{\eta\sigma^2/(2\lambda)}$, scales as $\sqrt{\eta}$. Figure 35.15 illustrates both phases on a two-dimensional quadratic bowl.

In practice we care about the expected loss rather than the squared distance. Because $\mathbb{E}f(x_k) = \frac{1}{2}\lambda e_k$, the loss floor is

$$\tfrac{1}{2}\lambda V = \frac{\eta\,\sigma^2}{2\,(2-\eta\lambda)} \;\approx\; \frac{\eta\,\sigma^2}{4}.$$

Surprisingly, for small steps the curvature λ cancels out completely: a flatter direction lets the iterate wander farther, by precisely the factor that its flatness discounts from the loss.

On a d-dimensional quadratic, the coordinates along the eigenvectors of the Hessian decouple into d independent copies of this scalar recursion, each with its own eigenvalue λ_i and noise variance σ_i^2. The total expected loss floor is therefore the sum $\sum_i \eta\sigma_i^2/(2(2-\eta\lambda_i))$. Because the σ_i^2 are the diagonal entries of Σ/b in the eigenbasis, they sum to $\operatorname{tr}\Sigma/b$. Whenever every $\eta\lambda_i \ll 1$, the total loss floor simplifies to approximately $\eta\operatorname{tr}\Sigma/(4b)$, which depends on the step size and batch size only through their ratio η/b. Halving the step size and

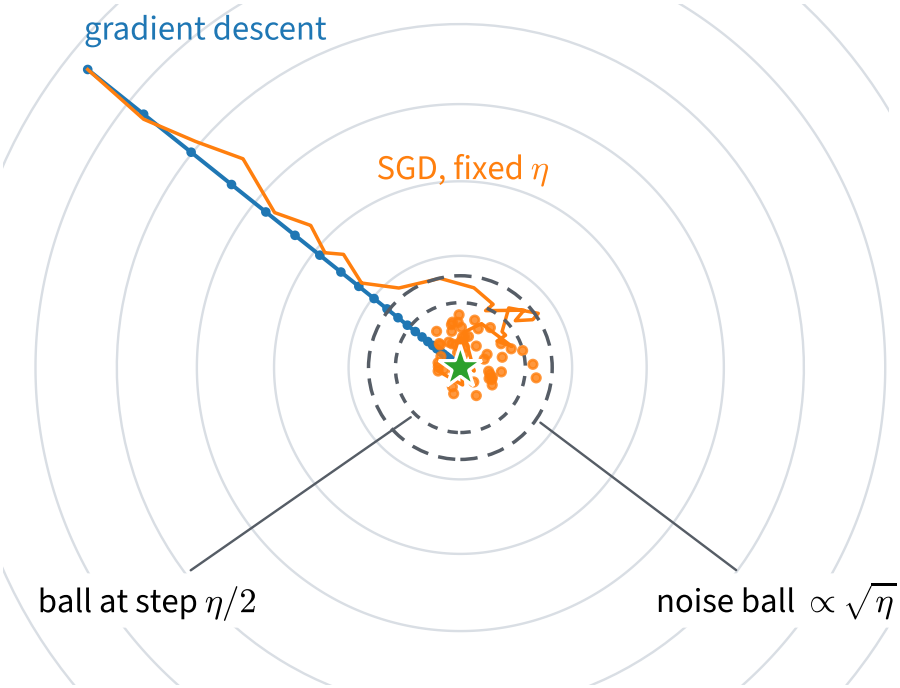


Figure 35.15: Gradient descent and fixed-step SGD on $\frac{1}{2}\|\mathbf{x}\|^2$ with $\eta = 0.15$ and gradient noise of standard deviation 0.6. The full-batch path converges to the minimizer, whereas SGD descends like gradient descent and then fluctuates inside a noise ball. Dashed circles mark the noise ball at this step size and at half the step size; their radii, drawn at three times the per-coordinate root-mean-square distance $\sqrt{\eta\sigma^2/(2\lambda)}$ to enclose the scatter, differ by a factor of $\sqrt{2}$.

doubling the batch size therefore lower the floor by the same factor—the coupling behind the large-batch learning-rate rules in Section 9.8.

How quickly does a run reach its floor? Take $\lambda = \sigma = 1$, as in the opening example, where $\eta = 0.1$ produced $V = 0.053$. Raising the step to $\eta = 0.3$ raises the floor to $V = 0.176$ (a little more than three times as high, with a loss floor of 0.088). Starting from $x_0 = 3$ ($e_0 = 9$), the transient $e_k - V$ drops below a tenth of V after 36 steps at $\eta = 0.1$ ($\rho^2 = 0.81$), compared with just 9 steps at $\eta = 0.3$ ($\rho^2 = 0.49$). Cutting the step size by a third cuts the floor by roughly a third, but demands four times as many steps to reach it. Every method in the rest of this section is a different way of navigating this trade-off between speed and accuracy.

Now return to our running example $\frac{1}{2}(x_1^2 + 1000\,x_2^2)$, modeling the noise directly by adding independent mean-zero noise of variance $\sigma^2 = 1$ to each coordinate of the gradient $(x_1, 1000\,x_2)$ at every step. The two coordinates obey the scalar recursion with $\lambda_1 = 1$ and $\lambda_2 = 1000$. Set $\eta = 1/L = 10^{-3}$, half the stability ceiling $2/1000$. The stiff coordinate has $\rho_2 = 0$: a single step erases its transient completely, leaving it on its floor $V_2 = 10^{-3}/1000 = 10^{-6}$. The slow coordinate has $\rho_1 = 0.999$ and floor $V_1 = 10^{-3}/1.999 = 5.0 \times 10^{-4}$, and its transient takes $\log 1000/(-2\log 0.999) \approx 3{,}450$ steps to shrink a thousandfold. The total loss floor is 7.5×10^{-4}, split between 2.5×10^{-4} from the slow coordinate and 5×10^{-4} from the stiff one. Notice the irony: the stiff coordinate that gradient descent eliminates in one step accounts for two-thirds of the loss floor, because at $\eta\lambda_2 = 1$ its loss floor is $\eta\sigma^2/2$, twice the small-step value $\eta\sigma^2/4$. Ill-conditioning slows down the flat direction, while noise raises the loss along both.

Our closed-form floor is exact only for a quadratic with additive noise. Next, we test whether the scaling "floor proportional to η" still holds on the non-quadratic logistic loss, and see how decaying step sizes break through the floor.

35.4.2 Step-Size Decay

35.4.2.1 The Robbins–Monro Conditions

Because the noise floor scales linearly with η, actually reaching the minimizer requires shrinking the step size toward zero. Shrink it too fast, however, and the iterate stalls short of the goal: with unit-sized gradients, the total distance traveled cannot exceed $\sum_k \eta_k$. The classical conditions of Robbins and Monro (1951) capture both requirements at once,

$$\sum_k \eta_k = \infty, \qquad \sum_k \eta_k^2 < \infty. \tag{35.42}$$

The first condition guarantees that the iterate can travel arbitrarily far from its starting point, while the second keeps the cumulative variance injected by the noise, $\sigma^2 \sum_k \eta_k^2$, finite. The schedule $\eta_k = c/k$ satisfies both conditions, whereas $\eta_k = c/\sqrt{k}$ and a constant step satisfy only the first. Together with unbiased gradients whose variance grows at most quadratically with the distance to the minimizer, these two conditions guarantee that SGD converges to the minimizer of any strongly convex loss (Robbins and Monro 1951; Bottou et al. 2018).

On our scalar quadratic, convergence in mean square actually needs less. Unrolling the recursion shows that $e_k \to 0$ whenever $\eta_k \to 0$ and $\sum_k \eta_k = \infty$ (with $\eta_k < 2/\lambda$); in particular, $\eta_k = c/\sqrt{k}$ yields $e_k \approx c\sigma^2/(2\lambda\sqrt{k})$, which simply tracks the instantaneous

floor of the current step. The stronger square-summability condition is what the general theorem requires to guarantee convergence along almost every individual sample path.

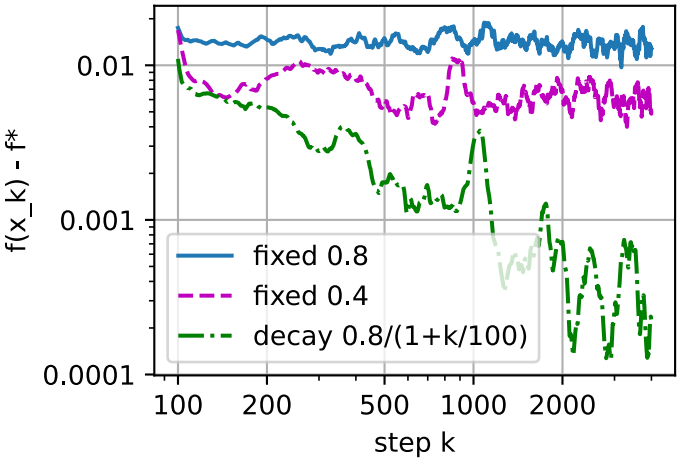
Two questions remain for a non-quadratic objective: does the noise floor still scale linearly with η, and does a decaying schedule push below it? To check on our logistic-regression problem, we first compute the reference optimum $f^\star$ with $4{,}000$ full-gradient steps, then define a minibatch SGD loop that records the suboptimality gap $f(\mathbf{x}_k) - f^\star$ at every step.

```
x_star = np.zeros(d)
for _ in range(4000):                              # full-gradient reference optimum
    x_star -= 0.5 * example_grads(x_star, A, y, reg).mean(axis=0)
f_star = loss(x_star, A, y, reg)

def sgd(schedule, A, y, reg, f_star, rng, steps=4000, b=8):
    """Minibatch SGD from x = 0; returns the gap f(x_k) - f* after every step."""
    x, gaps = np.zeros(A.shape[1]), np.empty(steps)
    for k in range(steps):
        idx = rng.integers(0, len(y), size=b)
        x -= schedule(k) * example_grads(x, A[idx], y[idx], reg).mean(axis=0)
        gaps[k] = loss(x, A, y, reg) - f_star
    return gaps
```

If the quadratic model is a reliable guide, halving a fixed step size should halve the gap at which SGD stalls. We run $4{,}000$ steps with $b = 8$ using constant steps $\eta = 0.8$ and $\eta = 0.4$, alongside the decaying schedule $\eta_k = 80/(k + 100) = 0.8/(1 + k/100)$, which begins at 0.8. The cell prints the mean gap over the final $1{,}000$ steps of each run, reports the ratios between them, and plots 100-step moving averages on log–log axes.

```
rules = {'fixed 0.8': lambda k: 0.8, 'fixed 0.4': lambda k: 0.4,
         'decay 0.8/(1+k/100)': lambda k: 0.8 / (1 + k / 100)}
gaps = {name: sgd(s, A, y, reg, f_star, rng) for name, s in rules.items()}
tail = [gap[-1000:].mean() for gap in gaps.values()]
print('mean gap, last 1000 steps: ' + ', '.join(f'{t:.1e}' for t in tail))
print(f'floor(0.8) / floor(0.4) = {tail[0] / tail[1]:.2f}, '
      f'floor(0.8) / decay = {tail[0] / tail[2]:.0f}')
window = np.ones(100) / 100                        # plot 100-step moving averages
d2l.plot(np.arange(100, 4001), [np.convolve(gap, window, 'valid') for gap in gaps.values()],
         'step k', 'f(x_k) - f*', xscale='log', yscale='log', legend=list(gaps))
```

```
mean gap, last 1000 steps: 1.4e-02, 5.9e-03, 4.1e-04
floor(0.8) / floor(0.4) = 2.35, floor(0.8) / decay = 34
```



Both fixed-step runs stall. At $\eta = 0.8$, the mean gap over the last $1{,}000$ steps settles around 1 to 2×10^{-2}, and halving the step size cuts it roughly in half: across repeated runs, the ratio fluctuates between about 1.7 and 2.8, centered on the factor of 2 predicted by the quadratic model even though the logistic loss is not quadratic. Meanwhile, the decaying schedule finishes roughly 20 to 90 times below the $\eta = 0.8$ floor, and its curve on the log–log plot is still trending downward despite minibatch jitter. Its large early steps cover the distance to the minimizer quickly, while its small late steps shrink the noise ball.

35.4.2.2 The Constant in a $1/k$ Schedule

The schedule $\eta_k = c/(k + k_0)$ satisfies both Robbins–Monro conditions for any $c > 0$, with the shift k_0 keeping the initial steps safely below the stability ceiling. Does every choice of c achieve an $O(1/k)$ rate? The scalar quadratic with curvature $\lambda = \mu$ gives an exact answer. Writing $t = k + k_0$, the second-moment recursion becomes

$$e_{k+1} = \left(1 - \frac{c\mu}{t}\right)^2 e_k + \frac{c^2\sigma^2}{t^2}.$$

Here the contraction weakens like $1/t$, while the injected noise power falls like $1/t^2$. Which effect dominates depends critically on c: if c is too small, the step size shrinks faster than the iterate can contract toward the minimizer, so the schedule itself becomes the bottleneck rather than the noise.

Proposition (the constant of a $1/k$ schedule). *On the scalar model with curvature μ and step sizes $\eta_k = c/(k + k_0)$, if $2c\mu > 1$ then $k\, e_k \to A$, where*

$$A = \frac{c^2\sigma^2}{2c\mu - 1}. \tag{35.43}$$

If $2c\mu < 1$, then e_k decays like $k^{-2c\mu}$, strictly slower than $1/k$, and at the boundary $2c\mu = 1$ it decays like $\log k/k$. For every $c > 0$, $e_k \to 0$.

We omit the full proof here, deriving the constant A by matching leading terms and sketching the other two regimes informally; exercise 2 provides a complete proof by induction above the threshold and by unrolling the recursion below it.

Derivation of the constant. Try the asymptotic form $e_k \approx A/t$. The left-hand side of the recursion is $A/(t+1) \approx A/t - A/t^2$. On the right-hand side, expanding $(1-c\mu/t)^2 \approx 1 - 2c\mu/t$ gives $A/t - 2c\mu A/t^2 + c^2\sigma^2/t^2$ up to terms of order t^{-3}. The $1/t$ terms match automatically, and equating the $1/t^2$ coefficients yields $-A = -2c\mu A + c^2\sigma^2$, or $A(2c\mu - 1) = c^2\sigma^2$. Notice the subtle term on the left: advancing from t to $t+1$ contributes $-A/t^2$, which is of the same order as the contraction term and creates the threshold $2c\mu > 1$. Ignoring that shift would falsely predict $A = c\sigma^2/(2\mu)$ for every $c > 0$.

Below the threshold (informally). When $2c\mu < 1$, the cumulative contraction sets the decay rate for both the initial error and the early noise. Because $\log(1-u) \approx -u$ and $\sum_{j \le t} 1/j \approx \log t$, the product of contractions behaves as $\prod_j (1 - c\mu/(j+k_0))^2 \approx (k_0/t)^{2c\mu}$, which decays more slowly than $1/t$. At the boundary $2c\mu = 1$, noise injected at step j survives at step t with weight roughly $c^2\sigma^2/(jt)$, and summing over j produces the extra $\log k$ factor.

To see the sensitivity in numbers, take $\mu = \sigma = 1$. Setting $c = 1$ gives $A = 1$, and iterating the recursion from $e_0 = 1$ with $k_0 = 10$ yields $k\,e_k = 1.0001$ at $k = 10^6$. Doubling to $c = 2$ raises the constant only modestly, to $A = 4/3$. Dropping below the threshold is far more damaging: $c = 0.25$ yields a log–log slope of -0.50 between $k = 10^5$ and 10^6, and $c = 0.1$ yields a slope of just -0.20. Minimizing A over c gives the optimal constant $c = 1/\mu$, where $A = \sigma^2/\mu^2$ and $e_k \approx \sigma^2/(\mu^2 k)$. Remarkably, averaging the iterates of a constant-step run achieves the exact same optimal constant without decaying the step size at all (exercise 5). The sharp threshold at $c = 1/(2\mu)$ makes $1/k$ schedules fragile in practice: the smallest curvature μ is rarely known, and underestimating c sacrifices the convergence rate itself. This fragility is one major reason practical learning-rate schedules decay more slowly than $1/k$.

On the running example $\frac{1}{2}(x_1^2 + 1000\,x_2^2)$, the shift k_0 creates a second difficulty. Stability at the very first step requires $c/k_0 < 2/L$, or $k_0 > cL/2$. Choosing the optimal constant $c = 1/\mu = 1$ for the slow coordinate therefore forces $k_0 > 500$. With $k_0 = 1000$ and $\mathbf{x}_0 = (1,1)$, the slow coordinate's transient still retains a fraction $(k_0/(k+k_0))^2 \approx 0.008$ of its initial error after $k = 10^4$ steps, leaving the expected loss at 4.2×10^{-3}—more than five times worse than the constant-step floor of 7.5×10^{-4}. The small initial steps demanded by the stiff coordinate starve the slow coordinate, and the condition number reappears inside k_0.

35.4.2.3 Schedule Shapes under the Exact Recursion

Modern training rarely uses $1/k$ decay; instead, practitioners favor cosine decay and warmup–stable–decay, which Section 9.7 compares on neural networks. Although no general theorem ranks those schedules on deep networks, our quadratic model lets us compute the expected loss of *any* schedule exactly, with no sampling and no random noise. Because our earlier derivation of the second-moment recursion holds step by step even when η_k varies, we simply have

$$e_{k+1} = (1 - \eta_k\lambda)^2 e_k + \eta_k^2\sigma^2, \qquad \mathbb{E}f(x_k) = \tfrac{1}{2}\lambda e_k.$$

Iterating this update computes the exact expected loss curve across a budget of K steps in K multiplications. We compare four shapes over the same budget K, illustrated in Figure 35.16: a constant step η_0; the $1/k$ schedule $c/(k+k_0)$ with $c = \eta_0 k_0$ so that it also starts

at η_0; **cosine decay** (Loshchilov and Hutter 2016); and **warmup–stable–decay** (WSD) (Hu et al. 2024), which holds η_0 constant for the first 80% of the budget and then decays linearly to zero:

$$\eta_k^{\cos} = \tfrac{1}{2}\eta_0\Big(1 + \cos\frac{\pi k}{K}\Big),$$
$$\eta_k^{\mathrm{WSD}} = \begin{cases} \eta_0, & k < 0.8K, \\ \eta_0\,\dfrac{K-k}{0.2K}, & k \geq 0.8K. \end{cases}$$

Both cosine and WSD start at η_0 and end at zero, but cosine decays continuously throughout the run while WSD delays all decay until the final fifth. We omit the brief initial warmup ramp (discussed in Section 35.5.4) because the quadratic model is already stable at η_0 from step zero. WSD is popular in large-scale pretraining for an operational reason: any checkpoint from its constant plateau can be decayed to any chosen horizon, so the total budget need not be committed in advance. Our quadratic model evaluates final expected loss, not that flexibility.

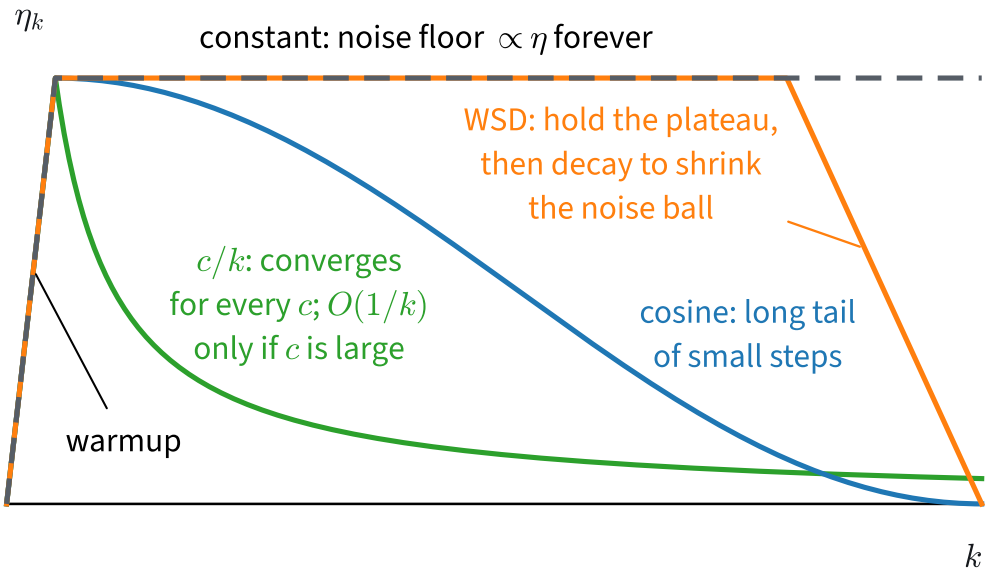
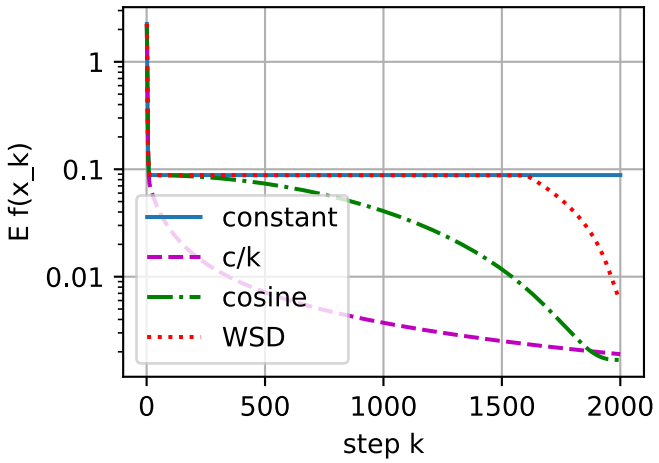


Figure 35.16: Schedule shapes over one budget K: constant, the $1/k$ decay (labelled c/k), cosine, and warmup–stable–decay (WSD), each preceded by a short linear warmup omitted in the computation below. The figure plots the step-size sequences; the expected loss each produces on the quadratic model is computed below.

Which of these four schedules reaches the lowest final loss? It is worth guessing before running the numbers. The cell below iterates the exact recursion with $\lambda = \sigma = 1$, $K = 2000$, $\eta_0 = 0.3$, $x_0 = 3$, and $k_0 = 50$ for the $1/k$ schedule, printing the expected loss at 80% of the budget and at the end before plotting $\mathbb{E}f(x_k)$.

```
lam, sigma, K, eta0, x0 = 1.0, 1.0, 2000, 0.3, 3.0

def expected_loss(schedule, lam, sigma, K, x0):
    """E f(x_k) = lam e_k / 2 for k = 1..K from the exact second-moment recursion."""
    e, Ef = x0**2, np.empty(K)
    for k in range(K):
        eta = schedule(k)
        e = (1 - eta * lam)**2 * e + eta**2 * sigma**2
        Ef[k] = 0.5 * lam * e
    return Ef

schedules = {
    'constant': lambda k: eta0,
    'c/k': lambda k: eta0 / (1 + k / 50),
    'cosine': lambda k: 0.5 * eta0 * (1 + np.cos(np.pi * k / K)),
    'WSD': lambda k: eta0 if k < 0.8 * K else eta0 * (K - k) / (0.2 * K)}
traces = {name: expected_loss(s, lam, sigma, K, x0) for name, s in schedules.items()}
print(f'floor {eta0 * sigma**2 / (2 * (2 - eta0 * lam)):.4f}   '
      + ''.join(f'{name:>10s}' for name in traces))
for label, k in [('E f at 80%', int(0.8 * K) - 1), ('E f at end', K - 1)]:
    print(f'{label:15s}' + ''.join(f'{Ef[k]:10.2e}' for Ef in traces.values()))
d2l.plot(np.arange(1, K + 1), list(traces.values()), 'step k', 'E f(x_k)',
         yscale='log', legend=list(traces))
```

```
floor 0.0882     constant       c/k    cosine       WSD
E f at 80%       8.82e-02  2.36e-03  7.87e-03  8.82e-02
E f at end       8.82e-02  1.90e-03  1.68e-03  6.13e-03
```



Because the recursion is deterministic, the numbers are identical on every run. The constant step settles within a tenth of its loss floor 0.0882 by step 9 and stays there. WSD *is* the constant run until its decay phase begins, so at 80% of the budget it sits on the exact same floor; its final 400 steps then drop the loss by a factor of 14.4, down to 6.1×10^{-3}, accounting for all of its gain. Cosine decay and $1/k$ decay finish close together at 1.7×10^{-3} and 1.9×10^{-3}, roughly three to four times lower than WSD.

The $1/k$ endpoint matches our proposition closely: here $c = 0.3 \cdot 50 = 15$, giving $A =$

$225/29 \approx 7.8$ and $\mathbb{E} f(x_K) \approx \frac{1}{2} A/(K+k_0) = 1.9\times10^{-3}$. Keep in mind that this ranking is exact for these quadratic parameters and need not transfer directly to neural networks. On the quadratic itself, holding a constant plateau does not improve the final loss: lengthening WSD's decay phase reduces the final loss monotonically, reaching 2.7×10^{-3} for a linear decay across the entire run.

35.4.3 Rates for a Fixed Budget

A decaying schedule is built for an indefinite run. In practice, however, we usually know our compute budget K ahead of time, which reframes the question: what constant step size should we choose for K steps, and what accuracy does it guarantee? Without strong convexity, there is no geometric contraction factor and no exact noise floor, so we prove upper bounds rather than exact trajectories. Both bounds below share the same two-term structure: a transient term that decreases as $1/(\eta K)$ plus a noise term that grows as η. Balancing the two selects $\eta \propto 1/\sqrt{K}$, as shown in Figure 35.17.

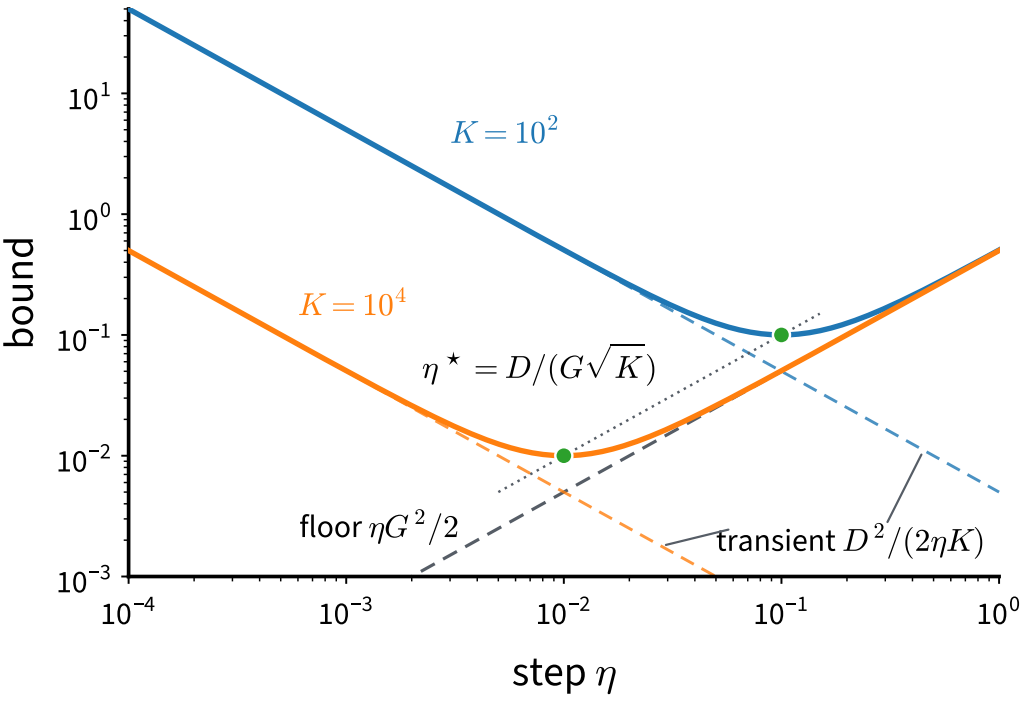


Figure 35.17: The two-term bound $D^2/(2\eta K) + \eta G^2/2$ of Equation 35.44 plotted against the step size η for $D = G = 1$ and two budgets. The transient falls with η while the floor rises with it; their sum reaches its minimum $DG/\sqrt{K}$ at $\eta^\star = D/(G\sqrt{K})$, so a $100\times$ larger budget lowers both the optimal step and the bound by $10\times$.

35.4.3.1 Averaged Iterates on Convex Losses

Our first bound requires nothing beyond convexity, applied through the subgradient inequality of Section 35.3: $f(\mathbf{y}) \geq f(\mathbf{x}) + \mathbf{s}^\top(\mathbf{y}-\mathbf{x})$ for all $\mathbf{y}$ and every subgradient $\mathbf{s} \in \partial f(\mathbf{x})$ (which reduces to $\nabla f(\mathbf{x})$ wherever f is differentiable).

Proposition (SGD on a convex loss, averaged iterate). *Let f be convex with a minimizer $\mathbf{x}^\star$ and optimal value $f^\star = f(\mathbf{x}^\star)$. Run $\mathbf{x}_{k+1} = \mathbf{x}_k - \eta\, \mathbf{g}_k$ from an initial point satisfying $\|\mathbf{x}_0 - \mathbf{x}^\star\| \le D$, where each stochastic gradient satisfies $\mathbb{E}[\mathbf{g}_k \mid \mathbf{x}_k] \in \partial f(\mathbf{x}_k)$ and $\mathbb{E}[\|\mathbf{g}_k\|^2 \mid \mathbf{x}_k] \le G^2$. Then the averaged iterate $\bar{\mathbf{x}}_K = \frac{1}{K}\sum_{k=0}^{K-1} \mathbf{x}_k$ satisfies*

$$\mathbb{E}\big[f(\bar{\mathbf{x}}_K)\big] - f^\star \;\le\; \frac{D^2}{2\eta K} + \frac{\eta\, G^2}{2}, \tag{35.44}$$

and choosing $\eta = D/(G\sqrt{K})$ gives $\mathbb{E}[f(\bar{\mathbf{x}}_K)] - f^\star \le DG/\sqrt{K}$.

Proof. We track the squared distance to the minimizer. Expanding one update gives

$$\|\mathbf{x}_{k+1} - \mathbf{x}^\star\|^2 = \|\mathbf{x}_k - \mathbf{x}^\star\|^2 - 2\eta\, \mathbf{g}_k^\top(\mathbf{x}_k - \mathbf{x}^\star) + \eta^2 \|\mathbf{g}_k\|^2.$$

Now take the conditional expectation given $\mathbf{x}_k$. Writing $\mathbf{s}_k = \mathbb{E}[\mathbf{g}_k \mid \mathbf{x}_k]$, the cross term becomes $-2\eta\, \mathbf{s}_k^\top(\mathbf{x}_k - \mathbf{x}^\star)$. By the subgradient inequality at $\mathbf{y} = \mathbf{x}^\star$, $\mathbf{s}_k^\top(\mathbf{x}_k - \mathbf{x}^\star) \ge f(\mathbf{x}_k) - f^\star$. The entire argument hinges on this step: whenever the loss gap is large, the gradient aligns with the displacement from $\mathbf{x}^\star$ and pulls the iterate closer. Bounding the final term by $\eta^2 G^2$, taking total expectations, and setting $r_k = \mathbb{E}\|\mathbf{x}_k - \mathbf{x}^\star\|^2$, we obtain

$$2\eta\, \mathbb{E}\big[f(\mathbf{x}_k) - f^\star\big] \;\le\; r_k - r_{k+1} + \eta^2 G^2.$$

Summing over $k = 0, \dots, K-1$ telescopes the distances, and since $r_0 - r_K \le r_0 \le D^2$, we have $2\eta \sum_{k<K} \mathbb{E}[f(\mathbf{x}_k) - f^\star] \le D^2 + K\eta^2 G^2$. Dividing by $2\eta K$ and applying Jensen's inequality to the convex function f, $f(\bar{\mathbf{x}}_K) \le \frac{1}{K}\sum_{k<K} f(\mathbf{x}_k)$, yields Equation 35.44. The two terms match at $\eta = D/(G\sqrt{K})$, where each equals $DG/(2\sqrt{K})$. ∎

The meaning of the two terms is transparent: $D^2/(2\eta K)$ is the transient cost of traversing distance D in K steps of size η, while $\eta G^2/2$ is the noise floor injected per step. Notably, the proof requires no smoothness at all, so the bound holds even for nonsmooth losses with corners such as the hinge loss. Multiplying the budget by 100 reduces both the optimal step size and the guaranteed error by a factor of 10. This $O(1/\sqrt{K})$ rate is unimprovable in the worst case: over the class of problems in the proposition, no algorithm that sees only unbiased stochastic gradients with $\mathbb{E}\|\mathbf{g}\|^2 \le G^2$ can guarantee better than a constant multiple of $DG/\sqrt{K}$ (Nemirovski and Yudin 1983).

Adding strong convexity changes the scaling dramatically. On our scalar quadratic, the optimal constant step for a budget K achieves $e_K = \Theta(\log K / K)$, because strong convexity shrinks the transient geometrically like ρ^{2K} rather than algebraically like $1/(\eta K)$.

Two caveats are worth noting. First, the proposition bounds the *averaged* iterate $\bar{\mathbf{x}}_K$; the final iterate obeys a weaker bound, and for the final iterate over a known budget the worst-case optimal schedule decays linearly to zero (Defazio et al. 2023). Second, the assumption $\mathbb{E}\|\mathbf{g}_k\|^2 \le G^2$ excludes unbounded-gradient losses such as a quadratic on all of $\mathbb{R}^d$, though it holds for Lipschitz losses with bounded features (such as unregularized logistic regression) or whenever the iterates remain inside a bounded set. Because it is a worst-case bound, it can also be conservative on benign problems: on our logistic-regression task restricted to a ball, it overestimates the true gap by roughly an order of magnitude.

35.4.3.2 Random Iterates on Nonconvex Losses

Neural networks confront us with nonconvex losses *and* noisy gradients at the same time, whereas our results so far have tackled one difficulty at a time. For a function f whose gradient is L-Lipschitz (an L-smooth function), the loss surface lies everywhere below a tangent parabola of curvature L:

$$f(\mathbf{y}) \;\le\; f(\mathbf{x}) + \nabla f(\mathbf{x})^{\top}(\mathbf{y}-\mathbf{x}) + \frac{L}{2}\|\mathbf{y}-\mathbf{x}\|^2 \quad \text{for all } \mathbf{x}, \mathbf{y}.$$

With exact gradients and $\eta = 1/L$, this quadratic upper bound yields $\min_{k<K} \|\nabla f(\mathbf{x}_k)\|^2 \le 2L\Delta/K$ without any convexity assumption, where $f^\star$ is a lower bound on f and $\Delta = f(\mathbf{x}_0) - f^\star$ (Section 35.1). With noisy gradients on a quadratic, a constant step stalls on a noise floor. The theorem of Ghadimi and Lan (2013) combines both ingredients. Model each SGD step as $\mathbf{x}_{k+1} = \mathbf{x}_k - \eta\,\mathbf{g}_k$, where the stochastic gradient satisfies

$$\mathbb{E}\left[\mathbf{g}_k \mid \mathbf{x}_k\right] = \nabla f(\mathbf{x}_k), \qquad \mathbb{E}\left[\|\mathbf{g}_k - \nabla f(\mathbf{x}_k)\|^2 \mid \mathbf{x}_k\right] \;\le\; \sigma^2. \tag{35.45}$$

In words, the stochastic gradient is unbiased and its noise power is uniformly bounded by σ^2. By Equation 35.39, a minibatch gradient satisfies both assumptions with $\sigma^2 = \sup_{\mathbf{x}} \operatorname{tr}\Sigma(\mathbf{x})/b$, provided that supremum is finite. It is finite for the bounded tanh model below, though it is infinite for least squares on all of $\mathbb{R}^d$, where the gradient noise grows with the distance from the minimizer.

Proposition (SGD finds approximate stationary points). *Let f be L-smooth and bounded below by $f^\star$, set $\Delta = f(\mathbf{x}_0) - f^\star$, and run SGD with stochastic gradients satisfying Equation 35.45 and a constant step size $\eta \le 1/L$. Then*

$$\frac{1}{K}\sum_{k=0}^{K-1} \mathbb{E}\left[\|\nabla f(\mathbf{x}_k)\|^2\right] \;\le\; \frac{2\Delta}{\eta K} + L\eta\sigma^2, \tag{35.46}$$

and choosing $\eta = \min\!\left(1/L,\ \sqrt{2\Delta/(L\sigma^2 K)}\right)$ ensures that for an index R drawn uniformly from $\{0, \dots, K-1\}$ independently of the run,

$$\mathbb{E}\left[\|\nabla f(\mathbf{x}_R)\|^2\right] \;\le\; \frac{2L\Delta}{K} + 2\sigma\sqrt{\frac{2L\Delta}{K}}. \tag{35.47}$$

Proof. Apply the L-smoothness upper bound to the actual step $\mathbf{y} - \mathbf{x} = -\eta\,\mathbf{g}_k$:

$$f(\mathbf{x}_{k+1}) \;\le\; f(\mathbf{x}_k) - \eta\,\nabla f(\mathbf{x}_k)^{\top}\mathbf{g}_k + \tfrac{L\eta^2}{2}\,\|\mathbf{g}_k\|^2.$$

Taking the conditional expectation given $\mathbf{x}_k$ accomplishes two things at once: unbiasedness turns the middle term into $-\eta\,\|\nabla f(\mathbf{x}_k)\|^2$, and the variance decomposition splits the quadratic term into signal plus noise, $\mathbb{E}[\|\mathbf{g}_k\|^2 \mid \mathbf{x}_k] = \|\nabla f(\mathbf{x}_k)\|^2 + \mathbb{E}[\|\mathbf{g}_k - \nabla f(\mathbf{x}_k)\|^2 \mid \mathbf{x}_k] \le \|\nabla f(\mathbf{x}_k)\|^2 + \sigma^2$. Collecting terms gives

$$\mathbb{E}\left[f(\mathbf{x}_{k+1}) \mid \mathbf{x}_k\right] \;\le\; f(\mathbf{x}_k) - \eta\left(1 - \tfrac{L\eta}{2}\right)\|\nabla f(\mathbf{x}_k)\|^2 + \tfrac{L\eta^2\sigma^2}{2}.$$

This is the standard descent lemma with one extra variance penalty $\frac{1}{2}L\eta^2\sigma^2$ per step. Whenever $\eta \le 1/L$, we have $1 - \frac{L\eta}{2} \ge \frac{1}{2}$. Taking total expectations and summing over $k = 0, \dots, K-1$, the losses telescope to $\mathbb{E}[f(\mathbf{x}_K)] - f(\mathbf{x}_0) \ge -\Delta$; dividing both sides by $\eta K/2$ yields Equation 35.46. The left-hand side is precisely $\mathbb{E}[\|\nabla f(\mathbf{x}_R)\|^2]$ for a uniformly random index R. Finally, for the balanced step size there are two cases. If $\sqrt{2\Delta/(L\sigma^2 K)} \le 1/L$, the two terms in Equation 35.46 are equal, each giving $\sigma\sqrt{2L\Delta/K}$. Otherwise $\eta = 1/L$ and $\sigma^2 < 2L\Delta/K$, so the bound is $2L\Delta/K + \sigma^2 \le 2L\Delta/K + \sigma\sqrt{2L\Delta/K}$. Both cases are bounded by Equation 35.47. ■

When $\sigma = 0$, the second term disappears and we recover the deterministic rate $2L\Delta/K$. When noise is present and the budget is large enough that $K > 2L\Delta/\sigma^2$ (so $\eta < 1/L$), the noise term dominates and decays only like $K^{-1/2}$. With $L = \Delta = \sigma = 1$ and $K = 10^4$, for instance, the balanced step is $\eta \approx 0.014$ and the bound reads $2 \times 10^{-4} + 0.028$—the noise term is 140 times larger than the deterministic term. Notice also that this bound controls the *squared* gradient norm: multiplying the budget by 100 reduces $\mathbb{E}\|\nabla f\|^2$ by a factor of 10, meaning that shrinking the gradient norm itself by a factor of 10 requires 10^4 times as many steps. If the gradient estimator is biased, an extra inner product survives the proof and adds a constant floor that never vanishes with K (exercise 4).

Why does the theorem return a *random* iterate $\mathbf{x}_R$ rather than the best one? With exact gradients, we observe $\|\nabla f(\mathbf{x}_k)\|$ at every step and can simply pick the iterate with the smallest gradient. With stochastic gradients, the true gradient norm is hidden from the algorithm; picking an index R uniformly at random is an implementable rule, and the time-average in Equation 35.46 is exactly its expected squared gradient norm.

Compared with the convex case, two things changed: we bound the squared gradient norm instead of the loss gap, and we evaluate at a random iterate instead of the average. Without convexity, a small gradient is all a first-order method can certify, since a stationary point need not be a global minimizer. Like the convex bound, this $K^{-1/2}$ rate is worst-case optimal for any method given access only to stochastic gradients satisfying Equation 35.45.

35.4.3.3 The Rate on a Single tanh Neuron

Because the proposition is an upper bound, an experiment cannot prove that the rate is tight; it can only check how real runs behave relative to the $K^{-1/2}$ ceiling. Below, we measure both the random-iterate quantity and the best iterate on a small nonconvex problem—fitting a single tanh neuron by least squares to $n = 64$ noisy examples in five dimensions:

$$f(\mathbf{x}) = \frac{1}{n}\sum_{i=1}^{n} \big(\tanh(\mathbf{a}_i^\top \mathbf{x}) - y_i\big)^2.$$

Each component is smooth and bounded, but none is convex because tanh saturates into flat plateaus for large arguments. The cell below creates the data and defines a vectorized helper computing the exact full gradient for a batch of parameter vectors, one per row of `X`.

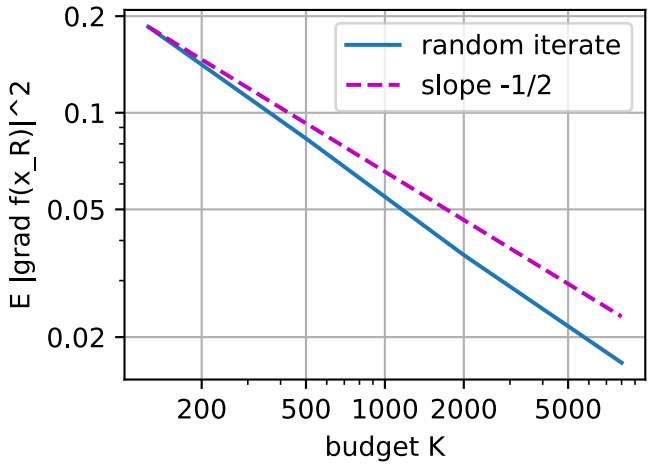
```
n_nc, d_nc = 64, 5                                  # a small nonconvex problem
A_nc = rng.normal(size=(n_nc, d_nc))
y_nc = np.tanh(A_nc @ rng.normal(size=d_nc)) + 0.5 * rng.normal(size=n_nc)
```

```
def full_grad(X, A, y):
    """Exact gradient of mean_i (tanh(a_i^T x) - y_i)^2 at each row x of X."""
    t = np.tanh(X @ A.T)                          # (runs, n)
    return 2 * ((t - y) * (1 - t**2)) @ A / len(y)
```

We run single-example SGD for budgets $K \in \{125, 500, 2000, 8000\}$ using the theorem's scaling $\eta = 0.5/\sqrt{K}$, executing 20 independent runs in parallel as the rows of `X`. For each budget, the cell records both the average of $\|\nabla f(\mathbf{x}_k)\|^2$ across the run (the random-iterate metric) and its minimum (the best iterate), prints their log–log slopes against K, and plots the random-iterate curve beside a reference slope of $-1/2$.

```
runs, Ks, avg, best = 20, np.array([125, 500, 2000, 8000]), [], []
for K in Ks:
    eta = 0.5 / np.sqrt(K)                        # the theorem's eta ~ 1/sqrt(K)
    X = np.zeros((runs, d_nc))                    # one row per independent run
    total, low = np.zeros(runs), np.full(runs, np.inf)
    for k in range(K):
        g2 = (full_grad(X, A_nc, y_nc)**2).sum(axis=1)   # |grad f(x_k)|^2
        total, low = total + g2, np.minimum(low, g2)
        i = rng.integers(0, n_nc, size=runs)             # one example per run
        t = np.tanh((X * A_nc[i]).sum(axis=1))
        X -= eta * (2 * (t - y_nc[i]) * (1 - t**2))[:, None] * A_nc[i]
    avg.append(total.mean() / K), best.append(low.mean())

def slope(v):
    return np.polyfit(np.log(Ks), np.log(v), 1)[0]

for name, v in [('random iterate', avg), ('best iterate', best)]:
    print(f'{name:15s}' + ' '.join(f'{u:.1e}' for u in v) + f'   slope {slope(v):.2f}')
d2l.plot(Ks, [avg, avg[0] * (Ks / Ks[0])**-0.5], 'budget K', 'E |grad f(x_R)|^2',
         xscale='log', yscale='log', legend=['random iterate', 'slope -1/2'])
```

```
random iterate 1.9e-01 8.3e-02 3.6e-02 1.7e-02   slope -0.58
best iterate   9.0e-03 2.1e-03 3.9e-04 4.9e-05   slope -1.25
```



Over a 64-fold increase in budget, the random-iterate metric drops by a factor of about 12 (typically between 9 and 16 across runs), corresponding to a log–log slope near -0.6. That is slightly steeper than the theorem's $-1/2$ upper bound, as expected. The best iterate falls much faster—with a slope around -1 or steeper—but as noted above, an algorithm that sees only noisy gradients has no way to identify it.

35.4.4 Variance Reduction

35.4.4.1 Control Variates

Decaying the step size eliminates the noise floor by shrinking η, but only by slowing progress to an algebraic rate. When the objective is a *finite* sum over n examples, there is a second option: eliminate the gradient noise itself. At the minimizer $\mathbf{x}^\star$, the full gradient vanishes, yet the individual example gradients $\nabla f_i(\mathbf{x}^\star)$ generally do not—and those nonzero component gradients are the noise that a constant step size keeps injecting.

One important regime needs no remedy at all. If a model interpolates the training set (fitting every single example with zero loss), then every $\nabla f_i(\mathbf{x}^\star)$ vanishes at the minimizer, and constant-step SGD on a strongly convex objective converges linearly with no noise floor (exercise 6). Large neural networks often interpolate their training sets, which helps explain why constant-step SGD performs much better on them than the additive-noise floor predicts.

When the component gradients do not vanish at $\mathbf{x}^\star$, we must cancel their variance directly. A standard Monte Carlo trick does this: from the noisy estimator $\nabla f_i(\mathbf{x})$, subtract a correlated random variable whose exact mean we know, and add that mean back. Such a correction is called a **control variate**; we meet it again in the baseline of policy gradients (Section 14.8). On a finite sum, a natural control variate is the gradient $\nabla f_i(\tilde{\mathbf{x}})$ of the same example evaluated at a saved snapshot $\tilde{\mathbf{x}}$, whose average over all i is the full gradient $\nabla f(\tilde{\mathbf{x}})$, computed once per pass over the dataset. This yields the **SVRG** (stochastic variance-reduced gradient) estimator (Johnson and Zhang 2013),

$$\tilde{\mathbf{g}}_i(\mathbf{x}) = \nabla f_i(\mathbf{x}) - \nabla f_i(\tilde{\mathbf{x}}) + \nabla f(\tilde{\mathbf{x}}). \tag{35.48}$$

In words, SVRG starts from the exact full gradient at the snapshot and adjusts it by how much a single example's gradient has changed between $\tilde{\mathbf{x}}$ and $\mathbf{x}$. For a uniformly drawn index i, the estimator remains unbiased,

$$\mathbb{E}_i\big[\tilde{\mathbf{g}}_i(\mathbf{x})\big] = \nabla f(\mathbf{x}) - \nabla f(\tilde{\mathbf{x}}) + \nabla f(\tilde{\mathbf{x}}) = \nabla f(\mathbf{x}),$$

while its variance shrinks to zero as the iterate and snapshot converge.

Proposition (SVRG variance is bounded by the distance to the snapshot). *If each component f_i has an L_i-Lipschitz gradient and $L_{\max} = \max_i L_i$, then for a uniformly drawn index i,*

$$\mathbb{E}_i\big\|\tilde{\mathbf{g}}_i(\mathbf{x}) - \nabla f(\mathbf{x})\big\|^2 \;\le\; \mathbb{E}_i\big\|\nabla f_i(\mathbf{x}) - \nabla f_i(\tilde{\mathbf{x}})\big\|^2 \;\le\; L_{\max}^2\,\|\mathbf{x} - \tilde{\mathbf{x}}\|^2. \tag{35.49}$$

Proof. The left-hand side is the variance of the random vector $\nabla f_i(\mathbf{x}) - \nabla f_i(\tilde{\mathbf{x}})$, whose mean is $\nabla f(\mathbf{x}) - \nabla f(\tilde{\mathbf{x}})$. Because the variance of any random vector is at most its second

moment, the first inequality follows immediately. Applying the L_i-Lipschitz bound to each term of the second moment gives the second inequality. ■

Because the noise power is bounded by a multiple of $\|\mathbf{x} - \tilde{\mathbf{x}}\|^2$, it vanishes automatically as both $\mathbf{x}$ and $\tilde{\mathbf{x}}$ approach $\mathbf{x}^\star$, removing the noise floor even for a constant step size. Intuitively, evaluating the same example at two nearby points produces gradients that share nearly identical noise; taking their difference cancels that shared noise while retaining the small genuine change in the gradient.

35.4.4.2 SVRG and SAGA

To keep the bound Equation 35.49 shrinking, the snapshot must periodically catch up to the iterate; yet recomputing a full gradient at every step would defeat the purpose of stochastic optimization. SVRG resolves this by running in epochs. At the beginning of each epoch, it saves the current iterate as the snapshot $\tilde{\mathbf{x}}$ and computes its full gradient $\nabla f(\tilde{\mathbf{x}})$. It then performs m inner updates $\mathbf{x} \leftarrow \mathbf{x} - \eta\, \tilde{\mathbf{g}}_i(\mathbf{x})$ with fresh random indices, and sets the final inner iterate as the next snapshot. One epoch therefore costs $n + 2m$ example-gradient evaluations. Johnson and Zhang (2013) (Theorem 1) prove that if every f_i is convex with an $L_{\max}$-Lipschitz gradient, f is μ-strongly convex, $\eta < 1/(4L_{\max})$, and m is large enough that the per-epoch contraction factor falls below one, then the expected suboptimality $\mathbb{E}[f(\tilde{\mathbf{x}})] - f^\star$ decays geometrically with the number of epochs. (The theorem selects a random inner iterate as the next snapshot; our implementation uses the last iterate, which behaves similarly in practice.)

We test SVRG on ridge regression, a strongly convex finite sum with $f_i(\mathbf{x}) = \frac{1}{2}(\mathbf{a}_i^\top \mathbf{x} - y_i)^2 + \frac{0.1}{2}\|\mathbf{x}\|^2$, $n = 300$ examples, and $d = 12$ parameters. Its exact minimizer solves the normal equations, and each component gradient is L_i-Lipschitz with $L_i = \|\mathbf{a}_i\|^2 + 0.1$. The cell below generates the data, defines the component and full gradients, and prints $L_{\max}$ along with the theoretical step-size ceiling $1/(4L_{\max})$.

```
n_r, d_r, reg_r = 300, 12, 0.1                    # ridge regression, a finite sum
A_r = rng.standard_normal((n_r, d_r))
y_r = A_r @ rng.standard_normal(d_r) + 0.5 * rng.standard_normal(n_r)
H_r = A_r.T @ A_r / n_r + reg_r * np.eye(d_r)     # Hessian of f
x_star_r = np.linalg.solve(H_r, A_r.T @ y_r / n_r)

def component_grad(x, i, A, y, reg):
    """Gradient of f_i(x) = (a_i^T x - y_i)^2 / 2 + reg |x|^2 / 2 (i: index or array)."""
    return A[i] * (A[i] @ x - y[i])[..., None] + reg * x

def full_grad_sum(x, A, y, reg):
    """Gradient of the finite sum f = mean_i f_i."""
    return A.T @ (A @ x - y) / len(y) + reg * x

def objective(x, A, y, reg):
    """The ridge objective f(x)."""
    return 0.5 * np.mean((A @ x - y)**2) + 0.5 * reg * x @ x

L_max = (A_r**2).sum(axis=1).max() + reg_r       # largest per-example smoothness
print(f'L_max = {L_max:.0f}, 1 / (4 L_max) = {1 / (4 * L_max):.4f}')
```

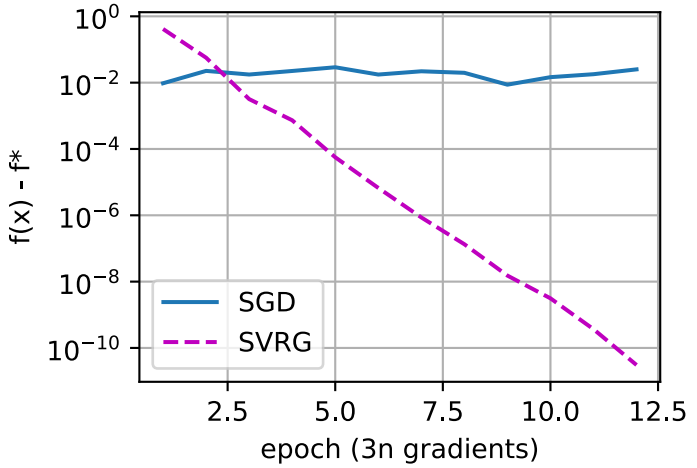
```
L_max = 29, 1 / (4 L_max) = 0.0088
```

Across random draws of the data, $L_{\max}$ is typically between 25 and 40, putting the theorem's step-size ceiling $1/(4L_{\max})$ between roughly 0.006 and 0.01.

Next, we compare SVRG ($m = n$) against single-example SGD at the same constant step size $\eta = 0.02$ (more than twice the theorem's bound) for 12 epochs costing $3n$ example-gradient evaluations each. We also test our earlier quadratic-floor formula on SGD: its steady-state loss gap should equal the sum of $\eta\sigma_i^2/(2(2-\eta\lambda_i))$ over the eigenvectors of the Hessian, where σ_i^2 is the variance of the example gradients at $\mathbf{x}^\star$ along eigenvector i.

```
eta, epochs = 0.02, 12
f_star_r = objective(x_star_r, A_r, y_r, reg_r)
snap, gaps_svrg = np.zeros(d_r), []
for _ in range(epochs):                          # n + 2n example gradients per epoch
    g_snap = full_grad_sum(snap, A_r, y_r, reg_r)
    x = snap.copy()
    for i in rng.integers(0, n_r, size=n_r):
        x -= eta * (component_grad(x, i, A_r, y_r, reg_r)
                    - component_grad(snap, i, A_r, y_r, reg_r) + g_snap)
    snap = x
    gaps_svrg.append(objective(snap, A_r, y_r, reg_r) - f_star_r)
x, gaps_sgd = np.zeros(d_r), []
for _ in range(epochs):
    for i in rng.integers(0, n_r, size=3 * n_r):  # the same 3n budget per epoch
        x -= eta * component_grad(x, i, A_r, y_r, reg_r)
    gaps_sgd.append(objective(x, A_r, y_r, reg_r) - f_star_r)
G_r = component_grad(x_star_r, np.arange(n_r), A_r, y_r, reg_r)  # noise at x*
ev, Q = np.linalg.eigh(H_r)                      # Hessian eigenvalues, eigenvectors
s2 = ((G_r @ Q)**2).mean(axis=0)                 # noise variance per eigenvector
floor = (eta * s2 / (2 * (2 - eta * ev))).sum()  # the loss floors add
print(f'SGD: predicted floor {floor:.1e}, mean gap in epochs 4-12 '
      f'{np.mean(gaps_sgd[3:]):.1e}')
print('SVRG: gap after epochs 1, 4, 12: '
      + ', '.join(f'{gaps_svrg[e - 1]:.0e}' for e in (1, 4, 12)))
d2l.plot(np.arange(1, epochs + 1), [gaps_sgd, gaps_svrg], 'epoch (3n gradients)',
         'f(x) - f*', yscale='log', legend=['SGD', 'SVRG'])
```

```
SGD: predicted floor 1.9e-02, mean gap in epochs 4-12 2.0e-02
SVRG: gap after epochs 1, 4, 12: 4e-01, 7e-04, 3e-11
```



SGD drops rapidly at first and then plateaus at a few times 10^{-2}, right where our floor formula predicts. Across runs, the measured plateau falls between roughly 10% below and 40% above the prediction—slightly above more often than below, because the formula evaluates gradient variance only at $\mathbf{x}^\star$, whereas the example gradients spread slightly more away from the minimizer. SVRG starts behind and finishes far ahead. After its first epoch (which takes only n inner steps anchored at the starting point), its gap still exceeds SGD's. From epoch 2 onward, however, it shrinks by roughly a factor of eight per epoch (between five and thirteen across runs), reaching 10^{-13} to 10^{-8} by epoch 12. Notably, it achieves this linear convergence at a step size more than twice the theorem's limit, showing that the theoretical step condition is sufficient rather than necessary. Keep in mind the three assumptions behind this win: a finite sum, multiple passes over it, and strong convexity for the linear rate. On streaming data, where each example is seen only once, there is no finite sum to snapshot.

SAGA (Defazio et al. 2014) avoids periodic full passes altogether by replacing the snapshot with a table $\mathbf{t}_1, \dots, \mathbf{t}_n$ storing the most recent gradient evaluated for each example. At each step it forms the estimator $\nabla f_i(\mathbf{x}) - \mathbf{t}_i + \frac{1}{n}\sum_j \mathbf{t}_j$ and then overwrites $\mathbf{t}_i$ with $\nabla f_i(\mathbf{x})$. SAGA computes only one new example gradient per step, at the price of storing $n \times d$ numbers (or just one scalar per example for generalized linear models; see exercise 3).

On deep neural networks, unfortunately, neither SVRG nor SAGA has delivered consistent gains: the control variate quickly goes stale when the loss landscape shifts rapidly or when data augmentation perturbs each example on every pass. Meanwhile, a second limitation has been with us since the ill-conditioned valley: every method in this section applies a single global step size to all coordinates, forcing the stiffest direction of our running example to cap η at $2/1000$. In the next section, we remove that bottleneck by letting each coordinate adapt its own step size from the gradients it has observed.

35.4.5 Summary

A minibatch gradient of size b is an unbiased estimator of ∇f whose noise power is $\operatorname{tr}\Sigma/b$. On a quadratic, a fixed step size η drives the second moment of each coordinate geometrically at rate $(1-\eta\lambda)^2$ onto the noise floor $\eta\sigma^2/(\lambda(2-\eta\lambda))$, and the per-coordinate loss floors $\eta\sigma^2/(2(2-\eta\lambda))$ add up across dimensions. Because the floor grows linearly with η

while the transient shrinks faster at larger η, no constant step reaches the minimizer, and every technique in this section manages that trade-off. A Robbins–Monro schedule removes the floor, with $\eta_k = c/k$ achieving $O(1/k)$ only when $c > 1/(2\mu)$. Over a fixed budget of K steps, a constant step $\eta \propto 1/\sqrt{K}$ guarantees $DG/\sqrt{K}$ for the averaged iterate on convex losses and $O(\sigma\sqrt{L\Delta/K})$ for the expected squared gradient at a random iterate on smooth nonconvex losses. Finally, on a finite sum, the SVRG control variate stays unbiased with variance at most $L_{\max}^2\|\mathbf{x} - \tilde{\mathbf{x}}\|^2$, allowing a fixed step size to converge linearly on strongly convex problems.

35.4.6 Exercises

1. code **Noise scale and critical batch size.** ● How much does one minibatch step lower the loss, and how does the gain depend on b? Write $\mathbf{g} = \nabla f(\mathbf{x})$ and $\mathbf{H} = \nabla^2 f(\mathbf{x})$, and approximate the loss after the step $\mathbf{x} - \eta\,\hat{\mathbf{g}}_b$ of Equation 35.38 by its second-order Taylor expansion at $\mathbf{x}$. The analysis follows McCandlish et al. (2018) .
   1. Use the covariance Σ/b of $\hat{\mathbf{g}}_b$, established in the proof of Equation 35.39, to show that the expected loss after the step is minimized by $\eta_{\text{opt}}(b) = \eta_{\max}/(1 + B_{\text{noise}}/b)$, with expected improvement $\Delta_{\max}/(1 + B_{\text{noise}}/b)$, where
   $$\eta_{\max} = \frac{\|\mathbf{g}\|^2}{\mathbf{g}^\top \mathbf{H}\mathbf{g}}, \qquad \Delta_{\max} = \frac{\|\mathbf{g}\|^4}{2\,\mathbf{g}^\top \mathbf{H}\mathbf{g}}, \qquad B_{\text{noise}} = \frac{\operatorname{tr}(\mathbf{H}\Sigma)}{\mathbf{g}^\top \mathbf{H}\mathbf{g}}.$$
   2. The linear scaling rule of Goyal et al. (2017) multiplies η by the same factor as b. For which batch sizes does this rule follow from $\eta_{\text{opt}}(b)$? What does a larger batch gain per step once $b \gg B_{\text{noise}}$?
   3. For $\mathbf{H}$ proportional to the identity, B_{noise} reduces to $B_{\text{simple}} = \operatorname{tr}\Sigma/\|\mathbf{g}\|^2$, the gradient-noise scale b_{noise} of the text. Compute B_{simple} and B_{noise} for the logistic-regression problem of this section at $\mathbf{x} = \mathbf{0}$ and after 5, 10, and 20 full-gradient steps with $\eta = 0.5$, using `example_grads` and the Hessian $\mathbf{H}(\mathbf{x}) = \frac{1}{n}\sum_i s_i(1 - s_i)\,\mathbf{a}_i\mathbf{a}_i^\top + 0.1\,\mathbf{I}$ with $s_i = 1/(1 + e^{y_i \mathbf{a}_i^\top \mathbf{x}})$. How does the noise scale change as the iterates approach the optimum? Treat B_{noise} as the critical batch size that separates the two regimes of (b). How should the batch size change over a training run?
2. code **The constant in a $1/k$ schedule.** ●● The text found the constant A of Equation 35.43 to leading order. This exercise makes it rigorous and carries it to the running example.
   1. Let $t = k + k_0$ and suppose $u_{k+1} \le (1 - a/t)\,u_k + q/t^2$ for all $k \ge 0$, with $a > 1$, $q > 0$, $u_k \ge 0$ and $k_0 \ge a$. Prove by induction that $u_k \le \nu/(k + k_0)$ with $\nu = \max\{q/(a-1),\ k_0 u_0\}$. (Hint: $1 - a/t \ge 0$, and $1/t - 1/t^2 \le 1/(t+1)$.)
   2. Apply the lemma to the recursion $e_{k+1} = (1 - c\mu/t)^2 e_k + c^2\sigma^2/t^2$. Show that $(1 - c\mu/t)^2 \le 1 - (2c\mu - (c\mu)^2/k_0)/t$ for $t \ge k_0$, so that any $2c\mu > 1$ gives $e_k = O(1/k)$ once k_0 is large enough. Identify a and q, and show that $q/(a-1) \to A$ as $k_0 \to \infty$.
   3. For $2c\mu < 1$, show $e_k = \Theta(k^{-2c\mu})$ from the product $\prod_{j<k}(1 - c\mu/(j + k_0))^2$ and the unrolled noise sum.
   4. Iterate the recursion with $\mu = \sigma = 1$, $k_0 = 10$ and $e_0 = 1$ up to $k = 10^6$ for $c \in \{0.1, 0.25, 0.5, 1, 2\}$. Report the log–log slope between $k = 10^5$ and 10^6 and the value of $k\,e_k$ at 10^6. Which costs more, a constant twice the minimizer of A or half of it?

5. On the running example $\frac{1}{2}(x_1^2+1000\,x_2^2)$ with unit noise per coordinate, stability at the first step forces $c/k_0 < 2/1000$. Take $c = 1/\mu = 1$, $k_0 = 1000$, $\mathbf{x}_0 = (1,1)$ and $K = 10^4$. Compute the expected loss by the two-coordinate recursion (it is about 4.2×10^{-3}) and compare it with the constant step $\eta = 10^{-3}$ (7.5×10^{-4}) and with cosine decay from the same η_0 (1.0×10^{-4}). Which coordinate does the $1/k$ decay starve, and why does the condition number reappear in k_0?

3. code **SVRG inner loop and SAGA.** Use the ridge data and the functions `component_grad`, `full_grad_sum` and `objective` of this section with a budget of $36n = 10{,}800$ example-gradient evaluations, the budget behind the 12 reported epochs of the SVRG cell. Measure each run by the number of evaluations it needs to bring the gap below 10^{-10}, checking once per epoch.
   1. Let the inner loop run m steps, so that one epoch costs $n + 2m$ evaluations, and vary $m \in \{10, 30, 75, 300, 1200\}$ at $\eta \in \{0.01, 0.02\}$. Which m is best at each step size? Explain the trade-off between amortizing the snapshot gradient and letting the control variate go stale, and why the best m moves with η.
   2. Implement SAGA: keep a table with the most recent gradient of every example, filled at $\mathbf{x}_0$ for n evaluations, and use $\nabla f_i(\mathbf{x}) - \mathbf{t}_i + \frac{1}{n}\sum_j \mathbf{t}_j$ as the estimator, where $\mathbf{t}_i$ is the table entry of example i. Compare SAGA with the best SVRG run at both step sizes. How many numbers does the table hold for this problem, and how many would suffice if you stored only the scalar $\mathbf{a}_i^\top \mathbf{x} - y_i$ per example?
4. **Biased gradient estimators.** ● Keep the variance bound of Equation 35.45, now measured around the conditional mean, but replace unbiasedness by $\mathbb{E}[\mathbf{g}_k \mid \mathbf{x}_k] = \nabla f(\mathbf{x}_k) + \delta_k$ with $\|\delta_k\| \le \delta$.
   1. Redo the proof of Equation 35.46 for $\eta \le 1/L$ and show

$$\frac{1}{K}\sum_{k=0}^{K-1} \mathbb{E}\left[\|\nabla f(\mathbf{x}_k)\|^2\right] \;\le\; \frac{2\Delta}{\eta K} + L\eta\sigma^2 + \delta^2. \tag{35.50}$$

      Use $L\eta \le 1$ on the term $\frac{L\eta^2}{2}\|\nabla f(\mathbf{x}_k) + \delta_k\|^2$ before expanding it.
   2. Show that the term δ^2 cannot be removed: give a one-dimensional f and a constant bias for which SGD with $\sigma = 0$ converges to a point where $|f'|^2 = \delta^2$.
   3. A single minibatch drawn uniformly without replacement is still an unbiased estimator of the full gradient. Which of the following estimators are biased in this sense: the example gradient at a step of random reshuffling (Section 9.2), which visits every example once per epoch in a fresh random order; a stochastic gradient clipped to a fixed maximum norm; the gradient evaluated at the iterate of τ steps earlier, as in asynchronous training?
5. code **Iterate averaging.** ● Polyak and Juditsky (1992) showed that averaging the iterates of a slowly decaying or constant-step run reaches the best possible constant. Consider the scalar model with a constant step $\eta \in (0, 2/\lambda)$, started in its stationary distribution.
   1. Show that $\mathrm{Cov}(x_j, x_k) = V\rho^{|j-k|}$.
   2. Show that the average $\bar{x}_K = \frac{1}{K}\sum_{k<K} x_k$ has $\mathbb{E}[\bar{x}_K^2] = \frac{V}{K}\cdot\frac{1+\rho}{1-\rho} - O(K^{-2})$, and that $V(1+\rho)/(1-\rho) = \sigma^2/\lambda^2$ for every such η: to leading order the averaged iterate does not depend on the step.
   3. Check by simulation with $K = 2000$ and $4{,}000$ runs at $\eta = 0.3$ and $\eta = 0.1$, with $\lambda = \sigma = 1$. You should find $\mathbb{E}[\bar{x}_K^2] \approx 5 \times 10^{-4}$ in both cases, against last-iterate floors of 0.18 and 0.053.
   4. Started at $x_0 = 3$ instead, the average remembers the transient. Show that its

contribution to $\mathbb{E}[\bar{x}_K^2]$ is about $(x_0/(\eta\lambda K))^2$, and decide when averaging only the second half of the run is better.

5. Compare with the best $1/k$ schedule, whose constant is $A = \sigma^2/\mu^2$. What does each of the two routes to the rate $\sigma^2/(\mu^2 K)$ require?

6. code **Interpolation and linear convergence.** ●● Suppose each f_i is convex with an L_i-Lipschitz gradient, f is μ-strongly convex, and every f_i is minimized at the same point $\mathbf{x}^\star$, as when a model fits every training example exactly. The analysis follows Vaswani et al. (2019) .
   1. Show that $\mathbb{E}_i\|\nabla f_i(\mathbf{x})\|^2 \le 2L_{\max}(f(\mathbf{x}) - f^\star)$. (Hint: a convex L-smooth h minimized at $\mathbf{x}^\star$ satisfies $\|\nabla h(\mathbf{x})\|^2 \le 2L(h(\mathbf{x}) - h(\mathbf{x}^\star))$.)
   2. Deduce that single-example SGD with a constant $\eta \le 1/L_{\max}$ satisfies $\mathbb{E}\|\mathbf{x}_{k+1} - \mathbf{x}^\star\|^2 \le (1 - \eta\mu)\,\mathbb{E}\|\mathbf{x}_k - \mathbf{x}^\star\|^2$: linear convergence with no floor. What is $\operatorname{tr}\Sigma(\mathbf{x}^\star)$ in this case?
   3. Check it on least squares with $n = 50$ examples and $d = 100$ parameters, $\eta = 1/L_{\max}$ and $\mathbf{x}_0 = \mathbf{0}$. The iterates stay in the row space of $\mathbf{A}$, where f is strongly convex, so (b) applies there. Show that the gap falls linearly with consistent labels and also with labels corrupted by noise of standard deviation 0.5, and explain why the noise does not matter here.
   4. Repeat with $n = 200$ and $d = 100$. Now the noisy labels bring the floor back. Why?

35.5 Adaptive Methods

The transformers in this book, like most large neural networks, are trained with Adam or its variant AdamW (Section 9.5.3) rather than with plain stochastic gradient descent. A primary reason is the step size. The methods of Section 35.4 apply a single step size to every coordinate, adjusting it only over time. Yet a modern network has 10^7 to 10^{11} parameters whose curvatures differ by orders of magnitude and are unknown in advance. When all coordinates share one step size, the steepest direction caps the step and the flat directions barely move. This section asks how an optimizer can choose a separate step size for each coordinate from the gradients alone—and where that strategy breaks down.

Our running example illustrates how much is at stake. Consider the narrow valley $f(\mathbf{x}) = \frac{1}{2}(x_1^2 + 1000\,x_2^2)$ starting at $(1, 1)$. Gradient descent with the optimal single step size, $\eta = 2/1001$, needs 6,160 steps to drive f below 10^{-8}, because a step small enough to keep x_2 stable is a thousand times too small for x_1. The heavy-ball method of Section 35.1, which adds a fraction of the previous step to each new one, cuts that to 296 steps. (Both counts are a little below the $6{,}908$ and 315 steps quoted there, because that section stopped on the distance to the minimizer and this one stops on the value of f.) An optimizer that gives each coordinate its own step size, estimated purely from the gradients it has observed, needs only a few dozen. Even more striking is what that count does not depend on: make the valley a hundred times narrower still, and the iteration count grows by just six steps.

Headline counts like these are easy to produce and easy to misread, so much of this section is devoted to making the comparison fair. We will run heavy ball at its theoretically optimal settings, give Adam a simple sweep of step sizes, and also rotate the valley by 45°. Under that fairer test, Adam's advantage looks quite different from what the first count suggests.

The underlying mechanism is simple: divide each coordinate's step by the root mean square of its past gradients, as AdaGrad, RMSProp, and Adam do. We first prove that this normal-

ization makes Adam blind to the scale of each coordinate, and then follow that symmetry to the boundaries where it breaks: a rotation of the valley, a periodic sequence of convex losses, weight decay, the opening steps of training, and the matrix structure of a layer. Two results carry the most weight: the scale-invariance proposition and the fixed-point analysis of AdamW. The regret bounds for AdaGrad and the structured preconditioners at the end can be skipped on a first reading. Throughout, t indexes optimizer steps, h_i denotes the curvature of coordinate i, and λ is the weight-decay coefficient.

35.5.1 Per-Coordinate Step Sizes

35.5.1.1 A Diagonal Quadratic

Gradient descent on a quadratic requires a number of steps proportional to the condition number $\kappa = h_{\max}/h_{\min}$, because its single step size must stay below $2/h_{\max}$ (Section 35.1). The simplest setting where a remedy becomes visible is the two-variable diagonal quadratic,

$$f(\mathbf{x}) = \tfrac{1}{2}(h_1 x_1^2 + h_2 x_2^2), \qquad \frac{\partial f}{\partial x_i} = h_i x_i.$$

Because each partial derivative depends only on its own coordinate, nothing forces the two coordinates to share a step size. With a separate step η_i for each coordinate, x_i updates to $x_i - \eta_i h_i x_i = (1 - \eta_i h_i)\, x_i$, and choosing $\eta_i = 1/h_i$ sends every coordinate to zero in a single step. For $h = (1, 1000)$, the ideal steps are 1 and 10^{-3}. This is Newton's method for a diagonal Hessian, and the condition number vanishes from the iteration count.

Two obstacles stand between this ideal and a practical optimizer: the curvatures h_i are unknown and shift over the course of training, and the gradients we observe are noisy mini-batch estimates. We must therefore infer each coordinate's step size from the gradients alone, using gradient magnitude as a proxy. On the diagonal quadratic, the gradient entry $g_i = h_i x_i$ has magnitude $h_i |x_i|$. Whenever coordinates are of comparable size, a coordinate with large gradients also has high curvature, so dividing by its gradient magnitude effectively divides by its curvature. When coordinates differ sharply in scale, however, $g_i/|g_i|$ still has unit magnitude: normalization fixes the distance traveled per step at η, rather than taking the true Newton step x_i.

35.5.1.2 AdaGrad

Varying curvature is one reason to give each coordinate its own step size. A second, independent reason is that coordinates are updated at very different frequencies. A global decaying step size such as $\eta/\sqrt{t}$—the classical choice in online and nonsmooth convex optimization—treats every coordinate identically. Suppose coordinate 1 receives a gradient of ± 1 at every step while coordinate 2 receives one only once every 10^4 steps, as happens for a rare word's weight in a bag-of-words model. By $t = 10^4$, a global schedule reduces both step sizes to $\eta/100$, so the single piece of evidence about the rare coordinate moves it by only a hundredth of η. If instead each coordinate accumulates its own squared gradients, the running sums are 10^4 and 1, and dividing η by their square roots gives the busy coordinate $\eta/100$ while leaving the rare coordinate at η.

This is **AdaGrad** (Duchi et al. 2011). Write $\mathbf{g}_t \odot \mathbf{g}_t$ for the vector of squared gradient entries (abbreviated as $\mathbf{g}_t^2$ from here on), with square roots and divisions of vectors likewise applied

elementwise. AdaGrad maintains the running sum $\mathbf{s}_t$ of squared gradients and divides the step by its square root:

$$\mathbf{s}_t = \mathbf{s}_{t-1} + \mathbf{g}_t \odot \mathbf{g}_t, \qquad \mathbf{x}_{t+1} = \mathbf{x}_t - \frac{\eta}{\sqrt{\mathbf{s}_t} + \epsilon}\,\mathbf{g}_t, \tag{35.51}$$

starting from $\mathbf{s}_0 = \mathbf{0}$, where a small constant ϵ prevents division by zero (Section 9.5 in the main text places ϵ inside the square root, which makes no qualitative difference here). Each coordinate's step size now decays according to its own gradient history rather than the global step count.

The metric view. Steepest descent depends on how we measure step length. Under the weighted norm $\|\mathbf{d}\|_{\mathbf{P}} = \sqrt{\mathbf{d}^\top \mathbf{P}\,\mathbf{d}}$ for a positive definite matrix $\mathbf{P}$, the steepest descent direction is $-\mathbf{P}^{-1}\nabla f$ (Exercise 1 of Section 35.1). AdaGrad is steepest descent in the diagonal metric $\mathbf{P}_t = \operatorname{diag}(\sqrt{\mathbf{s}_t})$: moving along coordinate i is penalized in proportion to the accumulated evidence $\sqrt{\sum_{\tau \le t} g_{\tau,i}^2}$ that its gradient has been large. A matrix such as $\mathbf{P}_t$ that rescales the gradient before taking a step is called a **preconditioner**. Whereas Newton's method uses the true curvature $\operatorname{diag}(h_i)$, AdaGrad substitutes the gradient history (Exercise 1).

The regret view. AdaGrad was originally designed for **online convex optimization**, in which convex losses $f_1, f_2, \dots$ arrive one at a time and the learner must choose $\mathbf{x}_t$ before seeing f_t. Performance over T rounds is measured by the **regret**, $\sum_{t=1}^{T} f_t(\mathbf{x}_t) - \min_{\mathbf{x}} \sum_{t=1}^{T} f_t(\mathbf{x})$: the cumulative excess loss relative to the best fixed point chosen in hindsight. An algorithm learns in this framework when its average regret (regret divided by T) tends to zero.

On a convex set of Euclidean diameter D_2 with bounded gradients $\|\mathbf{g}_t\| \le G$, online gradient descent with a tuned global step size achieves regret of order $D_2 G\sqrt{T}$ (proved in Exercise 4). AdaGrad, by contrast, achieves regret of order (Duchi et al. 2011)

$$D_\infty \sum_{i=1}^{d} \sqrt{\textstyle\sum_{t=1}^{T} g_{t,i}^2},$$

where D_∞ is the diameter of the set in the ℓ_∞ norm. Each of the d coordinates pays only for its own gradient history rather than for the worst-case gradient norm across all coordinates.

On the box $[-1, 1]^d$, we have $D_2 = \sqrt{d}\, D_\infty$. When gradient mass concentrates on a small subset of coordinates, AdaGrad's bound is up to a factor of $\sqrt{d}$ smaller; when the mass is spread evenly across coordinates, the two bounds match. For one-hot gradients that hit coordinate i with probability p_i, the expected ratio of the two bounds is at most $\sum_i \sqrt{p_i}/\sqrt{d}$ (Exercise 2). For uniform p_i this ratio is 1.00 even though every individual gradient is sparse, whereas for word frequencies following Zipf's law with exponent 1 at $d = 10^4$ it drops to 0.63. AdaGrad's advantage therefore comes from the *concentration* of gradient mass on certain coordinates, not from sparsity by itself.

The square root in Equation 35.51 is not arbitrary: it is the exact exponent that balances the two competing terms in the regret proof (Exercise 4c).

Outside online convex optimization, however, unbounded accumulation becomes AdaGrad's main weakness. Under persistent gradient noise of variance σ^2, the sum grows as

$s_{t,i} \approx \sigma^2 t$, so the effective step size shrinks as $\eta/(\sigma\sqrt{t})$. AdaGrad thus carries a built-in decay schedule tied to the step count rather than to optimization progress. That $1/\sqrt{t}$ rate is optimal for convex regret, but on long nonconvex training runs it causes the step size to decay relentlessly whether or not the loss is still improving, leaving almost no step size by the time the optimizer reaches a late plateau.

35.5.1.3 RMSProp

RMSProp (Tieleman and Hinton 2012) fixes this decay by replacing the all-time sum with an exponential moving average, keeping the scale estimate responsive to recent gradients:

$$\mathbf{v}_t = \beta_2\, \mathbf{v}_{t-1} + (1-\beta_2)\, \mathbf{g}_t^2, \qquad \mathbf{x}_{t+1} = \mathbf{x}_t - \frac{\eta}{\sqrt{\mathbf{v}_t} + \epsilon}\, \mathbf{g}_t. \tag{35.52}$$

Unrolling the recursion shows that $\mathbf{v}_t$ assigns the squared gradient from k steps ago the weight $(1-\beta_2)\beta_2^k$, so old gradients halve in weight roughly every $\log 2/(1-\beta_2)$ steps—every 7 steps at RMSProp's customary $\beta_2 = 0.9$, and every 693 steps at $\beta_2 = 0.999$. Step sizes no longer vanish by construction. The trade-off is that any single large gradient is forgotten on that same timescale, a vulnerability we will exploit in a counterexample below.

35.5.1.4 Adam and Its Bias Correction

Adam (Kingma and Ba 2015) combines RMSProp's second-moment estimate with an exponential moving average of the gradient itself, and corrects both averages for their initialization at zero. In the notation of Section 9.5, the update reads

$$\begin{aligned} \mathbf{m}_t &= \beta_1\, \mathbf{m}_{t-1} + (1-\beta_1)\, \mathbf{g}_t, & \hat{\mathbf{m}}_t &= \mathbf{m}_t/(1-\beta_1^t), \\ \mathbf{v}_t &= \beta_2\, \mathbf{v}_{t-1} + (1-\beta_2)\, \mathbf{g}_t^2, & \hat{\mathbf{v}}_t &= \mathbf{v}_t/(1-\beta_2^t), \\ \mathbf{x}_{t+1} &= \mathbf{x}_t - \eta\, \frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{v}}_t} + \epsilon}, \end{aligned} \tag{35.53}$$

with $\mathbf{m}_0 = \mathbf{v}_0 = \mathbf{0}$ and the standard defaults $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-8}$. The first moment $\mathbf{m}_t$ is the heavy-ball velocity of Section 35.1.3 scaled by $1-\beta_1$, which turns a running sum into a weighted average and smooths the minibatch noise that plain RMSProp steps along directly.

Dividing by $1-\beta_2^t$ compensates for starting from $\mathbf{v}_0 = \mathbf{0}$. Unrolling the recursion gives $\mathbf{v}_t = (1-\beta_2)\sum_{\tau \le t} \beta_2^{t-\tau}\, \mathbf{g}_\tau^2$. If the squared-gradient scale is stationary, with $\mathbb{E}[\mathbf{g}_\tau^2] = \overline{\mathbf{g}^2}$ for every τ, summing the geometric series yields

$$\mathbb{E}[\mathbf{v}_t] = (1-\beta_2)\, \overline{\mathbf{g}^2} \sum_{\tau=1}^{t} \beta_2^{t-\tau} = (1-\beta_2^t)\, \overline{\mathbf{g}^2}. \tag{35.54}$$

Thus $\hat{\mathbf{v}}_t$ is an unbiased estimate of $\overline{\mathbf{g}^2}$ at every step t under stationarity, and the same calculation with β_1 shows that $\hat{\mathbf{m}}_t$ is unbiased for the mean gradient.

Without this correction, the initialization deficit is both large and long-lived: at the default $\beta_2 = 0.999$, $\mathbf{v}_{10}$ captures only about 1% of its stationary target, and even at $t = 1000$ it has

reached just 63% (Section 9.5). More striking is how the error evolves over time. Dropping both bias corrections mis-scales the update $\mathbf{m}_t/\sqrt{\mathbf{v}_t}$ by the factor $(1-\beta_1^t)/\sqrt{1-\beta_2^t}$, which starts at 3.16 at $t=1$ and, perhaps surprisingly, climbs to a peak of 6.6 at $t=12$. The first-moment average fills in roughly ten steps, whereas the second-moment average needs about a thousand. Figure 35.18 plots both the deficit and the resulting step inflation.

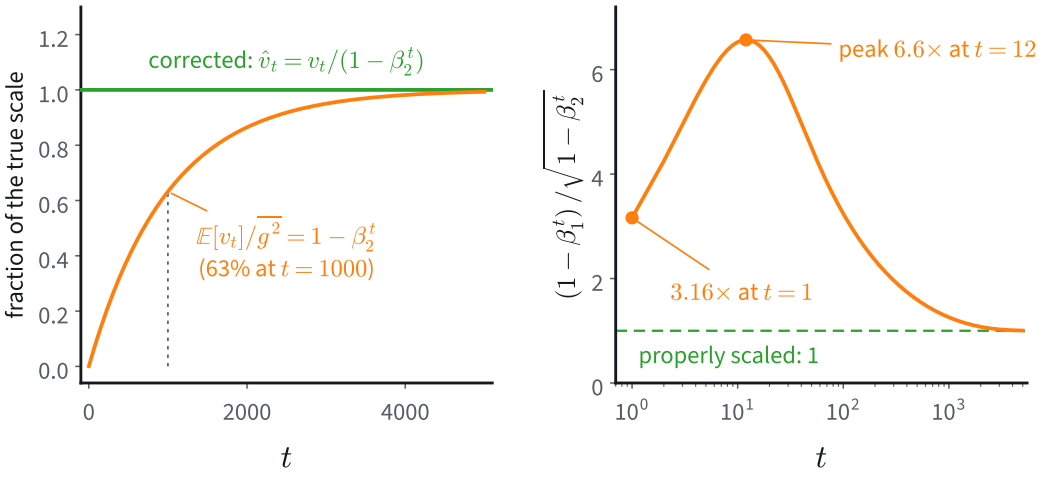


Figure 35.18: Bias correction in Adam. Left: starting from zero, the running average $\mathbf{v}_t$ captures only the fraction $1-\beta_2^t$ of the squared-gradient scale (63% at $t=1000$ for $\beta_2 = 0.999$); dividing by $1-\beta_2^t$ restores the full scale at every step. Right: without bias correction, the update $\mathbf{m}_t/\sqrt{\mathbf{v}_t}$ is inflated by $(1-\beta_1^t)/\sqrt{1-\beta_2^t}$, starting at 3.16 at $t=1$, peaking at 6.6 at $t=12$ as the numerator fills in ten steps while the denominator takes a thousand, and returning to 1 only on the $1/(1-\beta_2)$ timescale. Uncorrected Adam thus takes its largest steps when its preconditioner rests on the fewest gradients.

Even an unbiased $\hat{\mathbf{v}}_t$ does not make the step size itself unbiased. Because Adam divides by $\sqrt{\hat{\mathbf{v}}_t}$ rather than $\hat{\mathbf{v}}_t$, concavity of the square root biases the denominator downward: by Jensen's inequality, $\mathbb{E}\sqrt{\hat{v}_t} < \sqrt{\mathbb{E}[\hat{v}_t]}$ unless $\hat{v}_t$ is constant. The first step is the extreme case. At $t=1$, $\hat{m}_1 = g_1$ and $\sqrt{\hat{v}_1} = |g_1|$, so the ratio is $\operatorname{sign}(g_1)$ and the step has magnitude η in every coordinate, whatever the gradient, rather than $\eta\,|g_1|/\sigma$ for a gradient of scale σ. We examine how much this matters in Section 35.5.4.

35.5.1.5 Scale Invariance

In our opening example, $h_2 = 1000\,h_1$, yet Adam crosses the valley as if that thousandfold ratio did not exist. The reason is an exact symmetry of the update, and it is the central result of this section.

Proposition (Adam ignores the scale of each coordinate). *Let $\epsilon = 0$ and let $\mathbf{C} = \operatorname{diag}(c_1, \dots, c_d)$ with every $c_i > 0$. Suppose two problems have gradients related by $\nabla \tilde{f}(\mathbf{x}) = \mathbf{C}\,\nabla f(\mathbf{x})$ at every $\mathbf{x}$. Then Adam with the same β_1, β_2, step schedule, and starting point produces identical iterates on both problems, as long as the updates remain defined.*

In other words, the second problem is the first with coordinate i's gradient rescaled by a positive constant c_i. We denote its quantities with a tilde.

Proof. We proceed by induction on t. Suppose the iterates agree up to $\mathbf{x}_t$. Then $\tilde{\mathbf{g}}_\tau = \mathbf{C}\mathbf{g}_\tau$ for every $\tau \le t$. Because the recursions for $\mathbf{m}$ and $\mathbf{v}$ are linear in $\mathbf{g}_\tau$ and $\mathbf{g}_\tau^2$, we obtain $\tilde{m}_{t,i} = c_i m_{t,i}$ and $\tilde{v}_{t,i} = c_i^2 v_{t,i}$. The bias-correction denominators are the same scalars in both problems, so the corrected ratio $\hat{m}_{t,i}/\sqrt{\hat{v}_{t,i}}$ of the second problem is $c_i\hat{m}_{t,i}/(c_i\sqrt{\hat{v}_{t,i}}) = \hat{m}_{t,i}/\sqrt{\hat{v}_{t,i}}$. The updates therefore coincide, giving $\tilde{\mathbf{x}}_{t+1} = \mathbf{x}_{t+1}$. ■

On the diagonal quadratic, $\nabla f(\mathbf{x}) = \operatorname{diag}(h)\,\mathbf{x}$ is simply the gradient of the round bowl $\frac{1}{2}\|\mathbf{x}\|^2$ multiplied by $\mathbf{C} = \operatorname{diag}(h)$. Consequently, Adam on the $1000{:}1$ valley traces, step for step, the exact same trajectory as Adam on the round bowl—and on the bowl, two coordinates started at equal values remain equal forever. Each coordinate follows the same one-dimensional iteration on $\frac{1}{2}x^2$,

$$x_{t+1} = x_t\Big(1 - \frac{\eta}{\sqrt{\hat{v}_t}}\Big) \qquad (\beta_1 = 0),$$

where $\hat{v}_t$ averages the past squares x_τ^2 for $\tau \le t$. Because earlier iterates were larger, $\sqrt{\hat{v}_t}$ exceeds $|x_t|$ from the second step onward, so each update moves by less than η and contracts geometrically by the factor $1 - \eta/\sqrt{\hat{v}_t}$. At $\eta = 0.1$, this factor tightens from 0.9 at $t = 1$ to 0.67 at $t = 48$ as $\hat{v}_t$ catches up with the shrinking iterates. The iterates themselves are completely independent of h_2/h_1. Only the stopping test depends on the curvature ratio, because $f(\mathbf{x}_t) = \frac{1}{2}(h_1 + h_2)\,x_t^2$ weights the common iterate by the curvatures. Making the valley far narrower should therefore add only a handful of geometric contraction steps, never a factor of κ.

What the step count *does* depend on is the starting distance measured in units of η, since the early steps have length roughly η per coordinate regardless of curvature. An experiment in the next subsection checks both predictions directly.

In short, Adam measures every coordinate in units of its own gradient size, making coordinate-wise scaling invisible to it. Three caveats bound this invariance. First, $\epsilon > 0$ breaks the symmetry once a coordinate's gradients shrink to the order of ϵ. Second, the proposition says nothing about a change of basis that mixes coordinates, which we test next on the rotated valley. Third, Adam is not in general sign descent: its update reduces to the **sign step** $\eta\,\operatorname{sign}(\mathbf{g}_t)$ in only three special cases—at $t = 1$; when $\beta_1 = \beta_2 = 0$ (Exercise 5d); and, for $\beta_1 = 0$, whenever each coordinate's gradient magnitude stays constant over the memory of $\hat{\mathbf{v}}$.

35.5.2 Coordinate Scaling and Its Limits

35.5.2.1 Aligned and Rotated Valleys

How much of gradient descent's 6,160 steps does coordinate scaling alone eliminate, and does that gain survive when the valley is rotated? To separate per-coordinate scaling from momentum, we compare four methods: gradient descent at its optimal single step size; heavy ball at its optimal step size and momentum; Adam with $\beta_1 = 0$, which only rescales; and Adam with its default momentum $\beta_1 = 0.9$. To rotate the problem by an angle θ, we replace $\operatorname{diag}(h)$ with $\mathbf{A}_\theta = \mathbf{R}_\theta \operatorname{diag}(h)\,\mathbf{R}_\theta^\top$, where $\mathbf{R}_\theta$ is the 2×2 rotation matrix, and rotate the starting point alongside it, $\mathbf{x}_0 = \mathbf{R}_\theta(1, 1)^\top$, so that the initial point retains the same components along both curvature axes. At $\theta = \pi/4$, this gives $\mathbf{A} = \left(\begin{smallmatrix} 500.5 & -499.5 \\ -499.5 & 500.5 \end{smallmatrix}\right)$. The

code below defines the valley, a loop that counts steps to reach the target tolerance, and the three optimizer updates.

```
h = np.array([1.0, 1000.0])                    # curvatures h_i, kappa = 1000

def rotated(theta):
    """A = R diag(h) R^T and the start R (1, 1): the valley turned by theta."""
    c, s = np.cos(theta), np.sin(theta)
    R = np.array([[c, -s], [s, c]])
    return R @ np.diag(h) @ R.T, R @ np.ones(2)

def run(step, A, x0, K=40000, tol=1e-8):
    """First t with f(x_t) = x_t^T A x_t / 2 < tol, or None within K steps."""
    x, state = x0.copy(), {}
    for t in range(1, K + 1):
        x = step(x, A @ x, t, state)
        if 0.5 * x @ A @ x < tol:
            return t
    return None

def gd(eta):
    return lambda x, g, t, state: x - eta * g

def heavy_ball(eta, beta):
    def step(x, g, t, state):
        state['d'] = beta * state.get('d', 0.0) - eta * g    # velocity
        return x + state['d']
    return step

def adam(eta, beta1=0.9, beta2=0.999, eps=1e-8):
    def step(x, g, t, state):
        state['m'] = beta1 * state.get('m', 0.0) + (1 - beta1) * g
        state['v'] = beta2 * state.get('v', 0.0) + (1 - beta2) * g * g
        m_hat, v_hat = state['m'] / (1 - beta1**t), state['v'] / (1 - beta2**t)
        return x - eta * m_hat / (np.sqrt(v_hat) + eps)
    return step
```

Before reading the step counts, notice what information each method is given. Gradient descent and heavy ball run at their theoretically optimal settings, computed in closed form from the true curvatures 1 and 1000. That is a generous handicap, since in practice nobody knows the extreme curvatures of a neural network. Adam has no closed-form formula, so we give it what a practitioner would use: a log-spaced sweep of seven step sizes from 0.002 to 2 (spaced by factors of about three), reading each row at its best entry. The grid deliberately skips $\eta = 1$: because Adam's first step has length η in every coordinate (Section 35.5.4), starting at $(1, 1)$ with $\eta = 1$ would land directly on the origin in one step. That would be a coincidence of the starting point, not a fair measure of optimizer speed.

```
mu, L = h.min(), h.max()
rho = (np.sqrt(L) - np.sqrt(mu)) / (np.sqrt(L) + np.sqrt(mu))
tuned = {'GD, eta = 2/(mu + L)': gd(2 / (mu + L)),
         'heavy ball, tuned': heavy_ball(4 / (np.sqrt(L) + np.sqrt(mu))**2,
```

```
                                        rho**2)}
    valleys = {'aligned': rotated(0.0), 'rotated 45': rotated(np.pi / 4)}
    print(f'{"steps to f < 1e-8":28s}{"aligned":>9s}{"rotated 45":>12s}')
    for name, step in tuned.items():
        hits = [run(step, *problem) for problem in valleys.values()]
        print(f'{name:28s}{hits[0] or "never":>9}{hits[1] or "never":>12}')
    etas = np.geomspace(0.002, 2.0, 7)              # step sizes for Adam
    print(f'\n{"Adam, eta =":28s}' + ''.join(f'{eta:>7.2g}' for eta in etas))
    for beta1 in (0.0, 0.9):
        for name, problem in valleys.items():
            hits = [run(adam(eta, beta1=beta1), *problem) for eta in etas]
            print(f'{f"beta1 = {beta1}, {name}":28s}'
                  + ''.join(f'{k or "never":>7}' for k in hits))
```

```
steps to f < 1e-8             aligned  rotated 45
GD, eta = 2/(mu + L)             6160        6160
heavy ball, tuned                 296         296

Adam, eta =                  0.002 0.0063   0.02  0.063    0.2   0.63      2
beta1 = 0.0, aligned          2122    760    249     78     22  never  never
beta1 = 0.0, rotated 45      never  never  never  never  never  never  never
beta1 = 0.9, aligned          2033    648    195    165    205    170    196
beta1 = 0.9, rotated 45      12862   9292   5898   3171   1581    799    559
```

Consider the aligned valley first. Per-coordinate scaling alone ($\beta_1 = 0$) needs just 22 steps at its best step size, compared with 6,160 for gradient descent and 296 for heavy ball—thirteen times fewer steps than a heavy-ball method given both true curvatures, from an optimizer told neither. That row also clarifies what tuning Adam actually adjusts. The iteration count scales roughly as $1/\eta$, because the early steps have length about η and the total count mainly reflects the distance to the minimizer measured in step lengths. At the two largest step sizes, however, scaling alone never converges: the iterate overshoots the minimum and settles into a cycle that bounces each coordinate between $+\eta/2$ and $-\eta/2$. Adam has thus traded one unknown (the curvature) for another: the distance to the minimizer in units of η.

Adding Adam's default momentum ($\beta_1 = 0.9$) brings the count to between 165 and 205 steps across the entire range of step sizes from 0.02 to 2. Two features stand out. The first is robustness: a hundredfold change in η shifts the iteration count by only a quarter, and the two largest step sizes that cycled forever without momentum now converge cleanly because the momentum average damps the alternating signs. Heavy ball enjoys no such margin: its optimal step size sits within 0.1% of its stability limit $(2 + 2\beta)/\lambda_{\max}$ (Exercise 8a of Section 35.1), so its 296-step result presupposes near-exact knowledge of the largest curvature. The second feature is the price of that robustness: momentum makes Adam roughly seven times slower than scaling alone. The cause is the $\sqrt{\beta}$ contraction limit from Section 35.1.3: an iteration with momentum β contracts any mode with complex roots at rate $\sqrt{\beta}$ per step, and no faster. After per-coordinate scaling, both coordinates effectively run on a round bowl of condition number one, where the optimal momentum is zero. With $\beta_1 = 0.9$, the contraction factor is $\sqrt{0.9} \approx 0.95$ per step, which predicts roughly 230 steps to reach the tolerance, closely matching the observed counts. Momentum that accelerates progress along an ill-conditioned valley acts as a brake on a round bowl, yet Adam's default applies it unconditionally.

Now turn to the rotated valley. Gradient descent and heavy ball require the exact same

number of steps as before, because their updates are rotation-invariant and simply rotate with the problem. Adam's updates are not. Scaling alone fails to reach the tolerance at every single step size on the grid. Adam with momentum does converge, but only toward the upper end of the grid: its best count is 559 at $\eta = 2$, nearly twice heavy ball's 296, and at the smallest step size it needs 12,862 steps—twice as many as plain gradient descent. Its step-size robustness vanishes too, with counts now spanning more than a twentyfold range across the grid.

The fair verdict therefore has two halves. When the coordinate axes align with the principal curvature directions, per-coordinate scaling beats a perfectly tuned heavy ball by an order of magnitude while requiring far less prior knowledge. When the valley is rotated, the benefit of diagonal scaling disappears completely; whatever convergence remains comes from momentum, and a well-tuned heavy ball performs that job twice as fast. Let us examine both halves in closer detail.

On the aligned valley, our scale-invariance proposition predicts that both coordinates follow identical iterates while the stiff coordinate receives an effective step a thousand times smaller. Below, we print the iterates and effective step sizes $\eta/\sqrt{\hat{\mathbf{v}}_t}$ of scaling-only Adam ($\epsilon = 0$, $\eta = 0.1$). The last two lines then test what actually controls the iteration count by varying the curvature ratio across five orders of magnitude and moving the starting point a hundred times farther from the origin.

```
A, x = rotated(0.0)
step, state, eta, gap = adam(0.1, beta1=0.0, eps=0.0), {}, 0.1, 0.0
for t in range(1, 49):
    x = step(x, A @ x, t, state)
    gap = max(gap, abs(x[0] - x[1]))
    if t in (1, 10, 48):
        eff = eta / np.sqrt(state['v'] / (1 - 0.999**t))
        print(f't = {t:2d}: x = {x[0]:.3g}, {x[1]:.3g};  '
              f'eta/sqrt(v_hat) = {eff[0]:.3g}, {eff[1]:.3g}')
print(f'largest |x_1 - x_2| over the run: {gap:.1e}')
counts = [run(adam(0.1, beta1=0.0), np.diag([1.0, r]), np.ones(2))
          for r in (1.0, 1e3, 1e5)]
print('steps to f < 1e-8 at curvature ratios 1, 1e3, 1e5:', *counts)
print('steps from (100, 100) at ratio 1e3:',
      run(adam(0.1, beta1=0.0), A, 100 * np.ones(2)))
```

```
t =  1: x = 0.9, 0.9;  eta/sqrt(v_hat) = 0.1, 0.0001
t = 10: x = 0.262, 0.262;  eta/sqrt(v_hat) = 0.152, 0.000152
t = 48: x = 3.91e-06, 3.91e-06;  eta/sqrt(v_hat) = 0.329, 0.000329
largest |x_1 - x_2| over the run: 5.6e-17
steps to f < 1e-8 at curvature ratios 1, 1e3, 1e5: 40 48 54
steps from (100, 100) at ratio 1e3: 4145
```

From the very first step, the effective step sizes stand in the exact ratio $1000{:}1$—the true curvature ratio recovered from gradient magnitudes alone, without ever forming a Hessian. The two coordinates agree to floating-point precision throughout, and the effective step on the flat coordinate grows from 0.1 to 0.33 as $\hat{v}_t$ adapts to the shrinking gradients. Raising the curvature ratio from 1 to 10^3 to 10^5 increases the step count only from 40 to 48 to 54: a hundred-thousandfold worsening in conditioning costs just fourteen extra steps. Moving the starting point a hundred times farther away, however, pushes the count to 4,145. The iteration count measures starting distance in units of η, and barely feels the curvature at

all.

Why is a 45° rotation fatal to all of this? The rotated matrix has diagonal entries near 500, and its steep direction $(-1, 1)/\sqrt{2}$ projects equally onto both coordinate axes. Every coordinate's gradient is therefore dominated by the steep mode, so both coordinates accumulate nearly identical scale estimates (across the runs, their $\hat{v}$ values agree to within one percent). Because a per-coordinate preconditioner assigns only one scalar to each coordinate, it cannot shrink the steep component of a coordinate's gradient without equally shrinking its flat component. With equal scales on both coordinates, the preconditioner reduces to a scalar multiple of the identity. Scaling-only Adam ($\beta_1 = 0$) thus reverts to gradient descent whose step size is dictated by the steep mode, leaving the flat mode to crawl. Adam with $\beta_1 = 0.9$ reverts to heavy ball: its momentum is close to the optimal 0.88, but its effective step $\eta/\sqrt{\hat{v}_t}$ is governed by gradient history rather than curvature, leaving even the best grid point twice as slow as tuned heavy ball.

Without momentum, the rotated run ends in oscillation. As the steep component initially shrinks, $\hat{\mathbf{v}}$ decreases and the effective step size grows until it crosses the stability limit $2/h_{\max}$ of the steep mode. At $\eta = 0.2$—the optimal step size for the aligned valley—the loss drops to about 10^{-3} around $t = 1{,}700$, at which point the steep mode begins to oscillate. That steady oscillation locks the gradient magnitude in place, driving $\hat{\mathbf{v}}_t \to \mathbf{g}_t^2$ so that the update collapses to the sign step $\eta\,\mathrm{sign}(\mathbf{g}_t)$ of length $\eta\sqrt{2}$ across the valley. The iterate is trapped forever in a two-cycle of amplitude $\eta/\sqrt{2}$ with loss $f = 250\,\eta^2 = 10$ (Exercise 5 verifies this formula across the step-size grid). Momentum averages out these alternating steps, which is why the $\beta_1 = 0.9$ runs eventually reach the target tolerance. Figure 35.19 illustrates all three behaviors on a valley shallow enough to plot clearly.

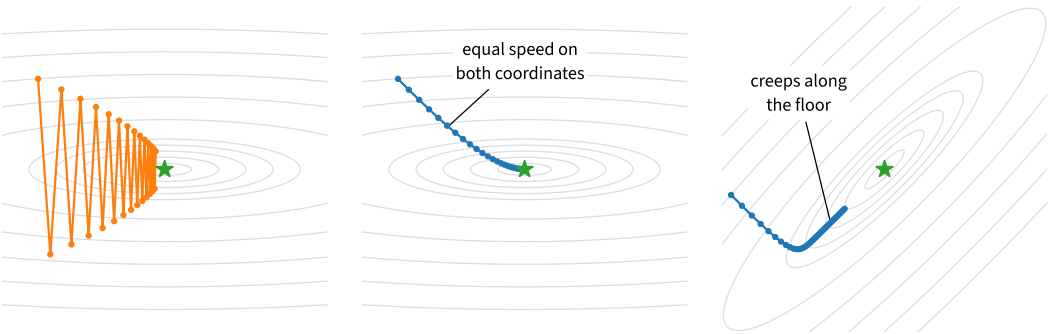


Figure 35.19: Comparison on a two-dimensional valley ($\kappa = 20$; the text's runs use $\kappa = 10^3$, whose contours are too thin to draw) from the same starting point. (a) Gradient descent with a single step size near the stability limit oscillates across the steep direction while crawling along the flat one. (b) Per-coordinate normalization (Adam with $\beta_1 = 0$) moves both coordinates at the same speed and curves smoothly into the minimum. (c) Rotating the valley and starting point by 45° misaligns the coordinate axes from the curvature directions, leaving the iterate creeping along the valley floor ($f = 0.19$ after 40 steps, versus 4×10^{-7} in (b)).

Per-coordinate scaling eliminates ill-conditioning if and only if the coordinate axes align with the eigenvectors of the curvature. Scaling changes the metric by a diagonal matrix, and no diagonal matrix can capture curvature along directions that mix coordinates. A full matrix built from outer products $\mathbf{g}_\tau \mathbf{g}_\tau^\top$ rotates with the problem and avoids this flaw, but storing and inverting it is far too expensive for large neural networks. The structured pre-

conditioners at the end of this section address this limitation for weight matrices by using compact factors that mix coordinates.

35.5.2.2 A Convex Counterexample to Adam

RMSProp and Adam forget past gradients, whereas AdaGrad remembers them indefinitely. Can that forgetting make even a simple convex problem unsolvable? Reddi et al. (2018) showed that it can, using a one-dimensional periodic sequence of convex losses.

Proposition (Adam need not converge). *Let $C > 2$ and consider online optimization over $x \in [-1, 1]$ with the periodic losses*

$$f_t(x) = \begin{cases} C\,x, & t \bmod 3 = 1, \\ -x, & \text{otherwise.} \end{cases} \tag{35.55}$$

Run Adam without bias correction, with $\beta_1 = 0$, $\beta_2 = 1/(1 + C^2)$, step sizes $\eta_t = \eta/\sqrt{t}$ satisfying $\eta < \sqrt{1 - \beta_2}$, and projection of each iterate back onto $[-1, 1]$, starting from $x_1 = 1$. Then $x_t > 0$ for every t and $x_{3k+1} = 1$ for every $k \geq 0$. Each three-step period adds at least $2C - 4 > 0$ to the regret, so the average regret never tends to zero. (Reddi et al. 2018)

Here, projection onto $[-1, 1]$ is simply clipping (Section 35.6 treats projection in general). Because the gradients over each three-step period sum to $C - 2 > 0$, the best fixed point in hindsight is $x^\star = -1$. Yet Adam returns to $x = +1$, the worst point in the interval, at the start of every period. Each period costs Adam at least $C - 2$ (it incurs C on the first step at $x = 1$, and at least -1 on each of the other two), whereas $x^\star = -1$ incurs $-(C - 2)$, producing a regret increment of at least $2C - 4$ per period. The proof proceeds by induction over periods; Exercise 3 works through the key step, where $\eta < \sqrt{1 - \beta_2}$ ensures that the downward step stays inside $(0, 1]$. Nor is this failure an artifact of setting $\beta_1 = 0$: the same paper constructs, for every fixed pair with $\beta_1 < \sqrt{\beta_2}$, both an online convex sequence and a stochastic convex problem on which Adam fails to converge.

We can see the mechanism directly in a single period with $C = 4$ and $\beta_2 = 1/17$, once v has settled into its steady periodic cycle. Following the gradients $+4, -1, -1$, the second-moment estimate v takes the values 15.12, 1.83, and 1.05, so $\sqrt{v}$ is 3.89, 1.35, and 1.02. The three resulting displacements $-\eta g_t/\sqrt{v_t}$ are

$$-1.03\,\eta, \qquad +0.74\,\eta, \qquad +0.98\,\eta,$$

yielding a net drift of $+0.69\,\eta$ per period toward $+1$. Normalization caps the rare large gradient $+C$ to a move of roughly η—barely larger than the move from a gradient of size 1—and v forgets the C^2 spike within two steps. Together, the two -1 gradients therefore push x farther upward than the single $+C$ gradient pulls it downward. By contrast, plain SGD with step $\eta g_t/C$ moves by $-\eta$, $+0.25\,\eta$, and $+0.25\,\eta$, for a net $-0.5\,\eta$ toward the optimum. Unlike the noise floor of constant-step SGD (Section 35.4), Adam's wrong-way drift does not disappear as the step size shrinks: it points toward $+1$ at every step size.

To repair this flaw, Reddi et al. (2018) proposed **AMSGrad**, which replaces $\mathbf{v}_t$ in the denominator with its running maximum $\tilde{\mathbf{v}}_t = \max(\tilde{\mathbf{v}}_{t-1}, \mathbf{v}_t)$ of the uncorrected average. The effective step $\eta_t/\sqrt{\tilde{v}_t}$ is then nonincreasing, so a large gradient is never forgotten. In our periodic example, $\tilde{v}$ remains pinned at 15.12 throughout, changing the three displacements

to $-1.03\,\eta$, $+0.26\,\eta$, and $+0.26\,\eta$—a net $-0.51\,\eta$ toward the optimum. Adam should therefore converge to $+1$, while AMSGrad and SGD converge to -1. The cell below tests all three with the same $1/\sqrt{t}$ decay starting from $x = 0$, the midpoint of the interval, so that Adam is not even initialized at the bad endpoint.

```
def online(update, C=4.0, T=15000, alpha=0.5):
    """Projected iterates on the periodic losses; beta_1 = 0, no bias correction."""
    beta2 = 1 / (1 + C * C)
    x, v, vmax, xs = 0.0, 0.0, 0.0, []
    for t in range(1, T + 1):
        g = C if t % 3 == 1 else -1.0           # rare +C, frequent -1
        v = beta2 * v + (1 - beta2) * g * g
        vmax = max(vmax, v)                     # running maximum (AMSGrad)
        x = min(1.0, max(-1.0, x - alpha / np.sqrt(t) * update(g, v, vmax, C)))
        xs.append(x)
    return xs

updates = {'adam': lambda g, v, vmax, C: g / (np.sqrt(v) + 1e-8),
           'amsgrad': lambda g, v, vmax, C: g / (np.sqrt(vmax) + 1e-8),
           'sgd': lambda g, v, vmax, C: g / C}
print('x_t at t =        30      300     3000    15000   (x* = -1)')
for name, update in updates.items():
    xs = online(update)
    print(f'{name:9s}' + ''.join(f'{xs[t - 1]:9.3f}' for t in (30, 300, 3000, 15000)))
```

```
x_t at t =        30      300     3000    15000   (x* = -1)
adam          0.745    1.000    1.000    1.000
amsgrad      -0.953   -0.985   -0.995   -0.998
sgd          -0.954   -0.986   -0.995   -0.998
```

Here `alpha` plays the role of η in the proposition, and $\eta = 0.5 < \sqrt{16/17}$, so the step-size hypothesis holds. Even from the midpoint of the interval, Adam still marches straight to $+1$: from $t \approx 300$ onward it returns to $+1$ at every period and never drops below 0.97. Meanwhile, AMSGrad and SGD steadily approach the optimum -1 and agree to two decimal places. This counterexample proves that Adam lacks a general convex convergence guarantee, though it should be read as an existence result rather than a prediction of failure on typical deep learning problems.

35.5.3 Decoupled Weight Decay

35.5.3.1 Coupled and Decoupled Decay

Almost every modern training recipe includes weight decay, which forces an adaptive optimizer to make a design choice: should the decay term pass through the preconditioner? Under SGD, the distinction does not exist. Adding an ℓ_2 penalty $\frac{\lambda}{2}\|\mathbf{w}\|^2$ to the loss adds $\lambda\mathbf{w}$ to the gradient, and the resulting update $\mathbf{w}_t - \eta\,(\mathbf{g}_t + \lambda\mathbf{w}_t) = (1-\eta\lambda)\,\mathbf{w}_t - \eta\,\mathbf{g}_t$ is simultaneously a gradient step on the loss and a uniform shrinkage of the weights (Section 9.5.3, which writes $\mathbf{x}$ for the weights). Inside Adam, however, those two mechanisms diverge. Feeding the penalized gradient into Equation 35.53 and isolating the decay term (setting $\beta_1 = 0$ for clarity, so the update is $-\eta\,(\mathbf{g}_t + \lambda\mathbf{w}_t)/(\sqrt{\widehat{\mathbf{v}}_t} + \epsilon)$), the per-step shrinkage of coordinate i becomes

$$\text{coupled } (\ell_2 \text{ through Adam}): \quad \frac{\eta\,\lambda}{\sqrt{\widehat{v}_{t,i}}+\epsilon}\,w_{t,i} \qquad \text{versus} \qquad \text{decoupled:} \quad \eta\,\lambda\,w_{t,i}. \tag{35.56}$$

In coupled Adam, the preconditioner divides both the loss gradient and the penalty gradient, so a coordinate's shrinkage rate is inversely proportional to its gradient scale: weights with large gradients are barely decayed, while quiet weights are decayed heavily. **AdamW** (Loshchilov and Hutter 2019) decouples the two operations so that only the loss gradient passes through the preconditioner while weight decay acts directly on the parameters:

$$\mathbf{w}_{t+1} = (1-\eta\lambda)\,\mathbf{w}_t - \eta\,\frac{\widehat{\mathbf{m}}_t}{\sqrt{\widehat{\mathbf{v}}_t}+\epsilon}. \tag{35.57}$$

Every coordinate now shrinks by the same factor $1-\eta\lambda$ per step, just as in SGD, and the moment averages track only the loss gradient.

`torch.optim.Adam(weight_decay=...)` adds $\lambda\mathbf{w}$ to the gradient: it implements the coupled rule. `torch.optim.AdamW` implements the decoupled rule Equation 35.57.

The sharpest test of Equation 35.56 is a weight whose loss gradient is pure zero-mean noise, leaving weight decay as the only systematic drift. If the noise has standard deviation σ, then $\sqrt{\widehat{v}} \approx \sigma$, so we expect coupled Adam to shrink the weight to roughly $e^{-\eta\lambda T/\sigma}$ after T steps, whereas AdamW should shrink every weight to $(1-\eta\lambda)^T$ regardless of σ. Below, we initialize 10^4 weights with noise scale $\sigma = 10$ and 10^4 with $\sigma = 0.1$, all at $w = 1$, run 4,000 steps of each method with $\eta = 10^{-3}$ and $\lambda = 0.1$, and compare the group means with these predictions.

```
sigma, eta, lam, T, n = np.array([[10.0], [0.1]]), 1e-3, 0.1, 4000, 10_000
rng = np.random.default_rng()

def decay(decoupled, beta1=0.9, beta2=0.999, eps=1e-8):
    """Mean of w over n weights per noise scale after T Adam steps from w = 1."""
    w, m, v = np.ones((2, n)), np.zeros((2, n)), np.zeros((2, n))
    for t in range(1, T + 1):
        g = sigma * rng.standard_normal((2, n))     # loss gradient: pure noise
        if decoupled:
            w = (1 - eta * lam) * w                 # decay outside the preconditioner
        else:
            g = g + lam * w                         # l2 penalty inside the gradient
        m = beta1 * m + (1 - beta1) * g
        v = beta2 * v + (1 - beta2) * g * g
        w = w - eta * (m / (1 - beta1**t)) / (np.sqrt(v / (1 - beta2**t)) + eps)
    return w.mean(axis=1)

print('per-step decay rates: coupled', eta * lam / sigma.ravel(), ' AdamW', eta * lam)
print('mean w, coupled:', decay(False).round(3),
      '  predictions exp(-eta lam T / sigma):', np.exp(-eta * lam * T /
      ↪ sigma.ravel()).round(3))
print('mean w, AdamW:  ', decay(True).round(3),
      '  prediction (1 - eta lam)^T:', round((1 - eta * lam)**T, 3))
```

```
per-step decay rates: coupled [1.e-05 1.e-03]  AdamW 0.0001
mean w, coupled: [0.961 0.027]   predictions exp(-eta lam T / sigma): [0.961 0.018]
mean w, AdamW:   [0.67 0.67]   prediction (1 - eta lam)^T: 0.67
```

A single value of λ gives the two groups coupled decay rates of 10^{-5} and 10^{-3} per step—a hundredfold difference. The noisy weights retain 0.96 of their initial magnitude as predicted, while the quiet weights drop to about 0.03, slightly above the 0.018 estimate. That small shortfall has a straightforward cause: near $w = 1$, the penalty gradient $\lambda w = 0.1$ on a quiet weight is comparable to its noise and inflates its own $\hat{v}$, slowing its initial decay. Under AdamW, by contrast, both groups finish right at $0.67 = (1 - \eta\lambda)^{4000}$. For AdamW this prediction is exact in expectation, because the normalized noise steps have mean zero and the multiplicative decay is identical at every step.

35.5.3.2 Fixed Points

The pure-noise experiment has no stationary point because its gradients never settle. With exact gradients, however, an optimizer can come to rest, and at a fixed point λ takes on a surprisingly different meaning.

Proposition (fixed points of coupled and decoupled decay). *Let the gradient $\mathbf{g}(\mathbf{w}) = \nabla\ell(\mathbf{w})$ of a loss ℓ be exact (noiseless), let $\epsilon > 0$, and call $\mathbf{w}$ a fixed point of an update if the update vanishes there once the moment estimates have settled. (a) Adam on $\ell + \frac{\lambda}{2}\|\mathbf{w}\|^2$ has a fixed point at $\mathbf{w}$ if and only if $\nabla\ell(\mathbf{w}) + \lambda\mathbf{w} = \mathbf{0}$, the same stationary point as SGD on the penalized loss. (b) AdamW has a fixed point at $\mathbf{w}$ if and only if, for every coordinate i,*

$$\lambda\, w_i(|g_i| + \epsilon) = -g_i, \qquad g_i = \partial_i\ell(\mathbf{w}). \tag{35.58}$$

Every fixed point of AdamW therefore lies strictly inside the box $|w_i| < 1/\lambda$, and as $\epsilon \to 0$ the condition reduces to a dichotomy: either $g_i = 0$, or $|w_i| = 1/\lambda$ with w_i opposite in sign to g_i.

Proof. At a fixed point, the vector $\mathbf{u}$ fed into the moment averages is constant, so $\widehat{\mathbf{m}}_t \to \mathbf{u}$ and $\widehat{\mathbf{v}}_t \to \mathbf{u}^2$. (a) For coupled Adam, $\mathbf{u} = \mathbf{g} + \lambda\mathbf{w}$, and coordinate i's update approaches $-\eta\, u_i/(|u_i| + \epsilon)$, which vanishes if and only if $u_i = 0$. (b) For AdamW, $\mathbf{u} = \mathbf{g}$, so coordinate i's update approaches $-\eta\lambda w_i - \eta\, g_i/(|g_i| + \epsilon)$; setting this to zero and multiplying through by $(|g_i| + \epsilon)/\eta$ yields Equation 35.58. Taking absolute values gives $|w_i| = |g_i|/(\lambda(|g_i| + \epsilon)) < 1/\lambda$. Letting $\epsilon \to 0$ turns the condition into $\lambda w_i|g_i| = -g_i$, which holds precisely when $g_i = 0$ or $\lambda w_i = -\text{sign}(g_i)$. ■

This proposition characterizes fixed points; it does not assert that one exists or that the iterates converge to it. Nor is the dichotomy merely a limiting curiosity: since $1 - \lambda|w_i| = \epsilon/(|g_i| + \epsilon)$, at $\epsilon = 10^{-8}$ every coordinate either sits within a fraction $\epsilon/|g_i|$ of the box boundary or has a gradient of order ϵ.

A one-dimensional example makes the contrast concrete. Consider $\ell(w) = \frac{1}{2}(w - 3)^2$, whose unconstrained minimizer is $w = 3$. With $\lambda = \frac{1}{4}$, coupled Adam rests at the penalized minimizer $3/(1 + \lambda) = 2.4$, whereas AdamW rests at 3.0: the box $|w| \le 4$ does not bind, so decoupled decay leaves the minimizer untouched. With $\lambda = 1$, coupled Adam rests at 1.5, while AdamW is pinned at 1.0, right on the boundary of the box.

The two interpretations of λ are thus completely distinct. In the noise-dominated regime of training, AdamW applies uniform per-step shrinkage across all coordinates while coupled Adam distorts it by gradient scale. At exact-gradient fixed points, the roles flip: cou-

pled Adam preserves the maximum a posteriori (MAP) solution of a Gaussian prior on the weights (Section 36.6), whereas AdamW turns λ into the radius $1/\lambda$ of an ℓ_∞ constraint box (Xie and Li 2024). Exercise 6 connects this box directly to constrained optimization.

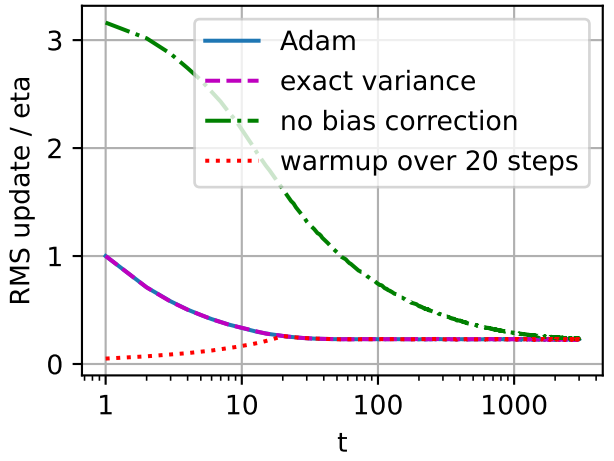
35.5.4 Warmup

At step $t = 1$, Adam's bias-corrected moments are $\hat{\mathbf{m}}_1 = \mathbf{g}_1$ and $\hat{\mathbf{v}}_1 = \mathbf{g}_1^2$, so the update is simply $\eta \operatorname{sign}(\mathbf{g}_1)$—a step of full size η in every coordinate. A coordinate whose gradient is 10^{-6} moves just as far as one whose gradient is 10^3, on the evidence of a single minibatch. **Warmup** ramps the step size up from near zero over the opening steps of training (Goyal et al. 2017), keeping these early updates small. It is tempting to blame this early instability on the preconditioner $\hat{\mathbf{v}}_t$, which at $t = 1$ rests on a single sample. A simple experiment shows that something else is at work.

Is a step of size η actually too large? Not for a coordinate with a consistent gradient, whose steady-state step is also η. For a coordinate whose gradient is pure noise, however, it is wasted motion. To measure how large that wasted motion is, we draw 2×10^4 independent gradient coordinates from $\mathcal{N}(0, 1)$ at each step, run default Adam, and track the root-mean-square update in units of η. We compare four curves: standard Adam; Adam with $\hat{\mathbf{v}}_t$ replaced by the exact variance $\sigma^2 = 1$ (eliminating preconditioner estimation error entirely); Adam with both bias corrections removed; and Adam multiplied by a linear warmup over the first 20 steps.

```
n, T, beta1, beta2, eps, T_w = 20_000, 3000, 0.9, 0.999, 1e-8, 20
m, v = np.zeros(n), np.zeros(n)
rms = {'Adam': [], 'exact variance': [], 'no bias correction': [],
       f'warmup over {T_w} steps': []}
for t in range(1, T + 1):
    g = rng.standard_normal(n)                      # pure noise, sigma = 1
    m = beta1 * m + (1 - beta1) * g
    v = beta2 * v + (1 - beta2) * g * g
    m_hat, v_hat = m / (1 - beta1**t), v / (1 - beta2**t)
    step = m_hat / (np.sqrt(v_hat) + eps)          # Adam's update / eta
    for r, u in zip(rms.values(), (step, m_hat, m / (np.sqrt(v) + eps),
                                   min(1.0, t / T_w) * step)):
        r.append(np.sqrt(np.mean(u**2)))
d2l.plot(np.arange(1, T + 1), list(rms.values()), 't', 'RMS update / eta',
         legend=list(rms), xscale='log')
ts = (1, 2, 3, 10, 20, 30, 100, 1000, 3000)
print(f'{"t:":24s}' + ''.join(f'{t:>6d}' for t in ts))
for name, r in rms.items():
    print(f'{name + ":":24s}' + ''.join(f'{r[t - 1]:6.2f}' for t in ts))
```

t:	1	2	3	10	20	30	100	1000	3000
Adam:	1.00	0.71	0.58	0.33	0.26	0.24	0.23	0.23	0.23
exact variance:	1.00	0.71	0.58	0.33	0.26	0.24	0.23	0.23	0.23
no bias correction:	3.16	3.02	2.87	2.17	1.61	1.32	0.74	0.29	0.24
warmup over 20 steps:	0.05	0.07	0.09	0.17	0.26	0.24	0.23	0.23	0.23



Standard Adam starts with an RMS update of $1.00\,\eta$, drops to $0.33\,\eta$ by $t = 10$, and settles within about fifty steps at $\sqrt{(1-\beta_1)/(1+\beta_1)}\,\eta = 0.23\,\eta$—the steady-state size of an exponential moving average of independent unit noise. On a pure noise coordinate, Adam's opening steps are therefore roughly four times larger than their steady-state size, and this transient lasts about ten steps.

Is the noisy preconditioner to blame? The second curve says no. Replacing $\hat{\mathbf{v}}_t$ with the exact variance leaves the RMS update unchanged to within 0.01 at every step (the sampling noise of 2×10^4 coordinates). The fourfold inflation comes from the *momentum* average, which needs roughly $1/(1-\beta_1) = 10$ steps to average out independent noise. What the preconditioner actually does at $t = 1$ is equalize coordinates of different scales, an effect hidden when every coordinate shares $\sigma = 1$.

Bias correction matters far more. Without it, the opening update jumps to $3.16\,\eta$—matching the $(1-\beta_1)/\sqrt{1-\beta_2}$ factor in Figure 35.18—and the transient stretches from ten steps to hundreds: the update is still around $0.75\,\eta$ at $t = 100$ and $0.29\,\eta$ at $t = 1000$, because the uncorrected $\mathbf{v}_t$ fills on the slow $1/(1-\beta_2) = 1000$ timescale. An uncorrected optimizer would need a thousand-step warmup merely to undo its own zero initialization. With bias correction in place, a linear ramp over just 20 steps tames what remains, keeping the warmup curve below $0.26\,\eta$ throughout.

That accounts for warmups of tens of steps, not the thousands of steps often used in large-scale training. Longer warmups respond to a second mechanism absent from this noise model: curvature. A step size is stable only below the ceiling $2/\lambda_{\max}$ imposed by the steepest curvature direction (Section 35.1.2). Immediately after random initialization, that curvature can be unusually large and shift rapidly, so a step size that is safe later in training may initially be unstable. Section 9.7.3 revisits both mechanisms on a real neural network and measures the opening step sizes directly.

35.5.5 Structured Preconditioners

35.5.5.1 Kronecker Factors

Our rotated valley defeated every diagonal preconditioner. All of the methods in this section share the preconditioned form $\mathbf{x}_{t+1} = \mathbf{x}_t - \eta\,\mathbf{P}_t^{-1}\mathbf{g}_t$ for a positive definite matrix $\mathbf{P}_t$: gradient

descent uses $\mathbf{P} = \mathbf{I}$, Newton's method uses the Hessian $\nabla^2 f$, and Adam uses a diagonal matrix estimated from past gradients. A full curvature matrix captures steepness along any direction—aligned with the axes or not—but it is far too large to store. A single 4096×4096 weight matrix already has 1.7×10^7 parameters; its full curvature matrix would contain 2.8×10^{14} entries, requiring roughly two petabytes in double precision.

What makes a middle ground possible is that neural network parameters are organized into matrices. Consider a linear layer $\mathbf{y} = \mathbf{W}\mathbf{a}$ with weight matrix $\mathbf{W} \in \mathbb{R}^{m \times n}$, input $\mathbf{a} \in \mathbb{R}^n$, and output gradient $\delta = \partial \ell / \partial \mathbf{y} \in \mathbb{R}^m$. The gradient with respect to $\mathbf{W}$ is the rank-one outer product $\delta \mathbf{a}^\top$. Writing $\mathrm{vec}(\mathbf{V})$ for the vector obtained by stacking the columns of a matrix $\mathbf{V}$, we have $\mathrm{vec}(\delta \mathbf{a}^\top) = \mathbf{a} \otimes \delta$, where the **Kronecker product** $\mathbf{B} \otimes \mathbf{D}$ is the block matrix whose (i, j) block is $b_{ij}\mathbf{D}$.

The outer-product curvature of this layer—the block of the Fisher information matrix $\mathbb{E}[\nabla \log p \, \nabla \log p^\top]$ (which we will study in Section 36.6) corresponding to $\mathbf{W}$—is therefore $\mathbb{E}\big[(\mathbf{a}\mathbf{a}^\top) \otimes (\delta\delta^\top)\big]$. **K-FAC** (Martens and Grosse 2015) approximates this expectation by the Kronecker product $\mathbf{S}_{\text{in}} \otimes \mathbf{S}_{\text{out}}$, where $\mathbf{S}_{\text{in}} = \mathbb{E}[\mathbf{a}\mathbf{a}^\top]$ is $n \times n$ and $\mathbf{S}_{\text{out}} = \mathbb{E}[\delta\delta^\top]$ is $m \times m$. This factorization would be exact if inputs and output gradients were statistically independent; nobody expects that to hold in a trained network, but the approximation is adopted because it can be inverted cheaply.

Specifically, the inverse of a Kronecker product acts directly on the unstacked gradient matrix. For symmetric invertible factors,

$$\big(\mathbf{S}_{\text{in}} \otimes \mathbf{S}_{\text{out}}\big)^{-1} \mathrm{vec}(\mathbf{V}) = \mathrm{vec}\big(\mathbf{S}_{\text{out}}^{-1} \mathbf{V} \, \mathbf{S}_{\text{in}}^{-1}\big). \tag{35.59}$$

To verify this identity, write column j of the product $\mathbf{DVB}$ as $\sum_k b_{kj} \, \mathbf{D}\mathbf{v}_k$, where $\mathbf{v}_k$ is column k of $\mathbf{V}$. Stacking these columns produces the block product $(\mathbf{B}^\top \otimes \mathbf{D})\,\mathrm{vec}(\mathbf{V})$, so $\mathrm{vec}(\mathbf{DVB}) = (\mathbf{B}^\top \otimes \mathbf{D})\,\mathrm{vec}(\mathbf{V})$. Using $(\mathbf{B} \otimes \mathbf{D})^{-1} = \mathbf{B}^{-1} \otimes \mathbf{D}^{-1}$ together with the symmetry of the factors, setting $\mathbf{B} = \mathbf{S}_{\text{in}}^{-1}$ and $\mathbf{D} = \mathbf{S}_{\text{out}}^{-1}$ yields Equation 35.59. In this way, a curvature matrix with $(mn)^2$ entries is inverted and applied through two modest inverses of sizes n and m. The cell below compares the storage and operation counts for our 4096×4096 layer.

```
m = n = 4096
print(f'stored entries: full {(m * n)**2:.1e}, two factors {m**2 + n**2:.1e}')
print(f'flops to invert: full {(m * n)**3:.1e}, two factors {m**3 + n**3:.1e}')
```

```
stored entries: full 2.8e+14, two factors 3.4e+07
flops to invert: full 4.7e+21, two factors 1.4e+11
```

Factoring cuts storage from 2.8×10^{14} entries down to 3.4×10^7, and cuts the inversion cost from 4.7×10^{21} operations to roughly 1.4×10^{11}.

35.5.5.2 K-FAC, Shampoo and Muon

The Kronecker identity leaves two choices open: which matrices to factor, and what exponent to apply to them. Modern structured optimizers share the two-sided preconditioning form $\mathbf{G} \mapsto \mathbf{P}_{\text{l}}^{-a} \mathbf{G} \, \mathbf{P}_{\text{r}}^{-a}$ applied to a layer's gradient matrix $\mathbf{G}$ (bold, distinguishing it from the scalar gradient bound G of the regret analysis). Together, they form a natural progression from Adam's diagonal scaling to matrix-level normalization.

K-FAC constructs its factors from the second moments of layer inputs and output gradients with exponent $a = 1$. It approximates preconditioning by the Fisher information matrix—the **natural gradient** (Amari 1998), which is the metric view of AdaGrad using the Fisher matrix as the metric. For softmax cross-entropy and squared loss, the Fisher matrix (with labels sampled from the model) coincides with the Gauss–Newton matrix, a positive semidefinite approximation to the Hessian that drops the term involving second derivatives of the network. For these losses, K-FAC therefore approximates genuine curvature.

Shampoo (Gupta et al. 2018) instead builds its left and right factors directly from accumulated gradient matrices in the spirit of AdaGrad, setting $\mathbf{P}_{\mathrm{l}} = \sum_t \mathbf{G}_t \mathbf{G}_t^\top$ and $\mathbf{P}_{\mathrm{r}} = \sum_t \mathbf{G}_t^\top \mathbf{G}_t$ with $a = \frac{1}{4}$. **Muon** (Jordan et al. 2024) can be viewed as memoryless Shampoo: using only the current matrix $\mathbf{G} = \mathbf{U}\Sigma\mathbf{V}^\top$ (assumed square and invertible),

$$\left(\mathbf{G}\mathbf{G}^\top\right)^{-1/4} \mathbf{G} \left(\mathbf{G}^\top\mathbf{G}\right)^{-1/4} = \mathbf{U}\Sigma^{-1/2}\,\Sigma\,\Sigma^{-1/2}\mathbf{V}^\top = \mathbf{U}\mathbf{V}^\top,$$

which is the polar factor of $\mathbf{G}$. It preserves the singular vectors of $\mathbf{G}$ while setting every singular value to one. Where Adam equalizes the coordinates of the update, Muon equalizes its singular directions—and unlike per-coordinate normalization, this map commutes with orthogonal rotations of the rows and columns (Exercise 5d). Muon as a practical optimizer, which applies this orthogonalization to the momentum matrix, is covered in the main text (Section 9.6), while the Newton–Schulz iteration that computes the polar factor using matrix multiplications alone is analyzed in Section 33.5.4.2.

None of these three methods reproduces full Newton's method. Muon stores no second-moment matrices and requires only a handful of matrix products per step, whereas K-FAC and Shampoo invert or take matrix roots of their two factors, typically amortizing that cost over several steps. Moreover, every positive result in this section assumes that curvature aligns either with the coordinate axes or with the Kronecker structure of individual layers. Because the true curvature of a deep neural network is neither diagonal nor fixed, explaining why these adaptive methods perform as well as they do in practice remains an active open question.

35.5.6 Summary

Dividing each coordinate's update by the root mean square of its past gradients, the common thread linking AdaGrad, RMSProp, and Adam, removes the condition number whenever curvature aligns with the coordinate axes. With $\epsilon = 0$, Adam's iterates are strictly invariant to per-coordinate rescaling, allowing scaling alone to beat an optimally tuned heavy-ball method by an order of magnitude on an aligned valley. Rotate the valley by 45°, however, and that advantage vanishes: scaling alone never converges, and Adam with momentum takes roughly twice as many steps as heavy ball.

Forgetting past gradients introduces further subtleties. On a periodic sequence of convex losses, Adam's short second-moment memory can drive it to the worst point of the domain, a failure fixed by AMSGrad's running maximum. With weight decay, passing an ℓ_2 penalty through Adam scales each coordinate's shrinkage inversely with its gradient size while preserving the MAP fixed point; AdamW decouples the decay so that every coordinate shrinks uniformly per step, turning λ into the radius $1/\lambda$ of an ℓ_∞ box at exact-gradient fixed points.

Finally, Adam's opening updates on pure-noise coordinates are roughly four times larger than their steady-state size because of the momentum average rather than the precondi-

tioner, and dropping bias correction inflates them much further for hundreds of steps; longer practical warmups address early curvature instead. Beyond diagonal scaling, Kronecker-factored preconditioners such as K-FAC, Shampoo, and Muon capture cross-coordinate structure within weight matrices using two compact factors, with Muon equalizing the singular directions of a layer's update.

35.5.7 Exercises

1. **AdaGrad from the metric view.** Exercise 1b of Section 35.1 showed that the steepest descent direction under the norm $\|\mathbf{d}\|_{\mathbf{P}} = \sqrt{\mathbf{d}^\top \mathbf{P}\mathbf{d}}$ is proportional to $-\mathbf{P}^{-1}\nabla f$.
   1. Set $\mathbf{P}_t = \operatorname{diag}\big(\sqrt{\mathbf{s}_t}\big) + \epsilon\mathbf{I}$ with $\mathbf{s}_t = \sum_{\tau \le t} \mathbf{g}_\tau^2$ and recover Equation 35.51.
   2. Replace the objective f by cf for a constant $c > 0$ and set $\epsilon = 0$. How does the AdaGrad update change? How would it change if $\mathbf{P}_t$ were built from $\mathbf{s}_t$ instead of $\sqrt{\mathbf{s}_t}$? Compare both with gradient descent and with Newton's method.
   3. Let the gradients of one coordinate be i.i.d. with $\mathbb{E}[g_{t,i}^2] = \sigma^2$. How fast does the effective step of that coordinate decay in t under each of the two choices, and how does it depend on σ?
2. **Concentrated gradients and AdaGrad's bound.** Up to constant factors, projected online gradient descent with a tuned global step has regret $D_2 G\sqrt{T}$ on a convex set of Euclidean diameter D_2 when $\|\mathbf{g}_t\| \le G$ (Exercise 4). Diagonal AdaGrad has regret $D_\infty \sum_{i=1}^{d} \sqrt{\sum_{t=1}^{T} g_{t,i}^2}$, where D_∞ is the diameter of the set in the ℓ_∞ norm (Duchi et al. 2011). Take the box $[-1, 1]^d$ and one-hot gradients: at each step one coordinate i, drawn independently with probability p_i, receives the gradient $+1$ or -1, and all other coordinates receive 0.
   1. Compute both bounds. For AdaGrad, bound the expectation of each square root with Jensen's inequality, and show that the ratio of the AdaGrad bound to the gradient-descent bound is at most $\sum_i \sqrt{p_i}/\sqrt{d}$.
   2. Evaluate the ratio for uniform p_i and for Zipf frequencies $p_i \propto i^{-\alpha}$ with $d = 10^4$ and $\alpha \in \{1, 2, 3\}$.
   3. Word frequencies in natural text follow Zipf's law with $\alpha \approx 1$, and your ratio there is about 0.63. How much of the factor $\sqrt{d}$ of Section 35.5.1 does a bag-of-words model obtain? Which distribution of gradient mass would obtain all of it?
3. **The Reddi example by hand.** Take $C = 4$, $\beta_1 = 0$, the proposition's $\beta_2 = 1/(1 + C^2) = 1/17$, and no bias correction, as in `online`.
   1. Assume a fixed step η, an iterate in the interior of $[-1, 1]$, and v_t settled into its periodic pattern. Write v_a for the value of v after the $+C$ step, solve the three-step recursion for v_a in closed form, and derive the periodic values 15.12, 1.83, 1.05 and the three displacements of the text. Compute SGD's three steps $\eta g_t/C$. Which steps does Adam enlarge relative to SGD, and why?
   2. AMSGrad replaces v_t by $\tilde{v}_t = \max(\tilde{v}_{t-1}, v_t)$. Show that its effective step $\eta_t/\sqrt{\tilde{v}_t}$ is nonincreasing in t whenever η_t is. Compute its three displacements in the periodic regime and show that the net displacement is $-0.51\,\eta$.
   3. The proposition needs $\eta < \sqrt{1 - \beta_2}$ for the steps $\eta_t = \eta/\sqrt{t}$. Show where this is used: starting a period at $x = 1$, use $v \ge (1 - \beta_2)C^2$ after the $+C$ gradient to show that the first step of the period lands in $(0, 1)$, so that the projection does not act on it.
4. **Regret of online gradient descent.** ● Projected online gradient descent plays $\mathbf{x}_{t+1} =$

$\Pi_{\mathcal{C}}(\mathbf{x}_t - \eta_t \mathbf{g}_t)$ with $\mathbf{g}_t = \nabla f_t(\mathbf{x}_t)$, where $\Pi_{\mathcal{C}}$ is the Euclidean projection onto a closed convex set $\mathcal{C}$ of diameter D, $\|\mathbf{g}_t\| \le G$, and the steps η_t are nonincreasing (Zinkevich 2003).

1. Prove that the regret satisfies

$$\sum_{t=1}^{T} f_t(\mathbf{x}_t) - \min_{\mathbf{x}\in\mathcal{C}} \sum_{t=1}^{T} f_t(\mathbf{x}) \;\le\; \frac{D^2}{2\eta_T} + \frac{G^2}{2}\sum_{t=1}^{T} \eta_t. \tag{35.60}$$

   Start from $f_t(\mathbf{x}_t) - f_t(\mathbf{x}) \le \mathbf{g}_t^{\top}(\mathbf{x}_t - \mathbf{x})$ and use that the projection does not increase distances to points of $\mathcal{C}$.
2. Show that $\eta_t = D/(G\sqrt{t})$ gives regret at most $\frac{3}{2} DG\sqrt{T}$, and that for $\eta_t = c/t$ the bound grows linearly in T for every $c > 0$.
3. Combine these two bounds with the decay rates of Exercise 1c to explain why AdaGrad divides by $\sqrt{\mathbf{s}_t}$ rather than by $\mathbf{s}_t$.

5. code **Rotation sensitivity.** ● Use `rotated`, `run` and the update factories of the `#adaptive-stochastic-methods-adam-vs-gd` cell.
   1. Show that the iterates of gradient descent and of heavy ball on $\mathbf{A}_\theta$ from $\mathbf{R}_\theta(1,1)^{\top}$ are $\mathbf{R}_\theta$ times their iterates on the aligned valley, so that their step counts do not depend on θ.
   2. For $\beta_1 = 0$ at $\theta = \pi/4$, print f after $40{,}000$ steps at each η of the table's grid and compare it with the value $250\,\eta^2$ of the text. Which component of Adam supplies what survives of its advantage over gradient descent?
   3. Print Adam's effective per-coordinate steps at $\theta = \pi/16$. Why does even this small rotation remove most of the step-size ratio that Adam builds on the aligned valley?
   4. Exercise 4 of Section 35.1 identifies $-\operatorname{sign}(\mathbf{g})$ as the steepest direction under the ℓ_∞ norm and $-\mathbf{U}\mathbf{V}^{\top}$ as the steepest direction under the spectral norm, where $\mathbf{G} = \mathbf{U}\Sigma\mathbf{V}^{\top}$. Show that Adam with $\beta_1 = \beta_2 = 0$ and $\epsilon = 0$ steps along $-\operatorname{sign}(\mathbf{g}_t)$. Then let $\mathbf{Q}_1$ and $\mathbf{Q}_2$ be orthogonal, show that the map $\mathbf{G} \mapsto \mathbf{U}\mathbf{V}^{\top}$ commutes with $\mathbf{G} \mapsto \mathbf{Q}_1\mathbf{G}\mathbf{Q}_2$, and give a 2×2 example in which the sign map does not. What does this suggest about the rotation sensitivity of Muon compared with Adam?

6. [extended] **AdamW's implicit ℓ_∞ constraint.** ●● The proposition of Section 35.5.3 found that the fixed points of AdamW lie in the box $|w_i| < 1/\lambda$ and, as $\epsilon \to 0$, on its boundary wherever the gradient does not vanish. Xie and Li (2024) prove that if full-batch AdamW with $\beta_1 \le \beta_2$ and a nonincreasing step schedule with divergent sum converges, its limit is a KKT point of $\min_{\mathbf{w}} \ell(\mathbf{w})$ subject to $\|\mathbf{w}\|_\infty \le 1/\lambda$.
   1. A feasible $\mathbf{w}$ is a KKT point of this problem (Section 35.6 derives the conditions in general) when every coordinate satisfies $\partial_i \ell(\mathbf{w}) = 0$ if $|w_i| < 1/\lambda$ and $w_i\,\partial_i \ell(\mathbf{w}) \le 0$ if $|w_i| = 1/\lambda$. Show that at such a point no feasible direction decreases ℓ to first order.
   2. Set $\epsilon = 0$, the setting of the theorem, in the AdamW condition Equation 35.58. Show that a coordinate with $\partial_i \ell \neq 0$ satisfies the KKT conditions. Why does the condition alone not show $|w_i| \le 1/\lambda$ for a coordinate whose gradient vanishes?
   3. Generate linearly separable logistic-regression data with $n = 100$, $d = 5$, standard normal features, and labels $y_i = \operatorname{sign}(\mathbf{x}_i^{\top}\mathbf{w}_{\text{true}})$, so that the unregularized loss has no finite minimizer. Run full-batch AdamW with $\epsilon = 10^{-16}$ for 5×10^4 steps, with η decaying from 10^{-2} to 0 by a cosine schedule, for $\lambda \in \{0.05, 0.1, 0.2\}$. Compare the result with the solution of the box-constrained problem computed by projected gradient descent, which here clips every coordinate to $[-1/\lambda, 1/\lambda]$ after each gradient step, and with Adam on $\ell + \frac{\lambda}{2}\|\mathbf{w}\|^2$.

4. Repeat the AdamW runs with minibatch gradients of size 10 and 50. Where does the iterate end up relative to the box, and why?
5. What does λ control in AdamW according to these results, and how does that differ from the Gaussian-prior reading of λ in Section 36.6?

7. **The Kronecker identity and its cost.** Use column stacking for vec.
   1. Prove $\mathrm{vec}(\mathbf{DVB}) = (\mathbf{B}^\top \otimes \mathbf{D})\,\mathrm{vec}(\mathbf{V})$ by writing out column j of the product, and prove $(\mathbf{B} \otimes \mathbf{D})(\mathbf{B}' \otimes \mathbf{D}') = (\mathbf{BB}') \otimes (\mathbf{DD}')$.
   2. Deduce Equation 35.59 for symmetric invertible factors. Which formula replaces it for factors that are not symmetric, and which for row stacking?
   3. For an $m \times n$ layer, count the operations needed to invert the full $mn \times mn$ matrix and apply it, against inverting the two factors and applying them by Equation 35.59 ($O(m^3 + n^3 + mn(m+n))$). Evaluate both counts at $m = n = 4096$ and at $m = 4096$, $n = 16384$.
   4. For a single gradient $\mathbf{G} = \mathbf{U}\Sigma\mathbf{V}^\top$, set $\mathbf{P}_\mathrm{l} = \mathbf{GG}^\top$ and $\mathbf{P}_\mathrm{r} = \mathbf{G}^\top\mathbf{G}$. Show that the Shampoo step $\mathbf{P}_\mathrm{l}^{-1/4}\mathbf{G}\,\mathbf{P}_\mathrm{r}^{-1/4}$ equals $\mathbf{UV}^\top$. What must the inverse fourth root mean when $\mathbf{G}$ is not square, and which singular vectors does the result keep?

35.6 Constrained Optimization

Up to this point, every parameter we optimized was free to take any real value, so finding a minimum meant hunting for a point where the gradient vanishes. Many quantities in deep learning do not have that freedom. Attention weights must be nonnegative and sum to one, gradient clipping caps the norm of an update at a threshold, a norm budget restricts the size of a weight vector, and a support vector machine requires every training margin to be at least one. When the variables must stay inside a feasible set, the gradient rarely vanishes at the optimum, and the zero-gradient test of the earlier sections breaks down. How do we recognize an optimal point under constraints, and how do we compute it?

Gradient clipping offers the simplest concrete example. Suppose we want to clip the vector $\mathbf{y} = (3, 4)$ so that its norm is at most 1. Rescaling it to unit length returns $(0.6, 0.8)$, which is the point of the unit disk closest to $\mathbf{y}$. In other words, $(0.6, 0.8)$ minimizes the squared distance $f(\mathbf{x}) = \frac{1}{2}\|\mathbf{x} - \mathbf{y}\|^2$ over the disk $\|\mathbf{x}\| \le 1$, reaching the optimal value $f = 8$. Yet the gradient there is $(-2.4, -3.2)$, nowhere near zero. What makes $(0.6, 0.8)$ optimal is that no *allowed* direction lowers f: every move that keeps $\|\mathbf{x}\| \le 1$ either increases f or leaves it unchanged to first order. To handle problems like this, we need both a replacement for the zero-gradient condition and a replacement for plain gradient descent, which on its own would step right out of the feasible set.

The replacement for "the gradient is zero" introduces one scalar per constraint, called its *multiplier*, which measures how hard the optimum presses against that constraint. At $(0.6, 0.8)$ the boundary pushes back with strength 4 along the outward normal, so that $(-2.4, -3.2) + 4\,(0.6, 0.8) = \mathbf{0}$. Multipliers give us the first-order optimality conditions: Lagrange's condition for equality constraints, and the Karush–Kuhn–Tucker (KKT) conditions for inequalities. Meanwhile, the projection that mapped $(3, 4)$ to $(0.6, 0.8)$ suggests a direct algorithm: take an ordinary gradient step, then project back to the nearest feasible point. And when a hard boundary is replaced by a kinked penalty, such as the absolute-value penalty that the lasso places on each weight, the projection generalizes to a *proximal step*, which drives small weights to exact zeros. Throughout the section, convexity (Section 35.3) is what turns these conditions into guarantees of global optimality and ensures that our algorithms converge.

Three ideas carry the section: the KKT conditions and their sufficiency for convex problems, the obtuse-angle inequality that characterizes a projection, and soft-thresholding.

35.6.1 Equality Constraints and Lagrange Multipliers

35.6.1.1 Tangency and the Lagrange Condition

Equality constraints are simpler than inequalities, so let us start by replacing the solid disk of the clipping example with its boundary: minimize $f(\mathbf{x}) = \frac{1}{2}\|\mathbf{x}-\mathbf{y}\|^2$ with $\mathbf{y} = (3,4)$ over the unit circle, written as $h(\mathbf{x}) = \frac{1}{2}(\|\mathbf{x}\|^2 - 1) = 0$. More generally, we want to minimize a differentiable objective $f : \mathbb{R}^n \to \mathbb{R}$ subject to a single equality constraint,

$$\min_{\mathbf{x}} \; f(\mathbf{x}) \quad \text{subject to} \quad h(\mathbf{x}) = 0.$$

Because the feasible set is a surface (a curve in two dimensions) rather than a full-dimensional region, every feasible move must stay on that surface.

Figure 35.20 illustrates the geometric signature of an optimum. Suppose we stand at a feasible point where a level set of f *crosses* the circle. Then ∇f has a nonzero component along the circle, so sliding along the circle opposite that component reduces f without leaving the feasible set. Such a point cannot be optimal. An optimum can occur only where the level set is *tangent* to the circle, so that every move along the circle leaves f unchanged to first order. Since tangent curves share parallel normal vectors, ∇f must be a scalar multiple of ∇h at the optimum.

Proposition (Lagrange condition). *Let $\mathbf{x}^\star$ be a local minimum of f on $\{\mathbf{x} : h(\mathbf{x}) = 0\}$. Assume that f is differentiable at $\mathbf{x}^\star$, that h is continuously differentiable near $\mathbf{x}^\star$, and that $\nabla h(\mathbf{x}^\star) \neq \mathbf{0}$. Then there exists a unique scalar $\nu^\star$, the* **Lagrange multiplier***, such that*

$$\nabla f(\mathbf{x}^\star) + \nu^\star \, \nabla h(\mathbf{x}^\star) = \mathbf{0}. \tag{35.61}$$

In words, the gradient of the objective is balanced by a multiple of the constraint's normal vector. This is the same Lagrange condition proved in Section 34.2.4, where the constraint is written as $g(\mathbf{x}) = c$ and the condition as $\nabla f = \mu \nabla g$. Setting $h = g - c$ and $\nu = -\mu$ reconciles the two forms; we use the plus sign here because it matches the sign of the Lagrangian introduced next.

Checking the tangent point $(0.6, 0.8)$ in Figure 35.20, we have $\nabla f = \mathbf{x} - \mathbf{y} = (-2.4, -3.2)$ and $\nabla h = \mathbf{x} = (0.6, 0.8)$. The two vectors are indeed parallel, with $\nabla f = -4\,\nabla h$, which gives $\nu^\star = 4$.

The assumption $\nabla h(\mathbf{x}^\star) \neq \mathbf{0}$ is our first example of a **constraint qualification**, and it is essential rather than a technicality. Consider minimizing $f(\mathbf{x}) = x_1$ subject to $h(\mathbf{x}) = x_1^2 + x_2^2 = 0$. The feasible set consists of the single point $\mathbf{0}$, which is automatically the minimum. Yet $\nabla f(\mathbf{0}) = (1, 0)$ while $\nabla h(\mathbf{0}) = \mathbf{0}$, so no multiplier $\nu^\star$ can satisfy Equation 35.61. Because the feasible set has collapsed to a single point with no tangent directions, the tangency argument has nothing to grab onto.

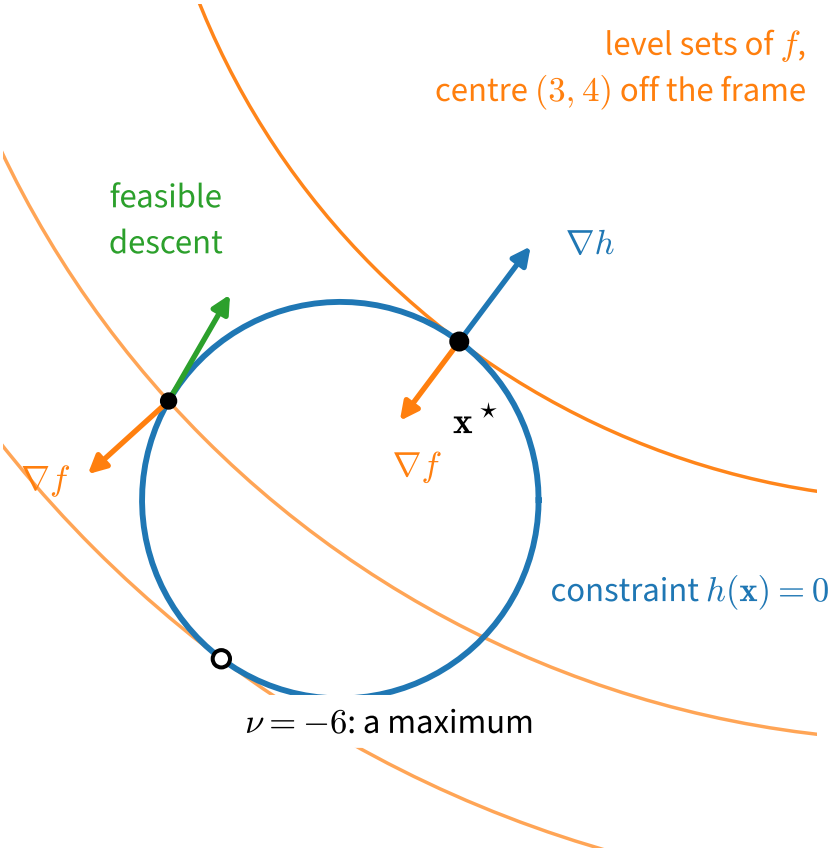


Figure 35.20: Minimizing $f = \frac{1}{2}\|\mathbf{x}-\mathbf{y}\|^2$ with $\mathbf{y} = (3, 4)$ on the unit circle $h(\mathbf{x}) = 0$. Where a level set crosses the circle, ∇f has a component along the constraint, giving a feasible descent direction. At $\mathbf{x}^\star = (0.6, 0.8)$ the level set is tangent to the circle and $\nabla f = -4\,\nabla h$. The antipode $(-0.6, -0.8)$ also satisfies the tangency condition, with $\nu = -6$, and is the maximum.

35.6.1.2 The Lagrangian

Notice that we have two unknowns, $\mathbf{x}$ and ν, and two requirements: parallel gradients and feasibility. Both requirements can be packaged into the stationarity of a single function of $(\mathbf{x}, \nu)$, called the **Lagrangian**, which adds the constraint weighted by its multiplier to the objective:

$$\mathcal{L}(\mathbf{x}, \nu) = f(\mathbf{x}) + \nu\, h(\mathbf{x}). \tag{35.62}$$

Setting all partial derivatives of $\mathcal{L}$ to zero yields

$$\nabla_{\mathbf{x}} \mathcal{L} = \nabla f + \nu \nabla h = \mathbf{0} \qquad \text{and} \qquad \frac{\partial \mathcal{L}}{\partial \nu} = h(\mathbf{x}) = 0,$$

recovering Equation 35.61 and the equality constraint together. In this way, a constrained optimization problem in $\mathbf{x}$ becomes an unconstrained stationarity problem in $(\mathbf{x}, \nu)$.

Let us solve the unit-circle problem using the Lagrangian. Stationarity with respect to $\mathbf{x}$ gives $\nabla_{\mathbf{x}} \mathcal{L} = (\mathbf{x} - \mathbf{y}) + \nu \mathbf{x} = \mathbf{0}$, so $\mathbf{x} = \mathbf{y}/(1+\nu)$ must lie on the line through the origin and $\mathbf{y}$. Enforcing feasibility, $\|\mathbf{x}\| = 1$, then requires $|1 + \nu| = \|\mathbf{y}\| = 5$, which produces two solutions:

- $\nu = 4$: $\mathbf{x} = (0.6, 0.8)$, at distance 4 from $\mathbf{y}$, with $f = 8$;
- $\nu = -6$: $\mathbf{x} = (-0.6, -0.8)$, at distance 6 from $\mathbf{y}$, with $f = 18$.

The second solution is the point on the circle *farthest* from $\mathbf{y}$. This highlights an important caveat: the Lagrange condition is necessary, not sufficient, because it holds at every stationary point of f along the surface, including maxima and saddles. Here the sign of the multiplier distinguishes the minimum from the maximum. When we move to inequality constraints in the next section, the KKT conditions will turn that sign into a general rule.

When there are multiple equality constraints $h_j(\mathbf{x}) = 0$ for $j = 1, \dots, p$, each constraint receives its own multiplier and the Lagrangian becomes $\mathcal{L} = f + \sum_j \nu_j h_j$. Provided that the constraint gradients $\nabla h_j(\mathbf{x}^\star)$ are linearly independent, the same reasoning shows that $\nabla f(\mathbf{x}^\star)$ lies in the *span* of the constraint normals—that is, the objective gradient is balanced by a linear combination of the constraint gradients.

35.6.1.3 The Closest Point on a Hyperplane

The simplest constrained problem with a linear constraint asks for the point on a hyperplane closest to the origin: minimize $f(\mathbf{x}) = \tfrac{1}{2}\|\mathbf{x}\|^2$ subject to $h(\mathbf{x}) = \mathbf{a}^\top \mathbf{x} - b = 0$. Setting the gradient of $\mathcal{L} = \tfrac{1}{2}\|\mathbf{x}\|^2 + \nu(\mathbf{a}^\top \mathbf{x} - b)$ to zero gives $\mathbf{x} + \nu \mathbf{a} = \mathbf{0}$, so the closest point must lie along the normal vector $\mathbf{a}$. Substituting $\mathbf{x} = -\nu \mathbf{a}$ into the constraint yields $-\nu \|\mathbf{a}\|^2 = b$, which fixes $\nu^\star = -b/\|\mathbf{a}\|^2$. The solution and optimal value are therefore

$$\mathbf{x}^\star = \frac{b}{\|\mathbf{a}\|^2}\,\mathbf{a}, \qquad p^\star = \frac{b^2}{2\,\|\mathbf{a}\|^2}.$$

Geometrically, the nearest point sits at distance $b/\|\mathbf{a}\|$ along the unit normal $\mathbf{a}/\|\mathbf{a}\|$, matching the projection formula of Section 33.1.

For $\mathbf{a} = (3, 4)$ and $b = 5$, we land on $(0.6, 0.8)$ once again, this time with $\nu^\star = -0.2$ and $p^\star = \frac{1}{2}$: the line $3x_1 + 4x_2 = 5$ is tangent to the unit circle at $(0.6, 0.8)$. Notice that the multiplier is more than an algebraic helper. If we nudge b from 5 to 5.1, pushing the line slightly farther from the origin, the optimal value rises by about $0.02 = -\nu^\star \cdot 0.1$. As Section 35.7.3 will show in full generality, this is what a multiplier always measures: the sensitivity of the optimal value to shifting its constraint.

35.6.2 Inequality Constraints and the KKT Conditions

35.6.2.1 Active and Inactive Constraints

Most constraints in machine learning are one-sided: a norm must be at most r, a probability at least 0, or a resource budget not exceeded. We write inequality constraints in the standard form

$$\min_{\mathbf{x}} \; f(\mathbf{x}) \quad \text{subject to} \quad g_i(\mathbf{x}) \le 0, \; i = 1, \dots, m,$$

so that each g_i is negative strictly inside its region and zero along its boundary. At a solution $\mathbf{x}^\star$, every inequality falls into one of two regimes. If $g_i(\mathbf{x}^\star) < 0$, the constraint is **inactive**: the solution lies strictly inside the region, so locally the constraint has no effect. If $g_i(\mathbf{x}^\star) = 0$, the constraint is **active**: the solution sits right on the boundary, where the inequality acts like an equality.

Gradient clipping illustrates both regimes cleanly. Suppose we project a vector $\mathbf{y}$ onto the Euclidean ball of radius r by minimizing $f(\mathbf{x}) = \frac{1}{2}\|\mathbf{x}-\mathbf{y}\|^2$ subject to $g(\mathbf{x}) = \frac{1}{2}(\|\mathbf{x}\|^2 - r^2) \le 0$. Introducing a multiplier λ, stationarity of $f + \lambda g$ yields $(\mathbf{x}-\mathbf{y}) + \lambda\mathbf{x} = \mathbf{0}$, so $\mathbf{x} = \mathbf{y}/(1+\lambda)$ is always a radial rescaling of $\mathbf{y}$. Which regime applies depends on $\|\mathbf{y}\|$:

- If $\|\mathbf{y}\| \le r$, then $\mathbf{y}$ is already feasible and minimizes f outright. The constraint is inactive, $\lambda^\star = 0$, and $\mathbf{x}^\star = \mathbf{y}$; for example, $\mathbf{y} = (0.3, 0.4)$ with $r = 1$ stays untouched.
- If $\|\mathbf{y}\| > r$, no interior point can be optimal, because any interior minimum would have to be an unconstrained minimum of f, and the only unconstrained minimum is the infeasible point $\mathbf{y}$. The constraint must therefore be active, and $\|\mathbf{x}^\star\| = r$ forces $\lambda^\star = \|\mathbf{y}\|/r - 1 > 0$. For $\mathbf{y} = (3, 4)$ and $r = 1$, we get $\lambda^\star = 4$, $\mathbf{x}^\star = (0.6, 0.8)$, and $p^\star = \frac{1}{2} \cdot 4^2 = 8$.

Both cases collapse into a single formula,

$$\mathbf{x}^\star = \frac{\mathbf{y}}{\max\left(1, \; \|\mathbf{y}\|/r\right)},$$

which leaves short vectors alone and rescales long ones to norm r. Applied to a gradient vector, this is *gradient clipping* (Pascanu et al. 2013). In code, the case split becomes a single `max`, and we can check directly that at the clipped point the gradient of f is cancelled by $\lambda^\star$ times the constraint gradient $\nabla g(\mathbf{x}) = \mathbf{x}$.

```
y, r = np.array([3.0, 4.0]), 1.0
lam = max(np.linalg.norm(y) / r - 1, 0.0)    # multiplier: 0 if y is inside the ball
x = y / (1 + lam)                            # x* = y / max(1, |y|/r)
print('x* =', x, ' lambda* =', lam, ' f(x*) =', 0.5 * ((x - y)**2).sum())
```

```
print('grad f(x*) =', x - y, ' lambda* grad g(x*) =', lam * x,
      ' sum =', (x - y) + lam * x)
```

```
x* = [0.6 0.8]  lambda* = 4.0  f(x*) = 8.0
grad f(x*) = [-2.4 -3.2]  lambda* grad g(x*) = [2.4 3.2]  sum = [0. 0.]
```

Stationarity holds even though ∇f is far from zero: the objective gradient $(-2.4, -3.2)$ and the boundary's push $(2.4, 3.2)$ sum to zero, with the multiplier $\lambda^\star = 4$ giving the ratio of their lengths.

Notice the geometric meaning of the multiplier's sign. Because $g(\mathbf{x})$ increases as we leave the ball, its gradient $\nabla g = \mathbf{x}$ points outward. Stationarity written as $-\nabla f = \lambda \nabla g$ with $\lambda > 0$ says that the downhill direction $-\nabla f$ pulls outward, while the boundary wall pushes back. If $-\nabla f$ pointed inward instead, we could step into the interior of the ball to lower f, so the point could not be optimal. A negative multiplier is therefore ruled out: $\lambda = -6$ would select the far point $(-0.6, -0.8)$ on the circle, which is a maximum rather than a minimum over the disk. Unlike an equality constraint, an inequality wall pushes in only one direction.

35.6.2.2 The Karush–Kuhn–Tucker Conditions

Combining inequality and equality constraints gives the general optimization problem

$$\min_{\mathbf{x}} \; f(\mathbf{x}) \quad \text{subject to} \quad g_i(\mathbf{x}) \le 0, \; i = 1, \dots, m, \qquad h_j(\mathbf{x}) = 0, \; j = 1, \dots, p. \tag{35.63}$$

Its **Lagrangian** assigns one multiplier to each constraint, with the inequality multipliers required to be nonnegative:

$$\mathcal{L}(\mathbf{x}, \lambda, \nu) = f(\mathbf{x}) + \sum_{i=1}^{m} \lambda_i \, g_i(\mathbf{x}) + \sum_{j=1}^{p} \nu_j \, h_j(\mathbf{x}), \qquad \lambda_i \ge 0. \tag{35.64}$$

At any local minimum $\mathbf{x}^\star$ where a constraint qualification holds (we specify which ones in the next subsection), there exist multipliers satisfying the four **Karush–Kuhn–Tucker (KKT) conditions** (Karush 1939; Kuhn and Tucker 1951):

$$\begin{aligned} &\text{stationarity:} && \nabla f(\mathbf{x}^\star) + \textstyle\sum_i \lambda_i^\star \nabla g_i(\mathbf{x}^\star) + \sum_j \nu_j^\star \nabla h_j(\mathbf{x}^\star) = \mathbf{0}, \\ &\text{primal feasibility:} && g_i(\mathbf{x}^\star) \le 0, \qquad h_j(\mathbf{x}^\star) = 0, \\ &\text{dual feasibility:} && \lambda_i^\star \ge 0, \\ &\text{complementary slackness:} && \lambda_i^\star \, g_i(\mathbf{x}^\star) = 0 \quad \text{for every } i. \end{aligned} \tag{35.65}$$

We have already seen the first three conditions in our examples: a balance of gradients, satisfaction of the constraints themselves, and the one-way sign restriction on inequality multipliers. The fourth, **complementary slackness**, states that for each i at least one of $\lambda_i^\star$ and $g_i(\mathbf{x}^\star)$ must vanish. Consequently, every inactive constraint ($g_i(\mathbf{x}^\star) < 0$) automatically gets $\lambda_i^\star = 0$ and drops out of the stationarity equation. Keep in mind a subtle asymmetry:

a positive multiplier guarantees that a constraint is active, but an active constraint can still have $\lambda_i^\star = 0$ if it happens to touch $\mathbf{x}^\star$ without exerting any force. In the ball-projection example, complementary slackness encodes the exact case split we carried out by hand, and at $(0.6, 0.8)$ we have $\lambda^\star g(\mathbf{x}^\star) = 4 \cdot 0 = 0$.

Figure 35.21 shows the geometry of stationarity and dual feasibility together. At the optimum, the descent direction $-\nabla f$ must be a nonnegative linear combination of the outward normals of the active constraints. Equivalently, $-\nabla f$ lies inside the cone spanned by those outward normals (the *normal cone*), while inactive constraints contribute nothing.

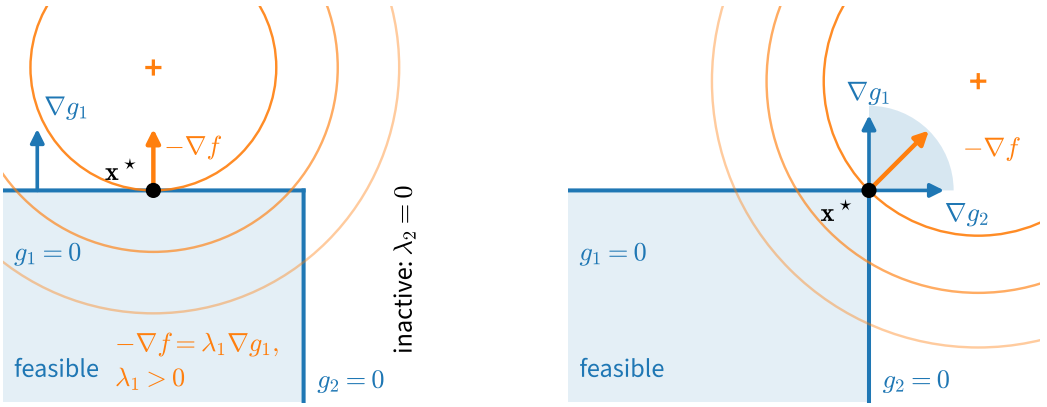


Figure 35.21: Geometry of the KKT conditions on a feasible region defined by two inequality constraints. In (a), one constraint is active at $\mathbf{x}^\star$ and the other is inactive: $-\nabla f$ aligns with the active outward normal with $\lambda_1 > 0$, while $\lambda_2 = 0$. In (b), both constraints are active at a corner, so $-\nabla f$ must lie in the shaded normal cone spanned by both outward normals. Orange arcs show level sets of f around its unconstrained minimizer (+).

Like the Lagrange condition, the KKT conditions are first-order conditions and are not sufficient on their own. Consider minimizing $f(x) = -x^2$ on $[-1, 1]$, written as $x - 1 \le 0$ and $-1 - x \le 0$. At $x = 0$, the gradient vanishes, both constraints are inactive, and setting both multipliers to zero satisfies all four KKT conditions—yet $x = 0$ is the *maximum*. The true minima $x = \pm 1$ are KKT points too, each with multiplier 2 on its active boundary constraint.

35.6.2.3 Constraint Qualifications and Sufficiency

This leaves two complementary questions. First, when is a local minimum *guaranteed* to satisfy the KKT conditions? Second, when is a KKT point *guaranteed* to be a minimum?

For the first question, the answer parallels the equality case: a minimum must satisfy the KKT conditions whenever a constraint qualification holds. The most common general condition is the *linear independence constraint qualification* (LICQ). Suppose $\mathbf{x}^\star$ is a local minimum of Equation 35.63 with f, g_i, and h_j continuously differentiable. If the gradients of all active inequality constraints together with the gradients of all equality constraints, $\{\nabla g_i(\mathbf{x}^\star) : g_i(\mathbf{x}^\star) = 0\} \cup \{\nabla h_j(\mathbf{x}^\star) : j = 1, \dots, p\}$, are linearly independent, then multipliers satisfying Equation 35.65 are guaranteed to exist (Nocedal and Wright (2006), chapter 12). LICQ directly generalizes $\nabla h(\mathbf{x}^\star) \neq \mathbf{0}$, and Exercise 1 illustrates what can go wrong when it fails.

For convex problems, there is a much simpler check that requires finding only a single feasible point rather than inspecting gradients at the unknown solution. Call problem Equation 35.63 *convex* when f and every g_i are convex and every h_j is affine. A convex problem satisfies **Slater's condition** (Slater 1950) if there exists a point $\bar{\mathbf{x}}$ such that

$$\begin{aligned} g_i(\bar{\mathbf{x}}) &< 0 \quad \text{for every } g_i \text{ that is not affine},\\ g_i(\bar{\mathbf{x}}) &\le 0 \quad \text{for every affine } g_i,\\ h_j(\bar{\mathbf{x}}) &= 0 \quad \text{for every } j. \end{aligned} \tag{35.66}$$

In plain terms, $\bar{\mathbf{x}}$ must be feasible and lie strictly inside every curved inequality constraint; if all constraints are affine, then mere feasibility already satisfies Slater's condition. (When f or some g_i is defined only on a subset of $\mathbb{R}^n$, which will not occur here, $\bar{\mathbf{x}}$ must also lie in the relative interior of that domain.) For the Euclidean ball, $\bar{\mathbf{x}} = \mathbf{0}$ immediately works because $g(\mathbf{0}) = -r^2/2 < 0$. Whenever Slater's condition holds for a convex problem with differentiable f and g_i, every minimum is a KKT point (Boyd and Vandenberghe 2004). The deeper reason is strong duality, which Section 35.7.2 derives from the very same condition.

Convexity also settles the second question, giving the single most useful result of this section.

Proposition (KKT sufficiency under convexity). *In problem Equation 35.63, let f and every g_i be convex and differentiable and every h_j affine. If $(\mathbf{x}^\star, \lambda^\star, \nu^\star)$ satisfies the KKT conditions Equation 35.65, then $\mathbf{x}^\star$ is a global minimum.*

Proof. Define $\varphi(\mathbf{x}) = \mathcal{L}(\mathbf{x}, \lambda^\star, \nu^\star)$. Because $\lambda_i^\star \ge 0$, φ is a nonnegative combination of convex functions plus affine terms, so φ is convex. Stationarity says $\nabla\varphi(\mathbf{x}^\star) = \mathbf{0}$, which means $\mathbf{x}^\star$ is a *global* minimizer of φ over all of $\mathbb{R}^n$. Now pick any feasible $\mathbf{x}$. Since $\lambda_i^\star \ge 0$, $g_i(\mathbf{x}) \le 0$, and $h_j(\mathbf{x}) = 0$, we have

$$f(\mathbf{x}) \;\ge\; f(\mathbf{x}) + \sum_i \lambda_i^\star g_i(\mathbf{x}) + \sum_j \nu_j^\star h_j(\mathbf{x}) = \varphi(\mathbf{x}) \;\ge\; \varphi(\mathbf{x}^\star) = f(\mathbf{x}^\star),$$

where the final equality follows from complementary slackness ($\sum_i \lambda_i^\star g_i(\mathbf{x}^\star) = 0$) and primal feasibility ($h_j(\mathbf{x}^\star) = 0$). $\blacksquare$

Notice a broader principle tucked inside this short proof. For *any* fixed multipliers with $\lambda \ge 0$, the first inequality holds for every feasible $\mathbf{x}$, so minimizing the Lagrangian over $\mathbf{x}$ always yields a lower bound on the constrained optimum; at the KKT multipliers, that lower bound is exact. Section 35.7 builds the entire theory of duality from this observation. For convex problems, then, the KKT conditions are always sufficient—and under Slater's condition they are necessary as well, characterizing the global minima completely.

35.6.3 Projections and Projected Gradient Descent

The KKT conditions tell us how to *recognize* an optimum, and for simple constraints they let us solve for one by hand. To *compute* solutions more generally, we pair gradient descent with the nearest-feasible-point operation: take a standard gradient step, and whenever it leaves the feasible set, project it right back in.

35.6.3.1 Projection onto a Convex Set

For a nonempty closed convex set $C \subseteq \mathbb{R}^n$, the **projection** of a point $\mathbf{y}$ onto C is the unique point of C closest to $\mathbf{y}$,

$$\Pi_C(\mathbf{y}) = \operatorname*{argmin}_{\mathbf{x}\in C} \ \tfrac{1}{2}\|\mathbf{x}-\mathbf{y}\|^2.$$

Why does a unique minimizer always exist? Because we can intersect C with a large closed ball centered at $\mathbf{y}$ without excluding the optimum; that intersection is closed and bounded (hence compact), so the continuous objective attains its minimum there. Strict convexity of the squared distance then rules out ties, making Π_C a well-defined map.

We have already computed two projections in closed form: onto a hyperplane (the closest-point problem) and onto a ball (the clipping formula). Projection onto a half-space $\{\mathbf{x} : \mathbf{a}^\top\mathbf{x} \ge b\}$ follows immediately from the hyperplane formula. If $\mathbf{y}$ already satisfies the inequality, we leave it alone; if it violates it, we move straight along $\mathbf{a}$ onto the boundary hyperplane:

$$\Pi(\mathbf{y}) = \mathbf{y} + \frac{\max(0,\ b - \mathbf{a}^\top\mathbf{y})}{\|\mathbf{a}\|^2}\,\mathbf{a}.$$

Projections onto closed convex sets obey a remarkably clean geometric characterization. Let $\hat{\mathbf{x}} = \Pi_C(\mathbf{y})$ be the minimizer of $\frac{1}{2}\|\mathbf{x}-\mathbf{y}\|^2$ over C. For any other point $\mathbf{x} \in C$, convexity guarantees that the entire line segment from $\hat{\mathbf{x}}$ to $\mathbf{x}$ stays inside C. Moving from $\hat{\mathbf{x}}$ toward $\mathbf{x}$ therefore cannot decrease the objective to first order, so its directional derivative is nonnegative: $(\hat{\mathbf{x}}-\mathbf{y})^\top(\mathbf{x}-\hat{\mathbf{x}}) \ge 0$. Flipping the sign gives the **obtuse-angle inequality** (or variational inequality):

$$\big(\mathbf{y}-\Pi_C(\mathbf{y})\big)^\top\big(\mathbf{x}-\Pi_C(\mathbf{y})\big) \ \le\ 0 \qquad \text{for all } \mathbf{x}\in C. \tag{35.67}$$

Geometrically, the *residual* vector $\mathbf{y}-\Pi_C(\mathbf{y})$ forms an angle of at least 90° with the vector from $\Pi_C(\mathbf{y})$ to every point $\mathbf{x} \in C$ (Figure 35.22, panel (a)). Put differently, the hyperplane through $\Pi_C(\mathbf{y})$ with normal $\mathbf{y}-\Pi_C(\mathbf{y})$ is a *supporting hyperplane* of C, keeping the entire set C on its far side.

Let us check this inequality on our running example. For $\mathbf{y} = (3,4)$ projected onto the unit disk, the residual is $(2.4, 3.2) = 4\,(0.6, 0.8)$—once again, the multiplier times the outward normal. For any $\mathbf{x}$ with $\|\mathbf{x}\| \le 1$,

$$(2.4,3.2)^\top(\mathbf{x}-(0.6,0.8)) = 4\,(0.6x_1 + 0.8x_2) - 4 \ \le\ 4\,\|\mathbf{x}\| - 4 \ \le\ 0,$$

where the first inequality uses Cauchy–Schwarz, $\mathbf{u}^\top\mathbf{v} \le \|\mathbf{u}\|\,\|\mathbf{v}\|$. The supporting line here is $3x_1 + 4x_2 = 5$, the very hyperplane from our closest-point example.

Just as importantly, the obtuse-angle inequality is also *sufficient*. Suppose some point $\hat{\mathbf{x}} \in C$ satisfies Equation 35.67, and write $\varphi(\mathbf{x}) = \frac{1}{2}\|\mathbf{x}-\mathbf{y}\|^2$, whose gradient at $\hat{\mathbf{x}}$ is $\hat{\mathbf{x}}-\mathbf{y}$. Because a convex differentiable function always lies above its tangent planes (Section 35.3), we have $\varphi(\mathbf{x}) \ge \varphi(\hat{\mathbf{x}}) + (\hat{\mathbf{x}}-\mathbf{y})^\top(\mathbf{x}-\hat{\mathbf{x}})$. The last term is nonnegative for every $\mathbf{x} \in C$ by assumption, so $\varphi(\mathbf{x}) \ge \varphi(\hat{\mathbf{x}})$ across all of C, proving that $\hat{\mathbf{x}} = \Pi_C(\mathbf{y})$. Thus Equation 35.67 *completely*

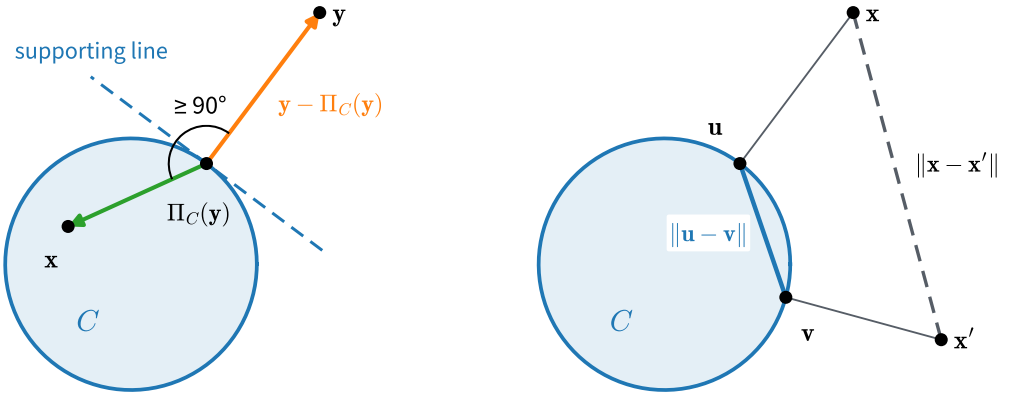


Figure 35.22: (a) The residual $\mathbf{y} - \Pi_C(\mathbf{y})$ makes an angle of at least 90° with every direction into C, so C lies entirely on the far side of the supporting line through $\Pi_C(\mathbf{y})$. (b) Projection onto a closed convex set never increases distances: $\|\mathbf{u} - \mathbf{v}\| \le \|\mathbf{x} - \mathbf{x}'\|$.

characterizes the projection. Every property we need for projected algorithms flows directly from it, beginning with the fact that projections never stretch distances.

Proposition (projections are nonexpansive). *For a nonempty closed convex set C and all $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$,*

$$\|\Pi_C(\mathbf{x}) - \Pi_C(\mathbf{y})\| \;\le\; \|\mathbf{x} - \mathbf{y}\|.$$

Proof. Set $\mathbf{u} = \Pi_C(\mathbf{x})$ and $\mathbf{v} = \Pi_C(\mathbf{y})$. Applying Equation 35.67 twice—first at $\mathbf{x}$ with $\mathbf{v} \in C$, then at $\mathbf{y}$ with $\mathbf{u} \in C$—gives

$$(\mathbf{x} - \mathbf{u})^\top (\mathbf{v} - \mathbf{u}) \le 0, \qquad (\mathbf{y} - \mathbf{v})^\top (\mathbf{u} - \mathbf{v}) \le 0.$$

Adding these two inequalities yields $(\mathbf{x} - \mathbf{y} - (\mathbf{u} - \mathbf{v}))^\top (\mathbf{v} - \mathbf{u}) \le 0$, which rearranges to

$$\|\mathbf{u} - \mathbf{v}\|^2 \;\le\; (\mathbf{x} - \mathbf{y})^\top (\mathbf{u} - \mathbf{v}) \;\le\; \|\mathbf{x} - \mathbf{y}\|\,\|\mathbf{u} - \mathbf{v}\|$$

by Cauchy–Schwarz. Dividing by $\|\mathbf{u} - \mathbf{v}\|$ finishes the proof (and when $\|\mathbf{u} - \mathbf{v}\| = 0$ the claim is trivial). $\blacksquare$

Panel (b) of Figure 35.22 depicts this visually: two points outside the disk are always at least as far apart as their projections. Convexity of C is essential here. Take the nonconvex two-point set $C = \{-1, 1\}$: then $y = 0.01$ projects to 1 while $y = -0.01$ projects to -1, turning an input gap of 0.02 into an output gap of 2—and at $y = 0$ the projection is not even unique. Projecting onto a nonconvex set can be both discontinuous and multivalued.

35.6.3.2 Projected Gradient Descent

Projected gradient descent (PGD) turns projection into an iterative solver by projecting after every gradient step:

$$\mathbf{x}_{k+1} = \Pi_C\left(\mathbf{x}_k - \eta\,\nabla f(\mathbf{x}_k)\right). \tag{35.68}$$

Every iterate stays feasible by construction. Better yet, the fixed points of PGD are precisely the constrained stationary points. For any differentiable f, closed convex C, and step size $\eta > 0$, the fixed-point equation $\mathbf{x}^\star = \Pi_C(\mathbf{x}^\star - \eta\nabla f(\mathbf{x}^\star))$ holds if and only if $\mathbf{x}^\star \in C$ satisfies the obtuse-angle inequality Equation 35.67 with $\mathbf{y} = \mathbf{x}^\star - \eta\nabla f(\mathbf{x}^\star)$. Cancelling $\mathbf{x}^\star$ and dividing by $\eta > 0$ turns that inequality into

$$\nabla f(\mathbf{x}^\star)^\top(\mathbf{x} - \mathbf{x}^\star) \;\geq\; 0 \qquad \text{for all } \mathbf{x} \in C.$$

In plain words, no feasible direction is a descent direction. This is the general constrained first-order condition, reducing to $\nabla f(\mathbf{x}^\star) = \mathbf{0}$ when $C = \mathbb{R}^n$. When f is convex, the tangent-plane inequality immediately makes this condition sufficient for a global minimum; when f is nonconvex, it certifies stationarity. And when $C = \{\mathbf{x} : g_i(\mathbf{x}) \leq 0\}$ for differentiable convex g_i satisfying Slater's condition, it is equivalent to the KKT conditions.

Does the extra projection step slow gradient descent down? Remarkably, no—and nonexpansiveness gives a three-line proof.

Proposition (PGD contracts at least as fast as GD). *Let C be nonempty, closed, and convex, let f be twice continuously differentiable with $\mu\mathbf{I} \preceq \nabla^2 f \preceq L\mathbf{I}$ for some $0 < \mu \leq L$, let $\mathbf{x}^\star$ be the minimizer of f over C, and let $\eta > 0$. Then the iterates of Equation 35.68 satisfy*

$$\|\mathbf{x}_{k+1} - \mathbf{x}^\star\| \;\leq\; \rho\,\|\mathbf{x}_k - \mathbf{x}^\star\|, \qquad \rho = \max\left(|1-\eta\mu|,\; |1-\eta L|\right). \tag{35.69}$$

Proof. (i) Because $\mathbf{x}^\star$ is a fixed point of PGD, we can write $\mathbf{x}^\star = \Pi_C(\mathbf{b})$ with $\mathbf{b} = \mathbf{x}^\star - \eta\nabla f(\mathbf{x}^\star)$. (ii) That choice of $\mathbf{b}$ is the key step: applying nonexpansiveness to $\mathbf{x}_k - \eta\nabla f(\mathbf{x}_k)$ and $\mathbf{b}$ strips away the projection altogether,

$$\|\mathbf{x}_{k+1} - \mathbf{x}^\star\| \;\leq\; \left\|(\mathbf{x}_k - \mathbf{x}^\star) - \eta\left(\nabla f(\mathbf{x}_k) - \nabla f(\mathbf{x}^\star)\right)\right\|.$$

(iii) By the fundamental theorem of calculus along the segment joining $\mathbf{x}^\star$ to $\mathbf{x}_k$, we have $\nabla f(\mathbf{x}_k) - \nabla f(\mathbf{x}^\star) = \bar{\mathbf{H}}(\mathbf{x}_k - \mathbf{x}^\star)$, where $\bar{\mathbf{H}} = \int_0^1 \nabla^2 f(\mathbf{x}^\star + t(\mathbf{x}_k - \mathbf{x}^\star))\,dt$. This averaged Hessian is symmetric with all eigenvalues in $[\mu, L]$, so every eigenvalue of $\mathbf{I} - \eta\bar{\mathbf{H}}$ has the form $1 - \eta h$ with $h \in [\mu, L]$ and magnitude at most ρ. The right-hand side is therefore at most $\rho\,\|\mathbf{x}_k - \mathbf{x}^\star\|$. ■

Once step (ii) removes the projection, all that remains is the contraction of unconstrained gradient descent. For a quadratic $f = \frac{1}{2}\mathbf{x}^\top\mathbf{A}\mathbf{x}$, we have $\bar{\mathbf{H}} = \mathbf{A}$, and step (iii) recovers the exact per-mode factor $|1-\eta\lambda_i(\mathbf{A})|$ from Section 35.1, which is strictly less than 1 whenever $0 < \eta < 2/L$. For a convex L-smooth f that is not strongly convex, PGD with $\eta = 1/L$ likewise preserves gradient descent's $O(1/k)$ objective-value rate (Nesterov 2018).

Let us test how tight the contraction factor ρ is in practice. Consider the ill-conditioned quadratic $f(\mathbf{x}) = \frac{1}{2}(x_1^2 + 1000\,x_2^2)$ with condition number $\kappa = 10^3$, minimized over the

half-space $x_1 + x_2 \geq 1$ (written $g(\mathbf{x}) = 1 - x_1 - x_2 \leq 0$). We can solve the KKT conditions by hand: the unconstrained minimizer $\mathbf{0}$ is infeasible, so the constraint must be active. Stationarity, $(x_1, 1000\, x_2) - \lambda(1, 1) = \mathbf{0}$, gives $x_1 = \lambda$ and $x_2 = \lambda/1000$, and plugging these into $x_1 + x_2 = 1$ yields $\lambda^\star = 1000/1001 \approx 0.999$, $\mathbf{x}^\star = (1000/1001,\ 1/1001) \approx (0.999, 0.001)$, and $p^\star = 500/1001 \approx 0.4995$.

Now run PGD with step size $\eta = 0.0015$, safely below the stability limit $2/L = 0.002$. The proposition guarantees a per-step factor of $\rho = \max(1-\eta,\ |1-1000\,\eta|) = 0.9985$. Starting from $(3, 4)$ at distance 4.47 from $\mathbf{x}^\star$, that bound predicts up to $\log(4.47 \times 10^6)/\log(1/\rho) \approx 10{,}200$ steps to reach distance 10^{-6}. The code below runs PGD and inspects the worst single-step ratio, the actual step count, the moment when every step begins hitting the boundary, the contraction ratio along the boundary, and the final gradient.

```
A = np.diag([1.0, 1000.0])                          # f(x) = 1/2 x^T A x, kappa = 1000
x_star = np.array([1000 / 1001, 1 / 1001])          # KKT point of x1 + x2 >= 1
project = lambda y: y + max(0.0, 1 - y.sum()) / 2   # onto the half-space x1 + x2 >= 1
eta = 1.5e-3
rho = max(abs(1 - eta), abs(1 - 1000 * eta))        # the GD contraction factor
x, dist, projected = np.array([3.0, 4.0]), [], []
for k in range(1000):
    y = x - eta * A @ x
    projected.append(y.sum() < 1)                   # did the step leave C?
    x = project(y)
    dist.append(np.linalg.norm(x - x_star))
dist, projected = np.array(dist), np.array(projected)
ratios = dist[1:] / dist[:-1]
steps = np.argmax(dist < 1e-6) + 1
on_line = np.flatnonzero(~projected).max() + 2      # every step from here on is projected
print(f'bound rho = {rho:.4f}   largest ratio in a step = '
      f'{ratios[dist[:-1] > 1e-12].max():.4f}')
print(f'steps to |x - x*| < 1e-6: {steps}   every step projected from step {on_line} on')
print(f'ratio per step on the line: {np.median(ratios[on_line:steps - 1]):.4f}'
      f'   1 - 500.5 eta = {1 - 500.5 * eta:.4f}')
print(f'grad f at the end = {A @ x}')
```

```
bound rho = 0.9985   largest ratio in a step = 0.9977
steps to |x - x*| < 1e-6: 738   every step projected from step 733 on
ratio per step on the line: 0.2492   1 - 500.5 eta = 0.2492
grad f at the end = [0.999001 0.999001]
```

The largest single-step ratio, 0.9977, comes close to $\rho = 0.9985$, so the worst-case per-step bound is nearly tight. The total step count, however, is a pleasant surprise: 738 steps instead of $10{,}200$. The printed diagnostics reveal why—the trajectory unfolds in two distinct phases. During the first phase, the constraint is inactive after the initial step: the stiff coordinate x_2 collapses within a few dozen iterations, while the flat coordinate x_1 creeps inward by the factor $1-\eta$ per step, just as in unconstrained gradient descent. Reaching the boundary line from $x_1 = 3$ takes roughly $\log 3/\eta \approx 730$ steps. From step 733 onward, every update hits the boundary and is projected onto the line $x_1 + x_2 = 1$. Along that line, the quadratic has curvature $(1 + 1000)/2 = 500.5$, so the distance to $\mathbf{x}^\star$ shrinks by $1 - 500.5\,\eta = 0.249$ per step—the exact rate of one-dimensional gradient descent along the line—and just a handful of additional steps finish the run. While ρ must guard against the worst curvature anywhere in the plane, once the constraint becomes active only the curvature *along the boundary* matters. Finally, notice the gradient at termination: $(0.999, 0.999) = \lambda^\star(1, 1) = -\lambda^\star \nabla g$. Far

from vanishing, it hands us back the multiplier $\lambda^\star = 1000/1001$ we derived by hand.

PGD is a workhorse whenever Π_C is inexpensive to compute. Nonnegativity constraints project onto the nonnegative orthant simply by zeroing negative entries, and max-norm constraints on the rows of a weight matrix rescale each row onto a ball after every update. Perhaps the most famous deep learning application is generating adversarial examples via projected gradient ascent on an input perturbation δ with $\|\delta\|_\infty \le \epsilon$, where projection simply clips each entry to $[-\epsilon, \epsilon]$ (Madry et al. 2018). Keep in mind that gradient clipping also uses a ball projection, but it projects the *gradient vector* rather than the parameters, so it is not PGD for any constraint on the weights (Exercise 2 highlights the difference). What clipping does achieve is a step that stays safe where the curvature grows with the gradient (Section 35.1.1.5).

35.6.3.3 Projection onto the Simplex

Attention weights and class probabilities live on the **probability simplex** $\Delta = \{\mathbf{x} : \mathbf{x} \ge 0,\ \sum_i x_i = 1\}$. Given a vector of real scores $\mathbf{y}$, how do we find the closest point on Δ? The problem

$$\min_{\mathbf{x}} \ \tfrac{1}{2}\|\mathbf{x} - \mathbf{y}\|^2 \quad \text{subject to} \quad \textstyle\sum_i x_i = 1, \ \ -x_i \le 0,$$

is a small **quadratic program** (QP)—a convex quadratic objective subject to affine constraints—and the KKT conditions collapse it to finding a single scalar. Introduce a multiplier τ for the sum constraint and multipliers $\lambda_i \ge 0$ for the nonnegativity constraints $-x_i \le 0$. Stationarity gives $x_i - y_i + \tau - \lambda_i = 0$, and complementary slackness splits the coordinates into two groups. Wherever $x_i > 0$, we must have $\lambda_i = 0$ and thus $x_i = y_i - \tau$. Wherever $x_i = 0$, stationarity gives $\lambda_i = \tau - y_i$, and dual feasibility $\lambda_i \ge 0$ forces $y_i \le \tau$. Putting both cases together gives

$$x_i^\star = \max(y_i - \tau,\ 0), \qquad \text{with } \tau \text{ set by } \sum_i \max(y_i - \tau,\ 0) = 1. \tag{35.70}$$

In other words, projecting onto the simplex simply shifts every score down by the same threshold τ and clips negative results to zero. Remarkably, this threshold formula automatically satisfies stationarity, complementary slackness, dual feasibility, and nonnegativity for *any* τ; the unit-sum budget $\sum_i x_i = 1$ alone pins down τ. Because the left-hand side $\varphi(\tau) = \sum_i \max(y_i - \tau, 0)$ is continuous, piecewise linear, and strictly decreasing wherever it is positive, there is a unique τ that solves $\varphi(\tau) = 1$.

A quick hand calculation shows how to find τ. Take the already-sorted vector $\mathbf{y} = (0.9, 0.5, -0.2)$, and guess that the k largest scores survive above zero. Then the budget $\sum_{j \le k}(y_j - \tau) = 1$ predicts the candidate threshold $\tau_k = (\sum_{j \le k} y_j - 1)/k$, which is self-consistent only if the k-th score stays strictly above τ_k:

- $k = 1$: $\tau_1 = -0.1$, and $y_1 = 0.9 > \tau_1$;
- $k = 2$: $\tau_2 = 0.2$, and $y_2 = 0.5 > \tau_2$;
- $k = 3$: $\tau_3 \approx 0.067$, but $y_3 = -0.2 < \tau_3$.

The largest valid choice is $k = 2$, giving $\tau = 0.2$ and $\mathbf{x}^\star = (0.7, 0.3, 0)$ (Figure 35.23). For comparison, softmax applied to the same scores yields $(0.50, 0.33, 0.17)$, keeping every entry strictly positive.

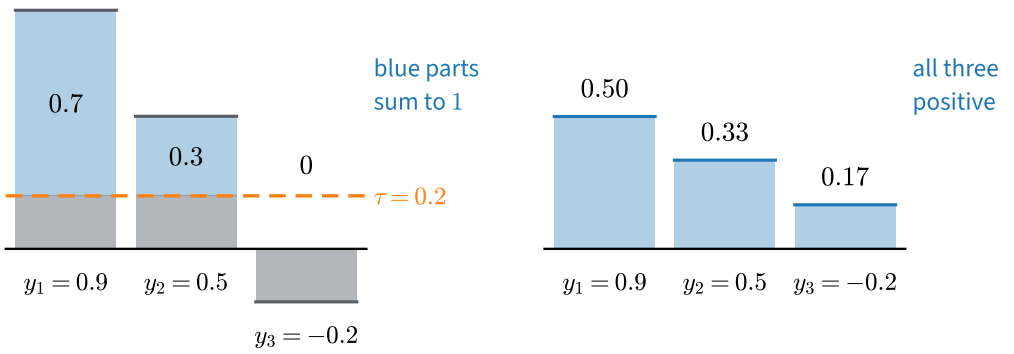


Figure 35.23: Projection of $\mathbf{y} = (0.9, 0.5, -0.2)$ onto the probability simplex in (a): every score is shifted down by the threshold $\tau = 0.2$ and clipped at zero, with τ uniquely fixed by $\sum_i x_i = 1$. Softmax on the same scores in (b) keeps every coordinate strictly positive.

Two simple observations turn this hand check into a general algorithm. First, the *support* $\{i : x_i^\star > 0\}$ always consists of the largest scores: if $y_i > y_j$ and $x_j^\star > 0$, then $x_i^\star \geq y_i - \tau > y_j - \tau = x_j^\star > 0$. (The *active* sign constraints are the remaining zero coordinates.) Second, once the scores are sorted as $u_1 \geq \cdots \geq u_n$, the test $u_k > \tau_k$ holds for every k up to the true support size and fails beyond it. Indeed, rearranging $u_k > \tau_k$ gives $\sum_{j<k}(u_j - u_k) < 1$, which is just $\varphi(u_k) < 1 = \varphi(\tau)$; because φ is strictly decreasing, this is equivalent to $u_k > \tau$. We therefore sort $\mathbf{y}$, pick the largest k with $u_k > \tau_k$, and set $\tau = \tau_k$, taking $O(n \log n)$ time (Held et al. 1974; Duchi et al. 2008).

When applied to attention scores, this simplex projection is known as *sparsemax* (Martins and Astudillo 2016), because it assigns exact zero weight to low-scoring positions. Softmax and sparsemax differ by just one regularization term: expanding the squared distance shows that sparsemax maximizes $\mathbf{y}^\top \mathbf{x} - \frac{1}{2}\|\mathbf{x}\|^2$ over Δ, whereas softmax maximizes $\mathbf{y}^\top \mathbf{x} + H(\mathbf{x})$, replacing the quadratic penalty with Shannon entropy (Section 35.7).

Whenever we implement a projection, it is good practice to verify its output. The KKT residuals are the natural first test, though we have to be careful about which ones actually discriminate. The cell below tests six random scores at both the computed τ and a perturbed value $\tau + 0.3$, and then checks the optimality certificate from the obtuse-angle inequality Equation 35.67. Because the left-hand side of Equation 35.67 is affine in $\mathbf{x}$, its maximum over Δ occurs at a vertex, so testing the n standard basis vectors $\mathbf{e}_i$ is enough to certify the entire simplex.

```
def project_simplex(y):
    """Sort-and-threshold projection onto {x >= 0, sum x = 1}; returns x* and tau."""
    u = np.sort(y)[::-1]
    tau_k = (np.cumsum(u) - 1) / np.arange(1, len(y) + 1)   # tau_k for k = 1..n
    k = np.nonzero(u > tau_k)[0][-1]                          # largest k with u_k > tau_k
    return np.maximum(y - tau_k[k], 0.0), tau_k[k]
```

```
y = np.random.default_rng().normal(size=6)
x, tau = project_simplex(y)
print('x* =', x.round(3), ' zeros:', int((x == 0).sum()), ' tau =', round(tau, 3))
for t in (tau, tau + 0.3):                    # three conditions hold for every tau ...
    xt, lam = np.maximum(y - t, 0), np.maximum(0, t - y)
    print(f'tau = {t:6.3f}: stationarity {abs(xt - y + t - lam).max():.0e}, '
          f'comp. slack {abs(lam * xt).max():.0e}, dual feas. {lam.min() >= 0}, '
          f'sum x = {xt.sum():.3f}')           # ... only the budget differs
cert = ((y - x) @ (np.eye(6) - x).T).max()    # max_i (y - x*)^T (e_i - x*)
print(f'certificate max_i (y - x*)^T(e_i - x*) = {cert:.0e}  '
      f'(<= 0 up to rounding: {cert <= 1e-12})')
```

```
x* = [0.666 0.334 0.    0.    0.    0.   ]  zeros: 4  tau = 1.662
tau =  1.662: stationarity 0e+00, comp. slack 0e+00, dual feas. True, sum x = 1.000
tau =  1.962: stationarity 0e+00, comp. slack 0e+00, dual feas. True, sum x = 0.400
certificate max_i (y - x*)^T(e_i - x*) = -3e-16  (<= 0 up to rounding: True)
```

Typically three to five of the six coordinates are set to exact zero; these are the active nonnegativity constraints, each carrying the positive multiplier $\tau - y_i$. Notice the diagnostic pitfall: stationarity, complementary slackness, and dual feasibility all pass up to rounding error for *both* the correct τ and the wrong one. Only the sum constraint catches the error, so checking those three residuals alone cannot confirm the solution. The vertex certificate does: seeing that $\max_i (\mathbf{y} - \mathbf{x}^\star)^\top (\mathbf{e}_i - \mathbf{x}^\star) \le 0$ up to rounding error proves that no point on the simplex lies closer to $\mathbf{y}$.

35.6.4 Proximal Operators and Coordinate Descent

35.6.4.1 Soft-Thresholding and the Proximal Operator

A hard boundary is not the only way to keep weights under control. The **lasso** (Tibshirani 1996) replaces a hard wall with a kinked ℓ_1 penalty added to a least-squares loss,

$$F(\mathbf{w}) = \tfrac{1}{2}\|\mathbf{A}\mathbf{w} - \mathbf{b}\|^2 + \lambda \|\mathbf{w}\|_1, \qquad \mathbf{A} \in \mathbb{R}^{n \times d},$$

whose minimizer typically sets many coordinates to exact zero. We write the penalty weight as λ on purpose, because it acts like the Lagrange multiplier for a norm constraint $\|\mathbf{w}\|_1 \le c$: every minimizer of F also minimizes the least-squares loss under the ℓ_1 budget c that it uses, and conversely every budget $c > 0$ corresponds to some penalty λ. Section 35.7.3.2 proves both directions, for the lasso and for ℓ_2 weight decay alike.

To see what happens numerically, let us plant a sparse vector $\mathbf{w}$ in $\mathbb{R}^8$ with five zero coordinates, generate $n = 100$ noisy linear observations, and set $\lambda = 3$. We choose the step size $\eta = 1/L$, where $L = \lambda_{\max}(\mathbf{A}^\top \mathbf{A})$ is the largest eigenvalue of $\mathbf{A}^\top \mathbf{A}$ and the smoothness constant of the least-squares term.

```
rng = np.random.default_rng()
w_planted = np.array([2.0, 0, 0, -1.5, 0, 0, 0, 1.0])
A = rng.normal(size=(100, 8))
b = A @ w_planted + 0.05 * rng.normal(size=100)
lam = 3.0
L = np.linalg.eigvalsh(A.T @ A).max()
```

```
eta = 1 / L
F = lambda w: 0.5 * ((A @ w - b)**2).sum() + lam * np.abs(w).sum()
soft = lambda z, t: np.sign(z) * np.maximum(np.abs(z) - t, 0.0)  # derived below
print(f'L = lambda_max(A^T A) = {L:.1f};  |a_j|^2 range from '
      f'{(A**2).sum(0).min():.0f} to {(A**2).sum(0).max():.0f}')
```

```
L = lambda_max(A^T A) = 145.6;  |a_j|^2 range from 85 to 109
```

Here L typically lands between 130 and 190, while the squared column norms $\|\mathbf{a}_j\|^2$ of $\mathbf{A}$ sit around 100.

The most naive algorithm to try is subgradient descent. Recall from Section 35.3 that the subgradients of $|w|$ are $\text{sign}(w)$ for $w \neq 0$ and the entire interval $[-1, 1]$ at the kink $w = 0$. Picking $\text{sign}(\mathbf{w})$ at each step gives the update $\mathbf{w} \leftarrow \mathbf{w} - \eta\,(\mathbf{A}^\top(\mathbf{A}\mathbf{w} - \mathbf{b}) + \lambda\,\text{sign}(\mathbf{w}))$. Does it recover the five zeros?

```
w_sub, signs = np.zeros(8), []
for k in range(50):                                  # subgradient descent, step 1/L
    w_sub = w_sub - eta * (A.T @ (A @ w_sub - b) + lam * np.sign(w_sub))
    signs.append(np.sign(w_sub))
null = w_planted == 0
S = np.array(signs)[-20:, null]
print(f'exact zeros: {int((w_sub == 0).sum())} of 8;  sign flips of the null '
      f'coordinates, last 20 steps: {(S[1:] != S[:-1]).sum()} of 95')
print(f'largest null coordinate: {np.abs(w_sub[null]).max():.3f};  '
      f'eta * lam = {eta * lam:.3f}')
```

```
exact zeros: 0 of 8;  sign flips of the null coordinates, last 20 steps: 95 of 95
largest null coordinate: 0.033;  eta * lam = 0.021
```

It fails completely to produce exact zeros. None of the coordinates land on zero, and the five *null* coordinates (those planted at zero) flip sign on nearly every iteration, bouncing back and forth at an amplitude of about $\eta\lambda$. The culprit is the kink at zero. At the true lasso solution, the smooth gradient on a null coordinate has magnitude strictly smaller than λ. Near zero, the penalty term $-\eta\lambda\,\text{sign}(w_j)$ therefore dominates the step and kicks the coordinate across zero by about $\eta\lambda$ every time. Because subgradient descent commits to a single slope out of the interval $[-\lambda, \lambda]$ at each step, a fixed step size can never settle at the exact slope that cancels the smooth gradient.

The cure is simple: instead of linearizing the kinked penalty, handle it *exactly*. Start in one dimension by minimizing $\lambda|x| + \frac{1}{2}(x - z)^2$ over $x \in \mathbb{R}$. Because the objective is convex, x is optimal if and only if 0 belongs to its set of subgradients:

- If $x \neq 0$, the objective is differentiable at x, so $\lambda\,\text{sign}(x) + x - z = 0$. When $x > 0$ this gives $x = z - \lambda$, which requires $z > \lambda$; when $x < 0$ it gives $x = z + \lambda$, which requires $z < -\lambda$.
- If $x = 0$, the subgradients form the interval $[-\lambda, \lambda] - z$, which contains 0 if and only if $|z| \leq \lambda$.

These two cases partition the real line and combine into **soft-thresholding**,

$$\text{prox}_{\lambda|\cdot|}(z) \;=\; \text{sign}(z)\,\max\left(|z| - \lambda,\, 0\right). \tag{35.71}$$

Any input with magnitude at most λ is snapped to exact zero, while larger inputs are shrunk toward zero by λ. For example, with $\lambda = 0.7$, $z = 0.5$ maps to 0, $z = 2$ maps to 1.3, and $z = -2$ maps to -1.3 (Figure 35.24). That dead zone $[-\lambda, \lambda]$ is precisely how ℓ_1 regularization creates exact sparsity, whereas ℓ_2 regularization only shrinks weights toward zero without ever zeroing them out.

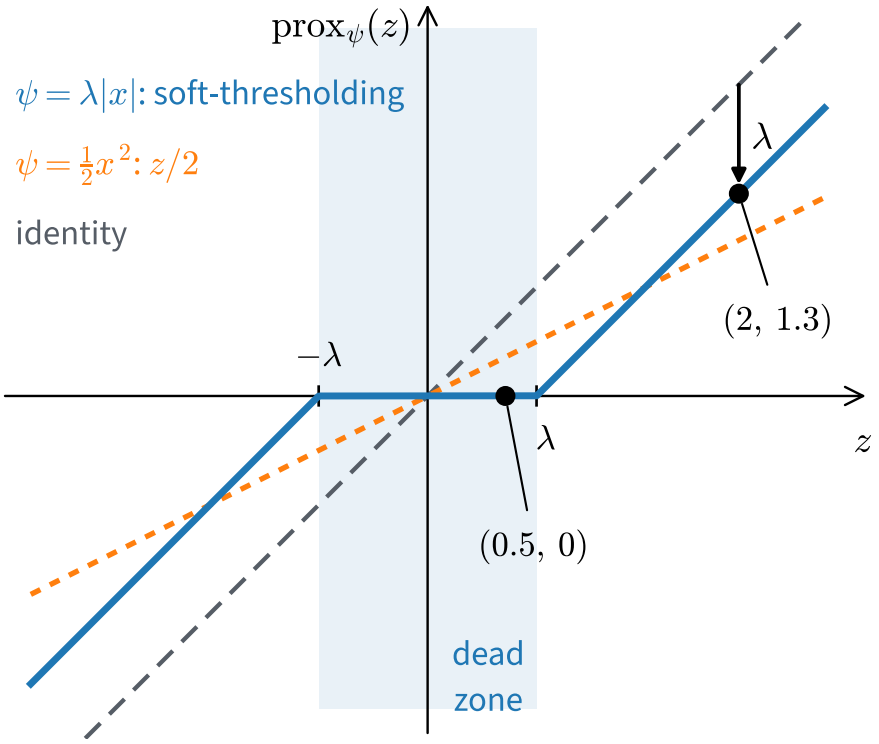


Figure 35.24: Soft-thresholding $\mathrm{prox}_{\lambda|\cdot|}(z)$ for $\lambda = 0.7$ (solid). Inputs in the dead zone $[-\lambda, \lambda]$ map to 0, as $z = 0.5$ does, and every other input moves toward 0 by λ, as $z = 2$ does. The dashed line is the identity. The dotted line is the corresponding map for the squared penalty $\frac{1}{2}x^2$, which halves every input and sends none but $z = 0$ to zero.

Soft-thresholding is one instance of a much more general operation. The **proximal operator** of a function ψ trades off lowering $\psi(\mathbf{x})$ against staying close to a reference point $\mathbf{z}$:

$$\mathrm{prox}_\psi(\mathbf{z}) \;=\; \operatorname*{argmin}_{\mathbf{x}} \; \left(\psi(\mathbf{x}) + \tfrac{1}{2}\,\|\mathbf{x} - \mathbf{z}\|^2\right). \tag{35.72}$$

This map is well defined for every *proper closed convex* function ψ—that is, convex, taking values in $\mathbb{R} \cup \{+\infty\}$, finite at least somewhere, and having a closed epigraph. Under those conditions, the objective in Equation 35.72 is 1-strongly convex and therefore has a unique minimizer (Parikh and Boyd 2014).

Three special cases reveal how the proximal operator unifies ideas we already know:

- When $\psi = 0$, the proximal operator is simply the identity map.

- When ψ is the *indicator* of a closed convex set C (equal to 0 on C and $+\infty$ outside), any minimizer must lie in C, so the proximal operator reduces to the Euclidean projection $\text{prox}_{\psi} = \Pi_C$. Proximal operators are projections generalized from sets to functions.
- When $\psi = \lambda\|\cdot\|_1$, the objective separates completely across coordinates, so the proximal operator applies soft-thresholding Equation 35.71 coordinate by coordinate.

Exercise 5 works out the proximal operator of the Euclidean norm, which zeroes an entire block of weights at once, and Exercise 6 shows that the proximal operator of a norm and the projection onto its dual ball sum to the identity.

35.6.4.2 Proximal Gradient Descent

The lasso objective has the composite form $F = s + \psi$, combining a smooth loss s with a nonsmooth penalty ψ whose proximal operator is cheap to evaluate. Suppose s is convex with an L-Lipschitz gradient, ψ is proper closed convex, and $0 < \eta \leq 1/L$. By the descent lemma (Section 35.1, with $1/\eta \geq L$), s is bounded above by the quadratic model $s(\mathbf{x}_k) + \nabla s(\mathbf{x}_k)^\top(\mathbf{x} - \mathbf{x}_k) + \frac{1}{2\eta}\|\mathbf{x} - \mathbf{x}_k\|^2$. Rather than linearizing ψ as well, we minimize this quadratic upper bound on s together with the *exact* ψ. Completing the square shows that the minimizer is

$$\operatorname*{argmin}_{\mathbf{x}} \; \psi(\mathbf{x}) + \frac{1}{2\eta}\left\|\mathbf{x} - (\mathbf{x}_k - \eta\nabla s(\mathbf{x}_k))\right\|^2 \;=\; \text{prox}_{\eta\psi}\left(\mathbf{x}_k - \eta\,\nabla s(\mathbf{x}_k)\right),$$

which is an ordinary gradient step on the smooth part followed by a proximal step on the kinked part. Iterating this update,

$$\mathbf{x}_{k+1} = \text{prox}_{\eta\psi}\left(\mathbf{x}_k - \eta\,\nabla s(\mathbf{x}_k)\right), \tag{35.73}$$

gives **proximal gradient descent**, known as **ISTA** (the iterative shrinkage-thresholding algorithm) when $\psi = \lambda\|\cdot\|_1$. It subsumes both projected gradient descent (take ψ to be the indicator of C) and plain gradient descent (take $\psi = 0$).

Even though F is not differentiable, proximal gradient descent with $\eta = 1/L$ achieves the same $O(1/k)$ rate as smooth gradient descent: $F(\mathbf{x}_k) - F^\star \leq L\|\mathbf{x}_0 - \mathbf{x}^\star\|^2/(2k)$ (Beck and Teboulle (2009), Theorem 3.1). Moreover, if s is twice continuously differentiable and μ-strongly convex ($\mu\mathbf{I} \preceq \nabla^2 s$), our PGD contraction proof carries over verbatim, because all it needs is that $\mathbf{x}^\star$ is a fixed point and that $\text{prox}_{\eta\psi}$ is nonexpansive just like Π_C (Exercise 7 proves this).

For the lasso, the proximal step is soft-thresholding with threshold $\eta\lambda$, which snaps every coordinate within $\eta\lambda$ of zero directly to zero. That threshold matches the exact amplitude at which subgradient descent was chattering, so ISTA should hold the null coordinates cleanly at zero. Let us check.

```
w = np.zeros(8)
for k in range(50):                                  # ISTA: gradient step, then prox
    w = soft(w - eta * A.T @ (A @ w - b), eta * lam)
print(f'exact zeros: {int((w == 0).sum())} of 8;  support recovered: '
      f'{np.array_equal(w != 0, w_planted != 0)};  '
      f'max deviation on the support: {np.abs(w - w_planted)[~null].max():.3f}')
```

```
exact zeros: 5 of 8;  support recovered: True;  max deviation on the support: 0.029
```

After the same fifty iterations—at the same cost per step as subgradient descent—ISTA recovers the exact support, setting all five null coordinates to zero and leaving the three active coordinates nonzero. Those three entries sit within about 0.05 of the planted values; they cannot get much closer, because the ℓ_1 penalty shrinks even the true lasso solution toward zero by roughly $\lambda/\|\mathbf{a}_j\|^2 \approx 0.03$.

35.6.4.3 Coordinate Descent

You may skip this subsection on a first reading: coordinate descent is the standard solver for the lasso, though it is not how neural networks are trained.

On the lasso, minimizing along a single coordinate w_j while holding all others fixed reduces to the exact one-dimensional problem we just solved. Let $\mathbf{r}_{-j} = \mathbf{b} - \sum_{i \neq j} \mathbf{a}_i w_i$ be the residual computed without coordinate j. The j-th subproblem, $\frac{1}{2}\|\mathbf{a}_j w_j - \mathbf{r}_{-j}\|^2 + \lambda |w_j|$, equals $\|\mathbf{a}_j\|^2 \left(\frac{1}{2}(w_j - c_j/\|\mathbf{a}_j\|^2)^2 + (\lambda/\|\mathbf{a}_j\|^2)|w_j|\right)$ up to an additive constant, where $c_j = \mathbf{a}_j^\top \mathbf{r}_{-j}$. Soft-thresholding with threshold $\lambda/\|\mathbf{a}_j\|^2$ solves it in closed form:

$$w_j \leftarrow \frac{\operatorname{sign}(c_j)\,\max(|c_j| - \lambda,\, 0)}{\|\mathbf{a}_j\|^2}, \qquad c_j = \mathbf{a}_j^\top \mathbf{r}_{-j}. \tag{35.74}$$

Whenever a feature's correlation c_j with the current residual has magnitude at most λ, its weight is set immediately to zero. Cycling through the coordinates with this update is the algorithm behind the widely used `glmnet` package (Friedman et al. 2010). By updating the residual incrementally after each coordinate change, one full sweep over all d coordinates costs the same as a single gradient evaluation.

The same principle applies to any differentiable objective f with coordinate-wise smoothness constants L_j,

$$f(\mathbf{x} + t\mathbf{e}_j) \leq f(\mathbf{x}) + t\,\partial_j f(\mathbf{x}) + \tfrac{1}{2} L_j t^2, \tag{35.75}$$

where minimizing the quadratic upper bound over t produces the **coordinate-descent** update

$$x_j \leftarrow x_j - \frac{1}{L_j}\partial_j f(\mathbf{x}). \tag{35.76}$$

Because L_j measures curvature along a single axis, it can be substantially smaller than the global smoothness constant L, allowing larger steps: in our lasso problem, $L_j = \|\mathbf{a}_j\|^2 \approx 100$, compared with $L \in [130, 190]$. For a positive-definite quadratic $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{A}\mathbf{x} - \mathbf{b}^\top \mathbf{x}$, the upper bound is exact with $L_j = A_{jj}$, and the update

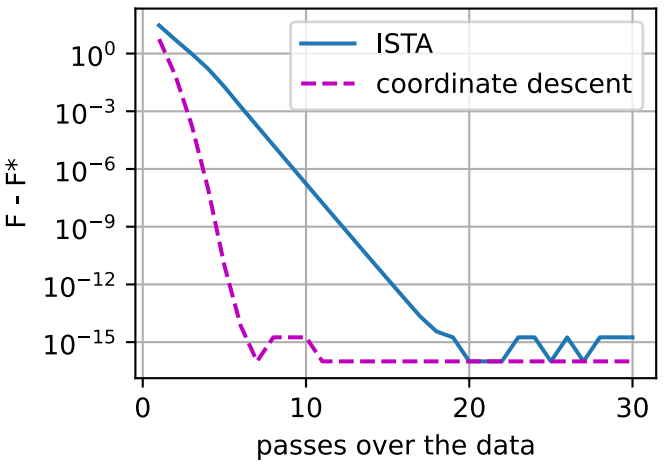
$$x_j \leftarrow x_j - \frac{(\mathbf{A}\mathbf{x} - \mathbf{b})_j}{A_{jj}} \tag{35.77}$$

minimizes f exactly along coordinate j. Sweeping cyclically through all coordinates recovers the classical Gauss–Seidel iteration for $\mathbf{A}\mathbf{x} = \mathbf{b}$.

How much faster is a coordinate sweep than an ISTA step of the same cost? Let us count how many passes over the data each method takes to push the objective gap $F(\mathbf{w}) - F^\star$ below 10^{-8} on our lasso problem.

```
w_ref = np.zeros(8)
for k in range(5000):                              # reference optimum by ISTA
    w_ref = soft(w_ref - eta * A.T @ (A @ w_ref - b), eta * lam)
F_star, col2 = F(w_ref), (A**2).sum(0)             # col2[j] = |a_j|^2
w_i, w_c, r = np.zeros(8), np.zeros(8), b.copy()   # r = b - A w_c, kept up to date
gap_i, gap_c = [], []
for sweep in range(30):
    w_i = soft(w_i - eta * A.T @ (A @ w_i - b), eta * lam)   # one ISTA step
    for j in range(8):                                       # one coordinate sweep
        c_j = A[:, j] @ r + col2[j] * w_c[j]                 # a_j^T r_{-j}
        w_new = soft(c_j, lam) / col2[j]
        r -= A[:, j] * (w_new - w_c[j])
        w_c[j] = w_new
    gap_i.append(F(w_i) - F_star); gap_c.append(F(w_c) - F_star)
passes = lambda gap: next((k + 1 for k, g in enumerate(gap) if g < 1e-8),
                          '> 30')                  # not reached in 30 passes
print(f'passes to gap < 1e-8: ISTA {passes(gap_i)}, coordinate descent {passes(gap_c)}')
d2l.plot(np.arange(1, 31), [np.maximum(gap_i, 1e-16), np.maximum(gap_c, 1e-16)],
         'passes over the data', 'F - F*', legend=['ISTA', 'coordinate descent'],
         yscale='log')                             # gaps floored at rounding level
```

```
passes to gap < 1e-8: ISTA 12, coordinate descent 5
```



On the logarithmic plot, both methods trace straight lines—linear convergence—but coordinate descent drops much more steeply, typically reaching a gap of 10^{-8} in four to six sweeps compared with a dozen to twenty-five steps for ISTA. Nothing we proved in this section predicts that exact speedup. The sharpest rates for coordinate descent are proved for randomly chosen coordinates (Nesterov 2012); cyclic sweeps like ours also converge at a guaranteed rate, but with weaker constants (Beck and Tetruashvili 2013).

Coordinate descent does have an important limitation. If a nonsmooth kink does not separate across coordinates, coordinate descent can get stuck far from the optimum: for $f(x, y) = \max(|x|, |y|)$ at $(1, 1)$, moving along x alone or along y alone cannot decrease f, even though the global minimum 0 sits at the origin. Coordinate descent succeeds on the lasso specifically because its nonsmooth penalty $\lambda \sum_j |w_j|$ is separable across coordinates (Tseng 2001). For sparse linear models and block-alternating minimization, it remains the method of choice. For training dense neural networks, however, it plays no role, because modern accelerators are engineered for parallel matrix multiplications rather than sequential scalar updates.

Notice one pattern running through everything we did in this section: every time we wanted a multiplier, we first solved for $\mathbf{x}$. In Section 35.7, we flip the question around and ask how to solve for the multipliers *directly*—and how any choice of multipliers provides a certificate proving that no feasible point can beat a given lower bound. Our clipping example, with $\lambda^\star = 4$ and $p^\star = 8$, will be its opening case.

35.6.5 Summary

At a constrained optimum, the gradient need not vanish; instead, no feasible direction can decrease the objective. For a single equality constraint with $\nabla h(\mathbf{x}^\star) \neq \mathbf{0}$, this means $\nabla f + \nu \nabla h = \mathbf{0}$ for a Lagrange multiplier ν that measures how hard the solution presses against the constraint. Adding inequality constraints leads to the four KKT conditions, where dual feasibility ($\lambda_i \geq 0$) enforces one-way walls and complementary slackness ($\lambda_i g_i = 0$) zeroes out the multiplier of every inactive constraint. For convex problems, every KKT point is a global minimum, and under a constraint qualification (such as LICQ or Slater's condition) every minimum is a KKT point.

To compute constrained solutions iteratively, projected gradient descent follows each gradient step with a Euclidean projection Π_C. Because projection onto a closed convex set is unique, characterized by the obtuse-angle inequality, and nonexpansive, projected gradient descent preserves the convergence rate of unconstrained gradient descent. Projections onto a ball, a half-space, and the probability simplex each reduce to solving for a single scalar multiplier. Finally, the proximal operator generalizes projection from sets to nonsmooth functions: soft-thresholding is the proximal operator of $\lambda|\cdot|$, and both ISTA and coordinate descent use it to solve the lasso with exact zeros that subgradient descent cannot attain.

35.6.6 Exercises

1. **Degenerate constraints.** The Lagrange condition Equation 35.61 and the KKT conditions Equation 35.65 hold at a minimum only under a constraint qualification. Each of the following problems violates one.
    1. Minimize $f(\mathbf{x}) = x_1 + x_2^2$ over $\mathbb{R}^3$ subject to $h_1(\mathbf{x}) = x_3 = 0$ and $h_2(\mathbf{x}) = x_3 - x_1^2 = 0$. Show that the feasible set is a line, find the minimizer, and show that both constraint gradients are nonzero everywhere. Show that no multipliers ν_1, ν_2 satisfy $\nabla f + \nu_1 \nabla h_1 + \nu_2 \nabla h_2 = \mathbf{0}$ at the minimizer. Describe the same line by two other equality constraints for which multipliers exist. Is a constraint qualification a property of the feasible set or of the functions that describe it?
    2. Minimize $f(x) = x$ over $\mathbb{R}$ subject to $x^2 \leq 0$. Find $p^\star$ and show that no $\lambda \geq 0$ satisfies the KKT stationarity condition at the minimizer. Which of LICQ and

Slater's condition fails? Explain why for each.

3. Minimize $f(x) = x$ over $\mathbb{R}$ subject to the constraint $-x \le 0$ written twice, $g_1(x) = g_2(x) = -x$. Show that LICQ fails and Slater's condition holds. Find all pairs $(\lambda_1^\star, \lambda_2^\star)$ that satisfy the KKT conditions at the minimizer. Under the relaxed constraints $-x \le u_1$ and $-x \le u_2$, compute the optimal value $p^\star(u_1, u_2)$, and verify that every KKT pair satisfies $p^\star(\mathbf{u}) \ge p^\star(\mathbf{0}) - \lambda^{\star\top}\mathbf{u}$ for all $\mathbf{u}$. Is $p^\star$ differentiable at $\mathbf{0}$, and what takes the place of the relation $\lambda^\star = -\nabla p^\star(\mathbf{0})$?

2. **Projected fixed points.** For a closed convex set C, the fixed points of projected gradient descent Equation 35.68 are the constrained stationary points, whatever the step size $\eta > 0$.

   1. Let $C = \{\mathbf{x} : \mathbf{x} \ge 0\}$. Show that $\mathbf{x}^\star$ is a fixed point if and only if, for every coordinate i, either $\partial f/\partial x_i(\mathbf{x}^\star) = 0$ and $x_i^\star \ge 0$, or $\partial f/\partial x_i(\mathbf{x}^\star) > 0$ and $x_i^\star = 0$. Show that these conditions are the KKT conditions for the constraints $-x_i \le 0$, and express the multipliers through $\nabla f(\mathbf{x}^\star)$.
   2. Gradient clipping updates $\mathbf{x}_{k+1} = \mathbf{x}_k - \eta\,\Pi_{B_c}(\nabla f(\mathbf{x}_k))$, where $B_c = \{\mathbf{z} : \|\mathbf{z}\| \le c\}$. Find the fixed points of this update and compare them with those of projected gradient descent onto the parameter ball $\{\mathbf{x} : \|\mathbf{x}\| \le r\}$. Which of the two updates solves a constrained problem?
   3. Binary weights restrict a parameter to the nonconvex set $C = \{-1, 1\}$, and projecting onto C rounds to the nearer element (to 1 at a tie). Let $f(x) = \frac{1}{2}(x - a)^2$ with $0 < a < 1$. For which step sizes is $x = 1$ a fixed point of Equation 35.68, and for which is $x = -1$ one? What do the iterates do when η exceeds both ranges? Which point minimizes f over C? Does either point satisfy $f'(x^\star)(x - x^\star) \ge 0$ for all $x \in C$? Which step of the fixed-point argument fails for this set?
   4. Run the PGD cell on the half-space with $\eta = 2.2/1000$ instead of 0.0015. The projection is nonexpansive; why does the run still diverge? Which factor of the contraction proposition exceeds 1?

3. code **Sparsemax and the ℓ_1 ball.** The function `project_simplex` computes the simplex projection Equation 35.70, which is sparsemax; its first output is the projected vector.

   1. Show that adding the same constant to every score leaves the projection unchanged. Divide the scores by a temperature $T > 0$, and let $u_1 > u_2$ be the two largest entries of $\mathbf{y}$. Show that the projection of $\mathbf{y}/T$ is one-hot exactly when $T \le u_1 - u_2$, and find its limit as $T \to \infty$. Plot the number of nonzero entries of the projection of $\mathbf{y}/T$ against T for ten standard normal scores.
   2. Let S be the support of $\mathbf{x}^\star$, and suppose that no y_i equals the threshold τ. Derive the Jacobian $\partial\mathbf{x}^\star/\partial\mathbf{y}$ from Equation 35.70 and check it against central finite differences. How many arithmetic operations does the backward pass through sparsemax need for an input of length n?
   3. Show that the projection onto the ℓ_1 ball $\{\mathbf{x} : \|\mathbf{x}\|_1 \le z\}$ keeps the signs of $\mathbf{y}$ and, when $\|\mathbf{y}\|_1 > z$, equals $\operatorname{sign}(\mathbf{y}) \odot z\,\Pi_\Delta(|\mathbf{y}|/z)$ (Duchi et al. 2008). Implement it with `project_simplex`. Certify the result with the variational inequality $(\mathbf{y} - \hat{\mathbf{x}})^\top(\mathbf{x} - \hat{\mathbf{x}}) \le 0$, checked at the $2n$ points $\pm z\,\mathbf{e}_i$. Why do these points suffice?

4. **Second-order conditions on the sphere.** ● The Lagrange condition is a first-order condition and cannot tell minima from saddles. A second-order test must account for the curvature of the constraint.

   1. Let f and h be twice continuously differentiable, and let $\mathbf{x}^\star$ satisfy Equation 35.61

with multiplier $\nu^\star$. For every curve $\mathbf{x}(t)$ on $\{h = 0\}$ with $\mathbf{x}(0) = \mathbf{x}^\star$ and $\dot{\mathbf{x}}(0) = \mathbf{v}$, show that

$$\left.\frac{d^2}{dt^2} f(\mathbf{x}(t))\right|_{t=0} = \mathbf{v}^\top \nabla^2_{\mathbf{x}} \mathcal{L}(\mathbf{x}^\star, \nu^\star)\, \mathbf{v},$$

with the Lagrangian Equation 35.62. Conclude that $\mathbf{v}^\top \nabla^2_{\mathbf{x}} \mathcal{L}\, \mathbf{v} \geq 0$ for every tangent direction $\mathbf{v}$ at a local minimum. Why is the Hessian of f alone the wrong matrix for this test? Conversely, positive definiteness on the tangent space implies a strict local minimum (Nocedal and Wright (2006), chapter 12).

2. Let $\mathbf{A}$ be symmetric with distinct eigenvalues $a_1 < a_2 < \cdots < a_n$ and unit eigenvectors $\mathbf{q}_1, \dots, \mathbf{q}_n$, and minimize $f(\mathbf{x}) = \mathbf{x}^\top \mathbf{A} \mathbf{x}$ subject to $\mathbf{x}^\top \mathbf{x} = 1$. Show that the Lagrange points are $\pm \mathbf{q}_i$ with $\nu^\star = -a_i$, and find the eigenvalues of $\nabla^2_{\mathbf{x}} \mathcal{L}$ on the tangent space at $\mathbf{q}_i$. Which Lagrange points are local minima, which are local maxima, and which are saddles? How many independent descent directions does $\mathbf{q}_i$ have?

3. Projected gradient descent onto the sphere, $\mathbf{x} \leftarrow (\mathbf{x} - \eta \nabla f(\mathbf{x}))/\|\mathbf{x} - \eta \nabla f(\mathbf{x})\|$, projects onto a nonconvex set. Show that for this f it is the power iteration for $\mathbf{I} - 2\eta \mathbf{A}$. Assume $a_1 + a_n > 0$. From a random start, what do the iterates $\mathbf{x}_k$ and the values $f(\mathbf{x}_k)$ do? How does the answer depend on $\eta > 0$?

5. **Block soft-thresholding (the group lasso).** The penalty $\lambda \|\mathbf{w}\|_2$ on a whole vector is kinked only at $\mathbf{w} = \mathbf{0}$. Its subgradients are $\mathbf{w}/\|\mathbf{w}\|$ for $\mathbf{w} \neq \mathbf{0}$ and every vector of the unit ball at $\mathbf{0}$ (times λ).

   1. Show that $\operatorname{prox}_{\lambda \|\cdot\|_2}(\mathbf{z}) = \max(0,\ 1 - \lambda/\|\mathbf{z}\|)\, \mathbf{z}$.
   2. Compare the zero pattern with that of coordinatewise soft-thresholding Equation 35.71. When does the block prox return exactly $\mathbf{0}$, and can it zero a single coordinate of a block?
   3. What does the penalty $\lambda \sum_u \|\mathbf{w}_u\|_2$, summed over the vectors $\mathbf{w}_u$ of outgoing weights of the hidden units u of a network, do to the network?
   4. code Verify (a) against a brute-force minimization on a 2-D grid for $\mathbf{z} = (0.3, 0.4)$ and $\mathbf{z} = (3, 4)$ with $\lambda = 1$. The answers are $\mathbf{0}$ and $(2.4, 3.2)$; where has $(2.4, 3.2)$ appeared in this section?

6. **Moreau decomposition.** ● The *convex conjugate* of ψ is $\psi^*(\mathbf{y}) = \sup_{\mathbf{x}} (\mathbf{y}^\top \mathbf{x} - \psi(\mathbf{x}))$. For a proper closed convex ψ, two facts connect it with the proximal operator Equation 35.72: $\mathbf{x} = \operatorname{prox}_{\psi}(\mathbf{z})$ exactly when $\mathbf{z} - \mathbf{x} \in \partial\psi(\mathbf{x})$ (the set of subgradients), and $\mathbf{y} \in \partial\psi(\mathbf{x})$ exactly when $\mathbf{x} \in \partial\psi^*(\mathbf{y})$ (Parikh and Boyd 2014).

   1. Derive the Moreau decomposition $\mathbf{z} = \operatorname{prox}_{\psi}(\mathbf{z}) + \operatorname{prox}_{\psi^*}(\mathbf{z})$.
   2. The conjugate of $\psi = \lambda \|\cdot\|_1$ is the indicator of the box $\{\mathbf{y} : \|\mathbf{y}\|_\infty \leq \lambda\}$. Show that the decomposition writes soft-thresholding Equation 35.71 as $\mathbf{z}$ minus a clipped copy of $\mathbf{z}$.
   3. The conjugate of $\psi = \lambda \|\cdot\|_2$ is the indicator of the ℓ_2 ball of radius λ, whose prox is the projection onto the ball, the clip-by-norm map of gradient clipping. Use the decomposition to recover the block soft-thresholding formula of the group-lasso exercise from the clipping formula. For $\mathbf{z} = (3, 4)$ and $\lambda = 1$, which two vectors of this section add up to $\mathbf{z}$?

7. code **Accelerated proximal gradient.** ● FISTA (Beck and Teboulle 2009) adds momentum to the ISTA update Equation 35.73 for $F = s + \psi$. With $\mathbf{v}_1 = \mathbf{x}_0$ and $t_1 = 1$, it iterates for $k \geq 1$

$$\begin{aligned} \mathbf{x}_k &= \operatorname{prox}_{\eta\psi}(\mathbf{v}_k - \eta \nabla s(\mathbf{v}_k)), \\ t_{k+1} &= \tfrac{1}{2}\Big(1 + \sqrt{1 + 4t_k^2}\Big), \\ \mathbf{v}_{k+1} &= \mathbf{x}_k + \frac{t_k - 1}{t_{k+1}} (\mathbf{x}_k - \mathbf{x}_{k-1}). \end{aligned} \tag{35.78}$$

With $\eta = 1/L$ its objective gap satisfies $F(\mathbf{x}_k) - F^\star \leq 2L\|\mathbf{x}_0 - \mathbf{x}^\star\|^2/(k+1)^2$, against $O(1/k)$ for ISTA.

1. Show that the prox is nonexpansive, $\|\operatorname{prox}_{\eta\psi}(\mathbf{u}) - \operatorname{prox}_{\eta\psi}(\mathbf{v})\| \leq \|\mathbf{u} - \mathbf{v}\|$, by adding the two subgradient inequalities that characterize $\operatorname{prox}_{\eta\psi}(\mathbf{u})$ and $\operatorname{prox}_{\eta\psi}(\mathbf{v})$, as in the proof for projections.
2. Add FISTA to the ISTA cell, on the same lasso, and plot $F(\mathbf{x}_k) - F^\star$ of both methods over 100 steps on a logarithmic axis, with $F^\star$ taken from 10^4 ISTA steps. Which method is ahead after 10 steps, and which after 30? Explain the straight line of ISTA with the PGD contraction proposition and part (a).
3. Is $F(\mathbf{x}_k)$ monotone along the FISTA iterates?
4. Replace $\mathbf{A}$ by a 30×100 Gaussian matrix, plant the same three nonzero coefficients, and run both methods for 1000 steps. Over which range of k is FISTA ahead now? Why does ISTA still end with a fast linear phase, although $\mathbf{A}^\top\mathbf{A}$ is singular?

35.7 Duality

When an iterative solver stops making progress and reports convergence, a natural question remains: is the point it returned actually optimal? For an unconstrained convex function, a zero gradient settles the question at once (Section 35.3). With constraints, however, the optimum usually sits on the boundary with a nonzero gradient (Section 35.6), so that simple test disappears. Both numerical solvers and mathematical proofs therefore need a different tool: a reliable lower bound on the optimal value. In this section we develop a systematic way to construct such bounds and uncover what the Lagrange multipliers of the KKT conditions actually measure.

To see the difficulty concretely, return to the clipping problem of Section 35.6.2. There we minimized the squared distance $\frac{1}{2}\|\mathbf{x} - \mathbf{x}_0\|^2$ to the target $\mathbf{x}_0 = (3, 4)$ over the unit disk $\|\mathbf{x}\| \leq 1$, finding the optimum $\mathbf{x}^\star = (0.6, 0.8)$ with optimal value 8. (We write $\mathbf{x}_0$ here instead of $\mathbf{y}$, reserving $\mathbf{y}$ for labels and for the argument of the convex conjugate later.) Now imagine an iterative solver stops at $\mathbf{x} = (0.55, 0.835)$. This point is feasible, since $\|\mathbf{x}\|^2 = 0.9997 \leq 1$, and achieves an objective value of 8.01. How can we tell whether 8.01 is nearly optimal, or whether some other point in the disk reaches 7? The gradient at the true optimum is $(-2.4, -3.2)$, nowhere near zero, and the disk contains infinitely many points, so checking them one by one is out of the question.

The key idea is to fold the constraint into the objective. Add a nonnegative multiple $\lambda \geq 0$ of the constraint function $\frac{1}{2}(\|\mathbf{x}\|^2 - 1)$ to the objective. At every feasible point this added

term is nonpositive, so minimizing the modified objective over *all* of $\mathbb{R}^2$ must yield a value no larger than the constrained optimum. Each choice of $\lambda \geq 0$ thus produces a lower bound through an unconstrained minimization, and $\lambda = 4$ gives a bound of 8—proving immediately that the solver's 8.01 is within 0.01 of optimal. Searching for the tightest such bound defines a companion optimization problem, the *dual problem*. Remarkably, the dual problem is always convex, even when the original problem is not. For a convex problem with a strictly feasible point, its optimal value matches the constrained optimum, and its optimal multipliers tell us how rapidly the optimal value improves when each constraint is relaxed.

In what follows, we first use this bounding idea to prove optimality for the clipping problem and state *weak duality*, a two-line argument that holds for every optimization problem. We then establish *strong duality* for convex problems, whose proof contains the one subtle geometric step in the section, and interpret the optimal multipliers as *shadow prices*, connecting weight decay directly to norm constraints. After introducing the *convex conjugate* as a convenient tool for computing duals of linearly constrained problems (which can be skimmed on a first reading), we work through three revealing examples: the support vector machine, water-filling power allocation, and a nonconvex problem whose best dual bound stays strictly below its optimum. All computations are small enough for NumPy, and `d2l` handles the plots.

35.7.1 The Dual Problem

Our first goal is to prove that no feasible point of the clipping problem achieves a value below 8, without checking feasible points one by one. Every lower bound in this section rests on a single elementary observation: at any feasible point, every inequality constraint function is nonpositive.

35.7.1.1 A Lower Bound from One Multiplier

Recall the clipping problem: minimize $f(\mathbf{x}) = \frac{1}{2}\|\mathbf{x} - \mathbf{x}_0\|^2$ with $\mathbf{x}_0 = (3, 4)$, subject to $g(\mathbf{x}) = \frac{1}{2}(\|\mathbf{x}\|^2 - 1) \leq 0$. Including the factor of $\frac{1}{2}$ in g simply makes its gradient equal to $\mathbf{x}$. From the KKT conditions we already know the solution: $\mathbf{x}^\star = \mathbf{x}_0/\|\mathbf{x}_0\| = (0.6, 0.8)$ with optimal value $p^\star = \frac{1}{2}(5-1)^2 = 8$, and optimal multiplier $\lambda^\star = 4$, determined by the stationarity condition $\nabla f(\mathbf{x}^\star) + \lambda^\star \mathbf{x}^\star = (-2.4, -3.2) + 4\,(0.6, 0.8) = \mathbf{0}$.

Now pick any multiplier $\lambda \geq 0$ and any feasible point $\mathbf{x}$. Because $g(\mathbf{x}) \leq 0$ and $\lambda \geq 0$, their product $\lambda g(\mathbf{x})$ is nonpositive, which means $f(\mathbf{x}) + \lambda g(\mathbf{x}) \leq f(\mathbf{x})$. Minimizing the left-hand side over all of $\mathbb{R}^2$ can only make it smaller still, so that unconstrained minimum is at most $f(\mathbf{x})$. Since this holds for every feasible $\mathbf{x}$, it also holds for the best feasible $\mathbf{x}$:

$$\min_{\mathbf{x} \in \mathbb{R}^2} \big[f(\mathbf{x}) + \lambda g(\mathbf{x}) \big] \;\leq\; \min_{\|\mathbf{x}\| \leq 1} f(\mathbf{x}) \;=\; p^\star \qquad \text{for every } \lambda \geq 0.$$

Let us evaluate the left-hand side explicitly. The function being minimized is the Lagrangian $\mathcal{L}(\mathbf{x}, \lambda) = \frac{1}{2}\|\mathbf{x} - \mathbf{x}_0\|^2 + \frac{\lambda}{2}(\|\mathbf{x}\|^2 - 1)$. For any $\lambda \geq 0$, this is a convex quadratic in $\mathbf{x}$ with Hessian $(1+\lambda)\mathbf{I}$, so its unique minimizer occurs where the gradient $(\mathbf{x} - \mathbf{x}_0) + \lambda \mathbf{x}$ vanishes: $\mathbf{x}(\lambda) = \mathbf{x}_0/(1+\lambda)$. Increasing λ shrinks $\mathbf{x}_0$ more strongly toward the origin. Substituting $\mathbf{x}(\lambda)$ back into the Lagrangian—using $\mathbf{x}(\lambda) - \mathbf{x}_0 = -\lambda \mathbf{x}_0/(1+\lambda)$ and $\|\mathbf{x}_0\|^2 = 25$—gives

$\frac{1}{2}\|\mathbf{x}(\lambda) - \mathbf{x}_0\|^2 = \frac{25}{2}\lambda^2/(1+\lambda)^2$ and $\frac{\lambda}{2}\|\mathbf{x}(\lambda)\|^2 = \frac{25}{2}\lambda/(1+\lambda)^2$. Combining these two fractions cancels a factor of $(1+\lambda)$, leaving the minimum value as a simple function of λ:

$$q(\lambda) = \frac{25}{2}\frac{\lambda}{1+\lambda} - \frac{\lambda}{2}. \tag{35.79}$$

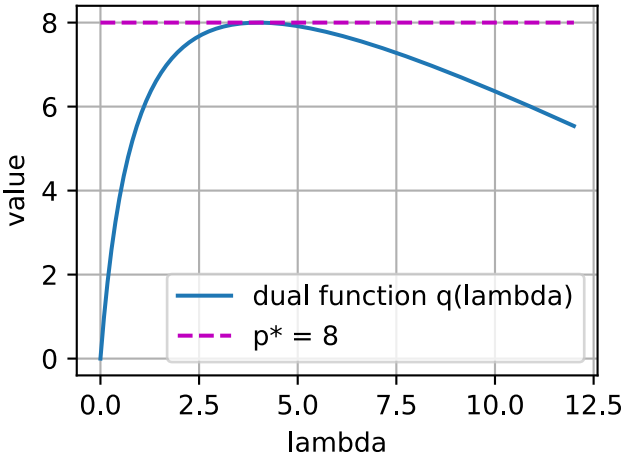
The two terms pull in opposite directions. As λ grows, the first term rises toward 12.5, while the second term—coming from the constant $-\frac{1}{2}$ inside g—drops linearly without bound. The tightest lower bound sits where these two effects balance.

Plugging in a few numbers makes this balance concrete: $q(0) = 0$, $q(1) = 5.75$, $q(2) = 7.33$, $q(3) = 7.875$, $q(4) = 8$, $q(5) = 7.92$, and $q(8) = 7.11$. Every single one of these values is a valid lower bound on $p^\star$. Differentiating gives $q'(\lambda) = \frac{25}{2(1+\lambda)^2} - \frac{1}{2}$, which vanishes when $(1+\lambda)^2 = 25$, or $\lambda = 4$. Since $q''(\lambda) = -25/(1+\lambda)^3 < 0$, $\lambda = 4$ is the unique maximum, attaining $q(4) = 10 - 2 = 8 = p^\star$. The best lower bound equals the true optimum, and the multiplier that achieves it is the KKT multiplier. This certifies that the solver's value of 8.01 is within 0.01 of optimal; even an unoptimized guess like $\lambda = 3$ already certifies that it is within 0.135.

Why does $\lambda = 4$ hit the optimum? Look at the constraint value at the Lagrangian minimizer: $g(\mathbf{x}(\lambda)) = \frac{1}{2}(25/(1+\lambda)^2 - 1)$. When $\lambda < 4$, the multiplier is too weak to pull the minimizer inside the unit disk (for example, $\mathbf{x}(1) = (1.5, 2)$ has norm 2.5), so $g(\mathbf{x}(\lambda)) > 0$ and the bound reflects an infeasible point. When $\lambda > 4$, the multiplier pulls the minimizer too far inside (for example, $\mathbf{x}(8)$ has norm $\frac{5}{9}$), so $\lambda g(\mathbf{x}(\lambda)) < 0$ drags the bound back down. Right at $\lambda = 4$, the minimizer $\mathbf{x}(4) = (0.6, 0.8)$ lands on the boundary circle, the penalty term $\lambda g(\mathbf{x}(\lambda))$ vanishes, and the bound coincides with the objective of a feasible point.

Let us plot $q(\lambda)$ alongside $p^\star$ across a grid of λ values to see how these bounds behave.

```
x0 = np.array([3.0, 4.0])                          # the point to clip
f = lambda x: 0.5 * np.sum((x - x0) ** 2)          # objective
g = lambda x: 0.5 * (np.sum(x ** 2) - 1.0)         # constraint g(x) <= 0
x_of = lambda lam: x0 / (1 + lam)                  # minimizer of L(., lam)
q = lambda lam: f(x_of(lam)) + lam * g(x_of(lam))  # dual function
p_star = f(x0 / np.linalg.norm(x0))                # value at the clipped point
lams = np.linspace(0.0, 12.0, 1201)
qs = np.array([q(l) for l in lams])
d2l.plot(lams, [qs, np.full_like(lams, p_star)], 'lambda', 'value',
         legend=['dual function q(lambda)', 'p* = 8'])
```



The resulting curve is concave and stays everywhere below the horizontal line $p^\star = 8$, touching it once at $\lambda = 4$. Surprisingly, concavity here has nothing to do with the geometry of the disk: as we now show, the dual function is concave for *every* optimization problem.

35.7.1.2 The Lagrange Dual Function

Nothing in the bounding argument above depended on the clipping problem. Let us apply the same construction to the general constrained problem from Section 35.6.2,

$$\min_{\mathbf{x}\in\mathcal{D}} \; f(\mathbf{x}) \quad \text{subject to} \quad g_i(\mathbf{x}) \le 0,\; i = 1,\dots,m, \qquad h_j(\mathbf{x}) = 0,\; j = 1,\dots,p,$$

with optimal value $p^\star$, defined as the infimum of f over all feasible points (with $p^\star = +\infty$ if no feasible point exists). Here $\mathcal{D}$ is the domain on which f and all constraint functions are defined; it is simply $\mathbb{R}^n$ unless we keep a set as a hard constraint, as we will do with $[0,1]$ in the duality-gap example at the end of this section. The Lagrangian introduces one multiplier per constraint,

$$\mathcal{L}(\mathbf{x},\lambda,\nu) = f(\mathbf{x}) + \sum_{i=1}^{m} \lambda_i\, g_i(\mathbf{x}) + \sum_{j=1}^{p} \nu_j\, h_j(\mathbf{x}),$$

with $\lambda_i \ge 0$ for inequalities and $\nu_j \in \mathbb{R}$ unconstrained in sign for equalities. We write $\lambda \succeq 0$ to mean $\lambda_i \ge 0$ for every coordinate i.

Minimizing the Lagrangian over $\mathbf{x}$ for fixed multipliers turns our lower-bound construction into a function of (λ,ν). The **Lagrange dual function** is

$$q(\lambda,\nu) = \inf_{\mathbf{x}\in\mathcal{D}} \; \mathcal{L}(\mathbf{x},\lambda,\nu). \tag{35.80}$$

In words, $q(\lambda,\nu)$ asks how low f can go once we drop the hard constraints and replace them with linear penalties weighted by λ and ν. For the clipping problem this gave Equation 35.79. In general, the infimum need not be attained, and it can even be $-\infty$. For example, suppose we minimize x subject to $-x \le 0$: the Lagrangian $(1-\lambda)x$ is unbounded below unless

$\lambda = 1$, so $q(1) = 0$ and $q(\lambda) = -\infty$ for all other λ. A bound of $-\infty$ is trivially valid but useless. The set of multipliers where $q(\lambda, \nu) > -\infty$ is called the *effective domain* of q; when we derive the support vector machine dual below, this implicit domain restriction will surface as an explicit equality constraint.

Two fundamental properties of q hold with no assumptions whatsoever on f, g_i, or h_j—not even convexity or differentiability.

Proposition (the dual function is concave). *For any primal problem, $q(\lambda, \nu)$ in Equation 35.80 is concave.*

Proof. Fix any $\mathbf{x} \in \mathcal{D}$. Because the multipliers enter $\mathcal{L}$ linearly, the map $(\lambda, \nu) \mapsto \mathcal{L}(\mathbf{x}, \lambda, \nu)$ is affine. The dual function q is the pointwise infimum of this family of affine functions. Its **hypograph**—the set of points lying on or below its graph—is therefore the intersection of the half-spaces below these affine functions. Since any intersection of convex sets is convex, q is concave. ■

Figure 35.25 draws this construction for the clipping problem. Each dashed line is the Lagrangian of one fixed $\mathbf{x}$ as a function of λ, and q is the curve that stays below all of them. The objective of the clipping problem happens to be convex, but the argument never used that: a lower envelope of lines is concave whatever shape f and g have.

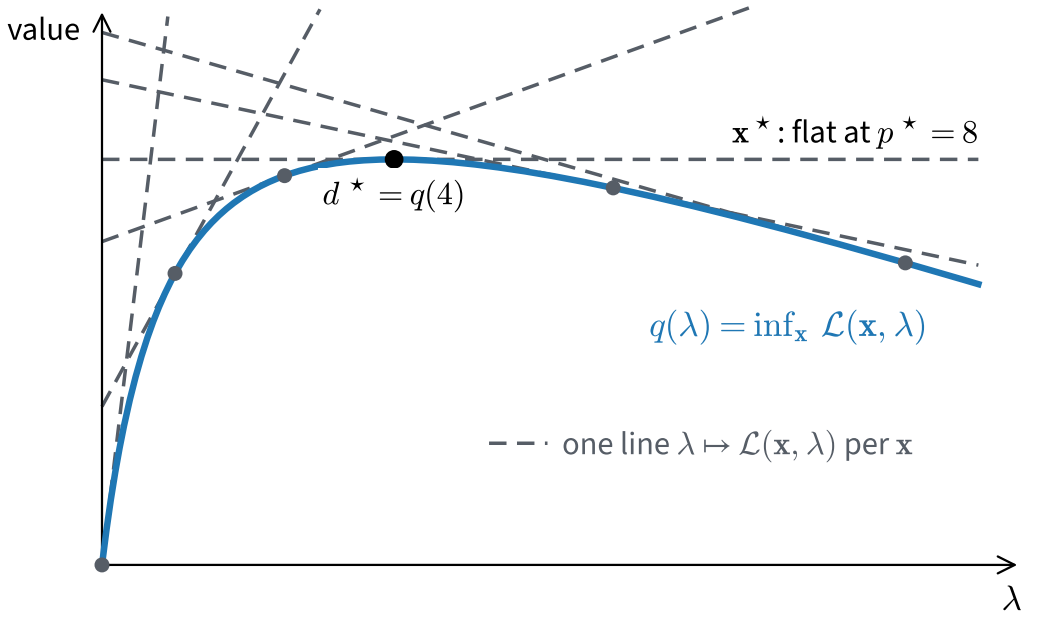


Figure 35.25: The dual function of the clipping problem as a lower envelope. Each fixed $\mathbf{x}$ contributes one line $\lambda \mapsto \mathcal{L}(\mathbf{x}, \lambda)$ (dashed), and $q(\lambda)$ is the lowest value that any of these lines takes at λ. The lines drawn belong to the minimizers $\mathbf{x}(\lambda) = \mathbf{x}_0/(1+\lambda)$ for several values of λ, and each touches q at its own λ (dots). The line of $\mathbf{x}^\star = \mathbf{x}(4)$ is flat because $g(\mathbf{x}^\star) = 0$; it touches q at the maximum $d^\star = q(4) = p^\star = 8$.

Proposition (weak duality). *If $\mathbf{x}$ is feasible and $\lambda \succeq 0$, then $q(\lambda, \nu) \le f(\mathbf{x})$. Consequently,*

$$d^\star \;=\; \sup_{\lambda \succeq 0,\, \nu} q(\lambda, \nu) \;\leq\; \inf_{\mathbf{x} \text{ feasible}} f(\mathbf{x}) \;=\; p^\star. \tag{35.81}$$

Proof. For any feasible $\mathbf{x}$ and any $\lambda \succeq 0$, we have $\lambda_i g_i(\mathbf{x}) \leq 0$ (since $\lambda_i \geq 0$ and $g_i(\mathbf{x}) \leq 0$) and $\nu_j h_j(\mathbf{x}) = 0$. Therefore,

$$q(\lambda, \nu) \;\leq\; \mathcal{L}(\mathbf{x}, \lambda, \nu) \;=\; f(\mathbf{x}) + \sum_i \lambda_i g_i(\mathbf{x}) + \sum_j \nu_j h_j(\mathbf{x}) \;\leq\; f(\mathbf{x}).$$

Taking the supremum over (λ, ν) with $\lambda \succeq 0$ on the left and the infimum over feasible $\mathbf{x}$ on the right yields Equation 35.81. ■

Weak duality is the two-line inequality that turned every $q(\lambda)$ in the clipping problem into a certified lower bound. It is also the exact engine behind KKT sufficiency in Section 35.6.2, where evaluating the same chain at the KKT multipliers makes both inequalities tight via complementary slackness.

Maximizing $q(\lambda, \nu)$ over $\lambda \succeq 0$ and ν is called the **dual problem**. Because it maximizes a concave function over a convex set, the dual problem is *always* a convex optimization problem, even when the primal problem is badly nonconvex. Its optimal value $d^\star$ is the tightest lower bound this construction can provide, and every dual-feasible (λ, ν) gives a valid lower bound. The nonnegative difference $p^\star - d^\star \geq 0$ is the **duality gap**. For the clipping problem the gap was zero, whereas our final worked example will exhibit a strict gap. The natural next question is when we can guarantee in advance that the gap vanishes.

35.7.2 Strong Duality

In the clipping problem, the best dual bound matched the primal optimum exactly. For a nonconvex problem, however, the dual bound can fall strictly short, certifying only part of the answer. Fortunately, two mild ingredients— convexity and a single strictly feasible point—are enough to close the gap. A simple geometric view of the attainable constraint and objective values shows why both are needed.

35.7.2.1 Slater's Condition

Proposition (Slater's condition implies strong duality). *Suppose the problem is convex (f and every g_i convex, every h_j affine) and satisfies Slater's condition (Section 35.6.2.3): there exists a feasible point $\bar{\mathbf{x}}$ in the relative interior of $\mathcal{D}$ such that $g_i(\bar{\mathbf{x}}) < 0$ for every inequality constraint g_i that is not affine. Then $d^\star = p^\star$, and whenever $p^\star$ is finite, the dual optimum is attained.*

This is the same constraint qualification introduced for convex problems in Section 35.6.2: at least one point must sit strictly inside all curved inequality constraints. (See Boyd and Vandenberghe (2004), section 5.3.2 for the full proof in multiple dimensions.) The clipping problem satisfies Slater's condition effortlessly: both f and g are convex, and the origin $\mathbf{0}$ satisfies $g(\mathbf{0}) = -\frac{1}{2} < 0$. Thus $q(4) = p^\star$ was guaranteed before we ever computed a derivative.

To understand *why* convexity and strict feasibility close the gap, consider a problem with a single inequality constraint $g(\mathbf{x}) \leq 0$. Map every point $\mathbf{x} \in \mathcal{D}$ to the pair $(g(\mathbf{x}), f(\mathbf{x}))$ of its constraint value and objective value, and collect all points lying above and to the right of these pairs:

$$\mathcal{A} = \{(u,t) : u \geq g(\mathbf{x}),\ t \geq f(\mathbf{x}) \text{ for some } \mathbf{x} \in \mathcal{D}\}.$$

In this (u,t)-plane, feasible points correspond to $u \leq 0$, so the primal optimum $p^\star$ is simply the lowest height t that $\mathcal{A}$ reaches on the vertical axis $u = 0$. Now fix $\lambda \geq 0$. Over all (u,t) contributed by a given $\mathbf{x}$, the linear function $t + \lambda u$ is minimized at $(g(\mathbf{x}), f(\mathbf{x}))$, so $q(\lambda) = \inf_{(u,t)\in\mathcal{A}}(t+\lambda u)$. Geometrically, we take a line $t+\lambda u = c$ of slope $-\lambda$ and push it upward from below until it just supports $\mathcal{A}$; its intercept on the vertical axis $u = 0$ is $q(\lambda)$. Because the entire set $\mathcal{A}$ lies on or above this supporting line, the intercept $q(\lambda)$ can never exceed $p^\star$. This is weak duality made visible (Figure 35.26).

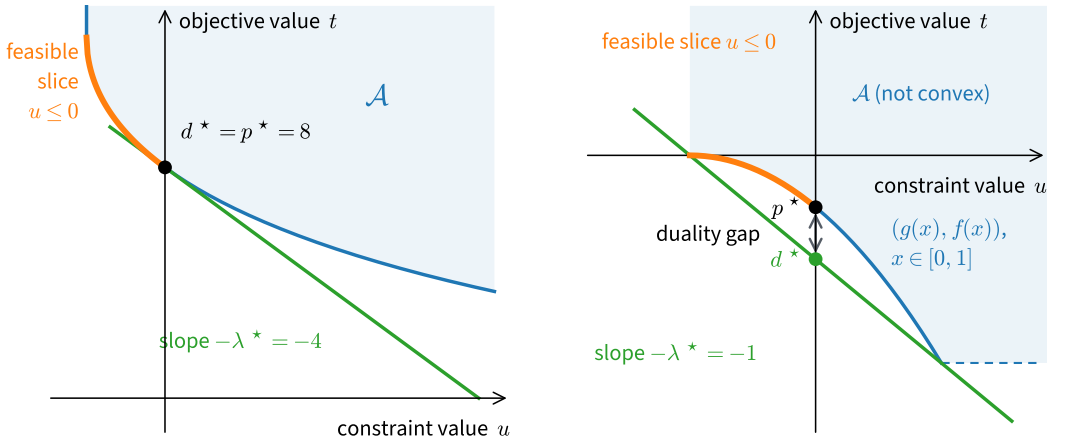


Figure 35.26: The supporting-line geometry of duality. Every $\mathbf{x}$ maps to $(g(\mathbf{x}), f(\mathbf{x}))$; the shaded set $\mathcal{A}$ contains these pairs and all points above and to the right of them. Evaluating the dual at λ pushes a line of slope $-\lambda$ up until it supports $\mathcal{A}$ from below, and its intercept at $u = 0$ is $q(\lambda)$. (a) The clipping problem: $\mathcal{A}$ is convex and $\mathbf{0}$ is strictly feasible, so the optimal supporting line touches at $(0, p^\star)$ with slope -4, giving $d^\star = p^\star = 8$. (b) Minimizing $-x^2$ on $[0,1]$ subject to $x \leq \frac{1}{2}$: $\mathcal{A}$ is nonconvex, so the highest supporting line passes strictly below $(0, p^\star)$, leaving $d^\star = -\frac{1}{2} < -\frac{1}{4} = p^\star$.

With this geometry in place, the proof of strong duality for a single inequality constraint becomes transparent.

Proof (one inequality constraint with $g(\bar{\mathbf{x}}) < 0$, $p^\star$ finite). First, $\mathcal{A}$ is convex. Take any two points (u_1, t_1) and (u_2, t_2) in $\mathcal{A}$, contributed by $\mathbf{x}_1$ and $\mathbf{x}_2$, and any $\theta \in [0,1]$. By convexity of g, the midpoint $\mathbf{x}_\theta = \theta\mathbf{x}_1 + (1-\theta)\mathbf{x}_2$ satisfies $g(\mathbf{x}_\theta) \leq \theta g(\mathbf{x}_1) + (1-\theta)g(\mathbf{x}_2) \leq \theta u_1 + (1-\theta)u_2$, and the same inequality holds for f. Hence the convex combination of (u_1, t_1) and (u_2, t_2) belongs to $\mathcal{A}$.

Second, $(0, p^\star)$ lies on the boundary of $\mathcal{A}$: it is a limit of points in $\mathcal{A}$, whereas $(0, p^\star - \varepsilon) \notin \mathcal{A}$ for every $\varepsilon > 0$. Every boundary point of a convex set admits a supporting line: there exists a nonzero vector $(\lambda, \gamma) \neq (0,0)$ such that $\gamma t + \lambda u \geq \gamma p^\star$ for all $(u,t) \in \mathcal{A}$ (Boyd and Vandenberghe (2004), section 2.5.2; this is the same geometric fact that produces subgradients in Section 35.3). Because $\mathcal{A}$ extends indefinitely upward and to the right, we must have $\gamma \geq 0$ and $\lambda \geq 0$.

Could $\gamma = 0$? If so, the supporting line is vertical and $\lambda > 0$, forcing $\lambda u \geq 0$ (and thus

$u \geq 0$) throughout $\mathcal{A}$. That contradicts the strictly feasible point $\bar{\mathbf{x}}$, which contributes $u = g(\bar{\mathbf{x}}) < 0$. Therefore $\gamma > 0$. Dividing by γ yields $t + (\lambda/\gamma)\,u \geq p^\star$ for all $(u,t) \in \mathcal{A}$, which means $q(\lambda/\gamma) \geq p^\star$. Combined with weak duality, $q(\lambda/\gamma) = p^\star$: the dual optimum is attained and equals $p^\star$. ■

Notice how cleanly the two hypotheses separate in the proof. Convexity is used once, to make $\mathcal{A}$ convex so that a supporting line exists at $(0, p^\star)$. Slater's condition is used once, to prevent that supporting line from being vertical, which would correspond to an infinite multiplier. In panel (a) of Figure 35.26, for the clipping problem, a point of norm $\|\mathbf{x}\| = r$ has minimum objective $\tfrac{1}{2}(5-r)^2$ and constraint value $u = \tfrac{1}{2}(r^2-1)$, so the lower boundary of $\mathcal{A}$ is the curve $t = \tfrac{1}{2}\big(5 - \sqrt{1+2u}\big)^2$. Its slope at $u = 0$ is -4, matching the optimal supporting line of slope $-\lambda^\star = -4$. Panel (b) previews the nonconvex example at the end of this section: there $\mathcal{A}$ bends the wrong way, so every line below $\mathcal{A}$ passes strictly underneath $(0, p^\star)$.

35.7.2.2 Saddle Points

Strong duality can also be restated purely in terms of the Lagrangian, without even mentioning the dual function. Observe that we can rewrite the primal problem as a min–max problem: for any $\mathbf{x} \in \mathcal{D}$,

$$\sup_{\lambda \succeq 0,\, \nu} \mathcal{L}(\mathbf{x}, \lambda, \nu) = \begin{cases} f(\mathbf{x}) & \text{if } \mathbf{x} \text{ is feasible,} \\ +\infty & \text{otherwise.} \end{cases}$$

If $\mathbf{x}$ violates any constraint, we can scale the corresponding multiplier to drive $\mathcal{L}$ to $+\infty$; if $\mathbf{x}$ is feasible, the best the multipliers can do is set $\lambda_i = 0$ on every slack constraint, leaving $f(\mathbf{x})$. Consequently, $p^\star = \inf_{\mathbf{x}} \sup_{\lambda \succeq 0, \nu} \mathcal{L}$, while by definition $d^\star = \sup_{\lambda \succeq 0, \nu} \inf_{\mathbf{x}} \mathcal{L}$. Weak duality is simply the universal inequality $\sup \inf \leq \inf \sup$, which holds for any function of two variables. Strong duality says that we can swap the order of minimization and maximization without changing the value. When both optima are attained, this swapability is equivalent to the existence of a saddle point of the Lagrangian.

Proposition (strong duality is a saddle point). *Suppose strong duality holds and both optima are attained, at a primal optimum $\mathbf{x}^\star$ and a dual optimum $(\lambda^\star, \nu^\star)$. Then $(\mathbf{x}^\star, \lambda^\star, \nu^\star)$ is a* **saddle point** *of the Lagrangian: for all $\mathbf{x} \in \mathcal{D}$, all $\lambda \succeq 0$, and all ν,*

$$\mathcal{L}(\mathbf{x}^\star, \lambda, \nu) \;\leq\; \mathcal{L}(\mathbf{x}^\star, \lambda^\star, \nu^\star) \;\leq\; \mathcal{L}(\mathbf{x}, \lambda^\star, \nu^\star). \tag{35.82}$$

Conversely, any saddle point of $\mathcal{L}$ over $\mathbf{x} \in \mathcal{D}$ and $(\lambda \succeq 0, \nu)$ closes the duality gap, with saddle value $p^\star = d^\star$.

Proof. By definition of q and weak duality,

$$d^\star = q(\lambda^\star, \nu^\star) = \inf_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \lambda^\star, \nu^\star) \;\leq\; \mathcal{L}(\mathbf{x}^\star, \lambda^\star, \nu^\star) \;\leq\; f(\mathbf{x}^\star) = p^\star,$$

where the final inequality holds because $\lambda_i^\star \geq 0$ and $g_i(\mathbf{x}^\star) \leq 0$ make every penalty term nonpositive. Strong duality ($d^\star = p^\star$) forces every inequality in this chain to be an equality. Equality on the left means $\mathbf{x}^\star$ minimizes $\mathcal{L}(\cdot, \lambda^\star, \nu^\star)$, giving the right-hand inequality of Equation 35.82. Equality on the right forces $\sum_i \lambda_i^\star g_i(\mathbf{x}^\star) = 0$, which is complementary

slackness. Hence, for any $\lambda \succeq 0$ and ν, $\mathcal{L}(\mathbf{x}^\star, \lambda, \nu) = f(\mathbf{x}^\star) + \sum_i \lambda_i g_i(\mathbf{x}^\star) \le f(\mathbf{x}^\star) = \mathcal{L}(\mathbf{x}^\star, \lambda^\star, \nu^\star)$, giving the left-hand inequality of Equation 35.82.

Conversely, if $(\mathbf{x}^\star, \lambda^\star, \nu^\star)$ is a saddle point, then

$$\begin{aligned}\inf_{\mathbf{x}} \sup_{\lambda \succeq 0,\, \nu} \mathcal{L} &\le \sup_{\lambda \succeq 0,\, \nu} \mathcal{L}(\mathbf{x}^\star, \lambda, \nu) = \mathcal{L}(\mathbf{x}^\star, \lambda^\star, \nu^\star) \\ &= \inf_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \lambda^\star, \nu^\star) \le \sup_{\lambda \succeq 0,\, \nu} \inf_{\mathbf{x}} \mathcal{L},\end{aligned}$$

which shows $p^\star \le d^\star$. Weak duality supplies the reverse inequality, so $p^\star = d^\star$. ■

In our clipping example, the saddle point is $(\mathbf{x}^\star, \lambda^\star) = ((0.6, 0.8), 4)$: perturbing $\mathbf{x}$ away from $\mathbf{x}^\star$ increases $\mathcal{L}(\cdot, 4)$, while varying λ leaves $\mathcal{L}(\mathbf{x}^\star, \lambda) = 8$ flat because $g(\mathbf{x}^\star) = 0$.

Strong duality brings three practical benefits:

- *The dual can be simpler or smaller than the primal.* The support vector machine dual has one variable per data point with simple sign constraints and at most one equality; the ridge regression dual in Exercise 8 reduces to a linear system of size equal to the number of examples rather than the feature dimension.
- *Every dual point gives a stopping criterion.* For any primal-feasible $\mathbf{x}$ and any dual-feasible (λ, ν), the gap $f(\mathbf{x}) - q(\lambda, \nu)$ upper-bounds how far $f(\mathbf{x})$ can be from $p^\star$.
- *The dual solution helps recover the primal solution.* By Equation 35.82, $\mathbf{x}^\star$ is *among* the minimizers of $\mathcal{L}(\cdot, \lambda^\star, \nu^\star)$. When that minimizer is unique (for instance, when f is strictly convex), it immediately gives $\mathbf{x}^\star$, as in the clipping problem where $\mathbf{x}(4) = \mathbf{x}_0/5 = \mathbf{x}^\star$. When the minimizer is not unique, feasibility and complementary slackness pick $\mathbf{x}^\star$ out of the minimizer set. In a linear program, $\mathcal{L}(\cdot, \lambda^\star)$ is flat in $\mathbf{x}$; in the SVM with an unpenalized offset below, minimizing the Lagrangian leaves the offset undetermined, and complementary slackness pins it down.

The saddle-point proposition also settles the necessity of the KKT conditions, which Section 35.6.2 established as sufficient for convex problems. Suppose a convex problem with differentiable f, g_i, and h_j on $\mathcal{D} = \mathbb{R}^n$ satisfies Slater's condition and attains its optimum at $\mathbf{x}^\star$. Then strong duality holds with an attained dual optimum $(\lambda^\star, \nu^\star)$. Because $\mathbf{x}^\star$ minimizes $\mathcal{L}(\cdot, \lambda^\star, \nu^\star)$ over all of $\mathbb{R}^n$, its gradient $\nabla_{\mathbf{x}} \mathcal{L}$ vanishes there, and as we just saw, complementary slackness holds as well. Combined with primal and dual feasibility, these are precisely the four KKT conditions: under Slater's condition, they are both necessary and sufficient.

Finally, the min–max structure appears even in deep learning problems that have no explicit constraints. Generative adversarial networks (Section 16.1) and adversarial training are formulated directly as $\min_\theta \max_\phi$ games (Goodfellow et al. 2014; Madry et al. 2018). Without a saddle point, a gap can separate their min max and max min values. Even when a saddle point does exist, taking simultaneous gradient descent steps in the minimizing variable and ascent steps in the maximizing variable can fail to converge: for the simple bilinear objective xy, whose saddle point is the origin, simultaneous descent–ascent spirals outward for every positive step size (Exercise 5).

35.7.3 Multipliers as Prices

What is a constraint worth? In the clipping problem, if we slightly enlarge the unit disk, how much closer to $\mathbf{x}_0$ can we get? The optimal Lagrange multiplier answers that question directly. Even better, the same sensitivity argument reveals that for a convex loss, weight decay and a norm constraint on the weights trace out the exact same family of solutions.

35.7.3.1 Shadow Prices

To measure the value of each constraint, perturb its right-hand side and track how the optimal objective changes. For perturbation vectors $\mathbf{u} \in \mathbb{R}^m$ and $\mathbf{v} \in \mathbb{R}^p$, define the perturbed optimal value

$$p^\star(\mathbf{u}, \mathbf{v}) = \inf\{f(\mathbf{x}) : g_i(\mathbf{x}) \le u_i,\ h_j(\mathbf{x}) = v_j\}.$$

At $(\mathbf{0}, \mathbf{0})$ we recover the original optimum $p^\star(\mathbf{0}, \mathbf{0})$, while taking $u_i > 0$ relaxes the i-th inequality constraint. We have already seen this function geometrically: for a single inequality constraint, $p^\star(u)$ is the lowest height t that the set $\mathcal{A}$ in Figure 35.26 reaches at horizontal coordinate u. The lower boundary of $\mathcal{A}$ is the graph of $p^\star(u)$, and the supporting line of slope $-\lambda^\star$ lies everywhere below it. The next proposition extends that geometric inequality to any number of constraints.

Proposition (multipliers are shadow prices). *Suppose strong duality holds with dual optimum $(\lambda^\star, \nu^\star)$. Then for all $\mathbf{u}$ and $\mathbf{v}$,*

$$p^\star(\mathbf{u}, \mathbf{v}) \ \ge\ p^\star(\mathbf{0}, \mathbf{0}) - {\lambda^\star}^\top \mathbf{u} - {\nu^\star}^\top \mathbf{v}, \tag{35.83}$$

and if $p^\star$ is differentiable at $(\mathbf{0}, \mathbf{0})$, then $\nabla_{\mathbf{u}} p^\star = -\lambda^\star$ and $\nabla_{\mathbf{v}} p^\star = -\nu^\star$ there.

Proof. Let $\mathbf{x}$ be any point feasible for the perturbed problem, so $g_i(\mathbf{x}) \le u_i$ and $h_j(\mathbf{x}) = v_j$. Using strong duality, the definition of q as an infimum, $\lambda_i^\star \ge 0$, and the perturbed constraints, we obtain

$$\begin{aligned} p^\star(\mathbf{0}, \mathbf{0}) = q(\lambda^\star, \nu^\star) &\le f(\mathbf{x}) + \sum_i \lambda_i^\star g_i(\mathbf{x}) + \sum_j \nu_j^\star h_j(\mathbf{x}) \\ &\le f(\mathbf{x}) + {\lambda^\star}^\top \mathbf{u} + {\nu^\star}^\top \mathbf{v}. \end{aligned}$$

Taking the infimum over all perturbed-feasible $\mathbf{x}$ yields Equation 35.83. If $p^\star$ is differentiable at the origin, the nonnegative function $p^\star(\mathbf{u}, \mathbf{v}) - p^\star(\mathbf{0}, \mathbf{0}) + {\lambda^\star}^\top \mathbf{u} + {\nu^\star}^\top \mathbf{v}$ attains its global minimum of 0 at $(\mathbf{0}, \mathbf{0})$, so its gradient must vanish there. $\blacksquare$

The optimal multiplier $\lambda_i^\star$ is called the **shadow price** of constraint i: it measures the rate at which the optimal value improves as the constraint is relaxed. Notice that Equation 35.83 is global, holding for perturbations of *any* size. Relaxing constraint i by $u_i > 0$ can lower the optimal value by at most $\lambda_i^\star u_i$, whereas tightening it by the same amount raises the optimal value by at least $\lambda_i^\star u_i$; to first order, when $p^\star$ is differentiable, the change is $-\lambda_i^\star u_i$. A slack constraint has $\lambda_i^\star = 0$ by complementary slackness, matching intuition: loosening a constraint that is not binding cannot improve the optimum.

Let us check this on the clipping problem. Relaxing $g(\mathbf{x}) \le 0$ to $g(\mathbf{x}) \le u$ expands the disk to radius $r = \sqrt{1+2u}$, giving $p^\star(u) = \frac{1}{2}(5 - \sqrt{1+2u})^2$—the lower boundary of $\mathcal{A}$ we computed earlier—whose derivative at $u = 0$ is indeed $-4 = -\lambda^\star$. If we relax the constraint by $u = \frac{1}{2}$ (a disk of radius $\sqrt{2}$), the new optimum is $p^\star(\frac{1}{2}) = \frac{1}{2}(5-\sqrt{2})^2 \approx 6.43$, safely above the global lower bound $8 - 4 \cdot \frac{1}{2} = 6$ guaranteed by Equation 35.83.

One subtlety is worth keeping in mind: a multiplier depends on how the constraint is scaled. Here $\lambda^\star = 4$ prices u, which is *half* the squared radius, rather than the radius r itself. Per unit of radius, the chain rule gives $\mathrm{d}p^\star/\mathrm{d}r = -(5-r)$, which equals r times the derivative with respect to u and matches $-\lambda^\star$ only because $r = 1$. Rescaling a constraint rescales its multiplier accordingly: writing $\|\mathbf{x}\|^2 - 1 \le 0$ without the factor of $\frac{1}{2}$ would halve the multiplier to 2.

The closest-point problem of Section 35.6.1.3 illustrates the equality case. There, the minimum of $\frac{1}{2}\|\mathbf{x}\|^2$ over the hyperplane $\mathbf{a}^\top\mathbf{x} = b$ is $p^\star(b) = b^2/(2\|\mathbf{a}\|^2)$, and $\partial p^\star/\partial b = -\nu^\star$ is simply Equation 35.83 applied to the perturbation $h(\mathbf{x}) = \mathbf{a}^\top\mathbf{x} - b = v$, which shifts the hyperplane to $b + v$. That explains the increase of $0.02 = 0.2 \cdot 0.1$ that we observed numerically in that section.

35.7.3.2 Weight Decay and Norm Constraints

In deep learning, we regularly control model complexity either by adding weight decay (Section 2.7) to a training loss $\ell(\mathbf{w})$ or by imposing a hard norm constraint on the weights. Using the book's convention, these two problems are

$$\min_{\mathbf{w}} \ \ell(\mathbf{w}) + \frac{\lambda}{2}\|\mathbf{w}\|^2 \qquad \text{and} \qquad \min_{\mathbf{w}} \ \ell(\mathbf{w}) \quad \text{subject to} \quad \tfrac{1}{2}\|\mathbf{w}\|^2 \le \tfrac{1}{2}r^2.$$

At a stationary point of the penalized problem, $\nabla\ell(\mathbf{w}) + \lambda\mathbf{w} = \mathbf{0}$. For the constrained problem, introducing a multiplier $\lambda \ge 0$ for $g(\mathbf{w}) = \frac{1}{2}\|\mathbf{w}\|^2 - \frac{1}{2}r^2$ produces the KKT stationarity condition $\nabla\ell(\mathbf{w}) + \lambda\mathbf{w} = \mathbf{0}$: the very same equation, with the weight-decay coefficient playing the role of the Lagrange multiplier.

When the loss ℓ is *convex*, this equivalence works in both directions. Going from penalty to constraint, let $\mathbf{w}_\lambda$ minimize the penalized objective and choose the norm budget $r = \|\mathbf{w}_\lambda\|$. Then $\mathbf{w}_\lambda$ is feasible, the constraint is active (satisfying complementary slackness), and KKT stationarity holds automatically. Because the KKT conditions are sufficient for convex problems (Section 35.6.2), $\mathbf{w}_\lambda$ solves the constrained problem.

Going from constraint to penalty, take any budget $r > 0$. The origin $\mathbf{w} = \mathbf{0}$ is strictly feasible, so Slater's condition holds. If the constrained minimum is attained at $\mathbf{w}^\star$, KKT necessity yields a multiplier $\lambda^\star(r) \ge 0$, and $\mathbf{w}^\star$ minimizes the convex penalized objective $\ell(\mathbf{w}) + \frac{\lambda^\star}{2}\|\mathbf{w}\|^2$. Whenever $\lambda^\star > 0$, the quadratic term makes this objective strictly convex, so $\mathbf{w}^\star$ is its unique minimizer; when the budget is slack, $\lambda^\star = 0$ and the penalized problem reduces to unregularized minimization. By the shadow-price proposition, $\lambda^\star(r) = -\partial p^\star/\partial(\frac{1}{2}r^2)$: the weight-decay coefficient is the marginal reduction in training loss per unit of half squared norm.

In fact, neither direction depends on the ℓ_2 norm specifically. Replace $\frac{1}{2}\|\mathbf{w}\|^2$ by any convex penalty $\Omega(\mathbf{w})$, such as the ℓ_1 norm $\|\mathbf{w}\|_1$ used in the lasso (Section 35.6.4.1). Any minimizer $\mathbf{w}_\lambda$ of $\ell + \lambda\Omega$ solves the constrained problem with budget $\Omega(\mathbf{w}) \le \Omega(\mathbf{w}_\lambda)$, since every $\mathbf{w}$

within that budget satisfies $\ell(\mathbf{w}) \geq \ell(\mathbf{w}_\lambda) + \lambda\,(\Omega(\mathbf{w}_\lambda) - \Omega(\mathbf{w})) \geq \ell(\mathbf{w}_\lambda)$. Conversely, for convex ℓ and any budget strictly above $\inf \Omega$, Slater's condition holds, and the saddle-point proposition ensures that any constrained minimizer also minimizes $\ell + \lambda^\star \Omega$.

Convexity of ℓ, however, is essential for the full equivalence. For a nonconvex loss, a penalized stationary point is still a KKT point of the matching constrained problem, but the global solution sets can differ, and certain norm budgets may be unattainable by *any* penalty weight λ (Exercise 6).

It is worth noting a third interpretation of the same quadratic term. For a squared loss with noise variance σ^2 and an isotropic Gaussian prior $\mathcal{N}(\mathbf{0}, \tau^2\mathbf{I})$ on the weights, the penalty $\frac{\lambda}{2}\|\mathbf{w}\|^2$ with $\lambda = \sigma^2/\tau^2$ is the negative log-prior up to a constant and overall scale (Section 36.6). Thus a single number λ can be read three ways: as a weight-decay coefficient, as a Lagrange multiplier for a norm budget, and as a ratio of noise to prior variance.

Let us verify this equivalence numerically. For the convex least-squares loss $\ell(\mathbf{w}) = \frac{1}{2}\|\mathbf{A}\mathbf{w} - \mathbf{b}\|^2$ on random data, we solve the penalized problem in closed form via $(\mathbf{A}^\top\mathbf{A} + \lambda\mathbf{I})\mathbf{w} = \mathbf{A}^\top\mathbf{b}$, set $r = \|\mathbf{w}_\lambda\|$, and then solve the constrained problem using projected gradient descent with step size $1/\lambda_{\max}(\mathbf{A}^\top\mathbf{A})$, projecting onto the r-ball by clipping. Does the constrained solver recover the same weights, and does the multiplier computed from stationarity—taking the inner product of $\nabla\ell(\mathbf{w}) + \lambda\mathbf{w} = \mathbf{0}$ with $\mathbf{w}$ to get $\lambda = -\mathbf{w}^\top\nabla\ell(\mathbf{w})/r^2$—match λ?

```
rng = np.random.default_rng()
A, b = rng.normal(size=(30, 5)), rng.normal(size=30)
grad = lambda w: A.T @ (A @ w - b)                    # gradient of 1/2 |Aw - b|^2
step = 1.0 / np.linalg.eigvalsh(A.T @ A).max()        # 1 / lambda_max(A^T A)
print(' lambda   r = |w_pen|   |w_con - w_pen|   recovered multiplier')
for lam in [0.1, 1.0, 10.0]:
    w_pen = np.linalg.solve(A.T @ A + lam * np.eye(5), A.T @ b)  # penalty form
    r = np.linalg.norm(w_pen)                         # its norm becomes the budget
    w = np.zeros(5)                                   # constraint form: PGD
    for _ in range(5000):
        w -= step * grad(w)
        w *= min(1.0, r / np.linalg.norm(w))          # project onto the r-ball
    lam_rec = -(w @ grad(w)) / r ** 2                 # KKT: grad l + lam w = 0
    print(f'{lam:7.2f}  {r:10.4f}  {np.linalg.norm(w - w_pen):15.2e}  {lam_rec:20.6f}')
```

```
 lambda   r = |w_pen|   |w_con - w_pen|   recovered multiplier
   0.10       0.4138         1.28e-16              0.100000
   1.00       0.3860         1.58e-16              1.000000
  10.00       0.2501         5.72e-17             10.000000
```

In every row, the constrained solution agrees with the penalized one to machine precision, and the multiplier recovered from stationarity matches λ to all six printed decimals (the exact budgets r vary with the random draw). Penalty and constraint are simply two parameterizations of the same solution path, with the Lagrange multiplier translating between them.

35.7.4 The Convex Conjugate

Evaluating the dual function q normally requires solving a fresh unconstrained minimization for each choice of multipliers. When the constraints are linear, however, we can fac-

tor out the shared work and perform that minimization once and for all. For constraints $\mathbf{Ax} \preceq \mathbf{b}$, the Lagrangian is $f(\mathbf{x}) + \lambda^\top(\mathbf{Ax} - \mathbf{b})$, so

$$q(\lambda) = \inf_{\mathbf{x}} [f(\mathbf{x}) + (\mathbf{A}^\top \lambda)^\top \mathbf{x}] - \mathbf{b}^\top \lambda.$$

Notice that the infimum depends on f and on the single vector $\mathbf{A}^\top\lambda$, not on $\mathbf{A}$ or $\mathbf{b}$ separately. If we compute, once for each common objective f, the value of $\inf_{\mathbf{x}}[f(\mathbf{x}) - \mathbf{y}^\top\mathbf{x}]$ as a function of $\mathbf{y}$, then the dual of *any* linearly constrained problem with objective f can be read straight off a lookup table. That lookup table is the convex conjugate.

35.7.4.1 Supporting Lines

Start in one dimension with $f(x) = \frac{1}{2}x^2$ and pick a slope $y = 2$. Which lines $t = 2x + c$ of slope 2 stay everywhere on or below the parabola? We need $2x + c \le \frac{1}{2}x^2$ for all x, meaning $c \le \min_x(\frac{1}{2}x^2 - 2x) = -2$, attained at $x = 2$. The highest such line, $t = 2x - 2$, touches the parabola tangentially at $(2, 2)$, where $f'(2) = 2$. Lowering a line of slope 2 until it touches the graph lands its vertical intercept at -2, and the number $\max_x(2x - \frac{1}{2}x^2) = 2$ is minus that intercept, as shown in Figure 35.27(a).

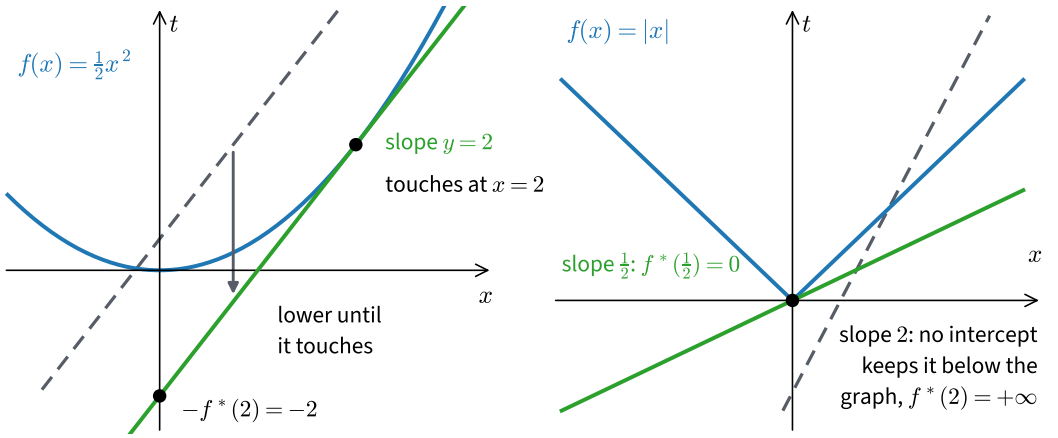


Figure 35.27: The convex conjugate as a supporting line. (a) For $f(x) = \frac{1}{2}x^2$ and slope 2, lowering the dashed line of slope 2 until it touches the graph lands at $x = 2$ with intercept -2, giving $f^*(2) = 2$. (b) For $f(x) = |x|$, a line of slope $\frac{1}{2}$ supports the graph at the origin with intercept 0, whereas a line of slope 2 climbs faster than $|x|$ for every intercept, giving $f^*(\frac{1}{2}) = 0$ and $f^*(2) = +\infty$. Thus the conjugate is finite precisely for the slopes that admit a supporting line below the graph.

The **convex conjugate** (also known as the Fenchel–Legendre transform) of a function $f : \mathbb{R}^n \to \mathbb{R} \cup \{+\infty\}$ records this negated intercept for every slope vector $\mathbf{y}$ (we use an asterisk $*$ for conjugates and a star $\star$ for optima):

$$f^*(\mathbf{y}) = \sup_{\mathbf{x}} \left(\mathbf{y}^\top\mathbf{x} - f(\mathbf{x})\right). \tag{35.84}$$

In geometric terms, $f^*(\mathbf{y})$ is minus the vertical intercept of the highest affine function of slope $\mathbf{y}$ that lies everywhere below the graph of f. Rather than describing f by its point values, the conjugate describes f by its family of supporting lines. For $f(x) = \frac{1}{2}x^2$ and any slope y, the supremum is attained at $x = y$, giving $f^*(y) = \frac{1}{2}y^2$: the quadratic is its own conjugate.

Allowing f to take the value $+\infty$ is a convenient way to encode a restricted domain: points where $f(\mathbf{x}) = +\infty$ never contribute to the supremum, and convexity of such an extended-valued function simply means convexity of its epigraph.

The supremum itself can also be $+\infty$. Take $f(x) = |x|$ and slope $y = 2$. For $x > 0$, $2x - |x| = x$ grows without bound: no line of slope 2 can stay below the V-shaped graph, no matter how low its intercept. Whenever $|y| \le 1$, on the other hand, $yx - |x| \le 0$ with equality at $x = 0$, so the line of slope y through the origin supports the graph. Therefore $f^*(y) = 0$ for $|y| \le 1$ and $f^*(y) = +\infty$ otherwise (Figure 35.27(b)). In other words, the conjugate of the absolute value is the *indicator function* of the interval $[-1, 1]$—the function equal to 0 inside the set and $+\infty$ outside it.

Two immediate properties follow straight from Equation 35.84. First, f^* is *always* convex, regardless of whether f is convex. For each fixed $\mathbf{x}$, the expression $\mathbf{y}^\top\mathbf{x} - f(\mathbf{x})$ is affine in $\mathbf{y}$, and the pointwise supremum of affine functions is convex because its epigraph is the intersection of the half-spaces above them (the exact mirror image of the argument that made q concave).

Second, because $f^*(\mathbf{y})$ is the supremum over all $\mathbf{x}$, we have $f^*(\mathbf{y}) \ge \mathbf{y}^\top\mathbf{x} - f(\mathbf{x})$ for every $\mathbf{x}$ and $\mathbf{y}$. Rearranging gives the **Fenchel–Young inequality**,

$$\mathbf{y}^\top\mathbf{x} \;\le\; f(\mathbf{x}) + f^*(\mathbf{y}) \qquad \text{for all } \mathbf{x}, \mathbf{y}. \tag{35.85}$$

Equality holds if and only if $\mathbf{x}$ attains the supremum in Equation 35.84. When f is convex and differentiable, this happens precisely when $\mathbf{y} = \nabla f(\mathbf{x})$, since $\mathbf{x} \mapsto \mathbf{y}^\top\mathbf{x} - f(\mathbf{x})$ is then concave and maximized where its gradient vanishes. Convexity matters here: the nonconvex double well $\phi_0(x) = (x^2 - 1)^2$ has $\phi_0'(0) = 0$, yet $\phi_0^*(0) = -\min \phi_0 = 0$ and $\phi_0(0) + \phi_0^*(0) = 1 > 0$, because $x = 0$ is a local maximum rather than a point where a horizontal line supports the graph from below. For the self-conjugate pair $\frac{1}{2}\|\mathbf{x}\|^2$, Fenchel–Young reduces to the familiar inequality $\mathbf{y}^\top\mathbf{x} \le \frac{1}{2}\|\mathbf{x}\|^2 + \frac{1}{2}\|\mathbf{y}\|^2$.

Does passing to the conjugate lose any information? For any convex function f that is *proper* (never $-\infty$ and not identically $+\infty$) and *closed* (its epigraph is a closed set), its supporting lines determine the graph completely, and taking the conjugate a second time recovers f: $f^{**} = f$ (Rockafellar 1970). When f is nonconvex (and bounded below by at least one affine function), f^{**} is the tightest closed convex function lying below f—its convex envelope (Exercise 1d).

Here are the four conjugate pairs we use in this section and its exercises:

- $\frac{1}{2}\|\mathbf{x}\|^2$ and $\frac{1}{2}\|\mathbf{y}\|^2$, the self-conjugate pair;
- $\frac{1}{2}\mathbf{x}^\top\mathbf{A}\mathbf{x}$ and $\frac{1}{2}\mathbf{y}^\top\mathbf{A}^{-1}\mathbf{y}$ for $\mathbf{A} \succ 0$ (attained at $\mathbf{x} = \mathbf{A}^{-1}\mathbf{y}$);
- e^x and $y \log y - y$ for $y > 0$, taking the value 0 at $y = 0$ and $+\infty$ for $y < 0$ (attained at $x = \log y$, so for instance $f^*(1) = -1$ and $f^*(e) = 0$);
- $\lambda\|\mathbf{x}\|_1$ and the indicator of the box $\{\mathbf{y} : \|\mathbf{y}\|_\infty \le \lambda\}$, the vector version of the absolute value above.

35.7.4.2 Duals of Linearly Constrained Problems

Now return to the linearly constrained problem that motivated the conjugate: minimize $f(\mathbf{x})$ subject to $\mathbf{A}\mathbf{x} \preceq \mathbf{b}$. Pulling a minus sign out of the infimum turns it directly into the supremum defining f^*:

$$\begin{aligned} q(\lambda) &= \inf_{\mathbf{x}} \left[f(\mathbf{x}) + (\mathbf{A}^\top \lambda)^\top \mathbf{x} \right] - \mathbf{b}^\top \lambda \\ &= -\sup_{\mathbf{x}} \left[(-\mathbf{A}^\top \lambda)^\top \mathbf{x} - f(\mathbf{x}) \right] - \mathbf{b}^\top \lambda \\ &= -f^*(-\mathbf{A}^\top \lambda) - \mathbf{b}^\top \lambda. \end{aligned} \tag{35.86}$$

For equality constraints $\mathbf{A}\mathbf{x} = \mathbf{b}$, the exact same three lines yield $q(\nu) = -f^*(-\mathbf{A}^\top \nu) - \mathbf{b}^\top \nu$, with ν unconstrained in sign. Notice that this gives a second explanation for why q is concave: f^* is convex, its argument is linear in λ, and the minus sign flips convexity to concavity.

Let us try Equation 35.86 on the smallest possible example: finding the point closest to the origin in a half-plane, $\min_{\mathbf{x}} \frac{1}{2}\|\mathbf{x}\|^2$ subject to $\mathbf{a}^\top \mathbf{x} \le b$, with $\mathbf{a} = (1, 2)$ and $b = -1$. Since the origin violates the constraint ($\mathbf{a}^\top \mathbf{0} = 0 > -1$), the constraint must bind at the optimum. Plugging $f^*(\mathbf{y}) = \frac{1}{2}\|\mathbf{y}\|^2$ and $\|\mathbf{a}\|^2 = 5$ into Equation 35.86 gives the dual function in one step:

$$q(\lambda) = -\tfrac{1}{2}\lambda^2 \|\mathbf{a}\|^2 - b\lambda = \lambda - \tfrac{5}{2}\lambda^2.$$

Setting $q'(\lambda) = 1 - 5\lambda = 0$ yields $\lambda^\star = \frac{1}{5}$ and $d^\star = \frac{1}{5} - \frac{1}{10} = \frac{1}{10}$. The minimizer of the Lagrangian at $\lambda^\star$ is the point attaining the conjugate supremum, $\mathbf{x} = -\lambda^\star \mathbf{a} = (-0.2, -0.4)$. Checking directly, $\mathbf{a}^\top \mathbf{x} = -1 = b$ and $f(\mathbf{x}) = \frac{1}{2}(0.04 + 0.16) = \frac{1}{10} = d^\star$, so weak duality certifies that $\mathbf{x}$ is optimal. This matches the projection formula $(b/\|\mathbf{a}\|^2)\,\mathbf{a}$ from Section 35.6.1.3, obtained here for a different hyperplane—and where the primal had two variables, the dual had one and fell in a single line.

35.7.4.3 Log-Sum-Exp and Entropy

Throughout deep learning, softmax converts raw scores into probabilities by exponentiating and normalizing. Why exponentials, rather than some other positive function? The conjugate of the log-sum-exp function, $\mathrm{lse}(\mathbf{x}) = \log \sum_i e^{x_i}$, gives a clean answer. Let $\Delta = \{\mathbf{p} : \mathbf{p} \succeq 0,\ \mathbf{1}^\top \mathbf{p} = 1\}$ denote the probability simplex and $H(\mathbf{p}) = -\sum_i p_i \log p_i$ the Shannon entropy.

To compute $\mathrm{lse}^*(\mathbf{p}) = \sup_{\mathbf{x}} (\mathbf{p}^\top \mathbf{x} - \mathrm{lse}(\mathbf{x}))$, first check when the supremum can be finite. If any coordinate p_i is negative, sending $x_i \to -\infty$ drives $\mathbf{p}^\top \mathbf{x}$ to $+\infty$ while $\mathrm{lse}(\mathbf{x})$ stays bounded, so the supremum is $+\infty$. If $\mathbf{1}^\top \mathbf{p} \neq 1$, shifting all coordinates by a constant $\mathbf{x} = c\mathbf{1}$ gives $c(\mathbf{1}^\top \mathbf{p} - 1) - \log n$ (since $\mathrm{lse}(c\mathbf{1}) = c + \log n$), which again blows up as $c \to \pm\infty$. Hence $\mathrm{lse}^*(\mathbf{p})$ is finite only on the probability simplex Δ.

For $\mathbf{p}$ in the interior of Δ (all $p_i > 0$), setting the gradient with respect to $\mathbf{x}$ to zero gives $\mathrm{softmax}(\mathbf{x}) = \mathbf{p}$, which is solved by $\mathbf{x} = \log \mathbf{p}$ (up to an additive constant), yielding the value $\sum_i p_i \log p_i$. If $\mathbf{p}$ lies on the boundary of Δ with some $p_i = 0$, no finite maximizer exists because softmax outputs are always strictly positive; nevertheless, sending those coordinates $x_i \to -\infty$ approaches the same value, so the formula extends continuously to all of

Δ under the standard convention $0 \log 0 = 0$. Because lse is convex and finite everywhere, $\text{lse}^{**} = \text{lse}$, giving the conjugate pair

$$\begin{aligned} \text{lse}^*(\mathbf{p}) &= \begin{cases} \sum_i p_i \log p_i = -H(\mathbf{p}), & \mathbf{p} \in \Delta, \\ +\infty, & \text{otherwise}, \end{cases} \\ \text{lse}(\mathbf{x}) &= \max_{\mathbf{p} \in \Delta} \left(\mathbf{p}^\top \mathbf{x} + H(\mathbf{p}) \right) \quad \text{(dually)}. \end{aligned} \tag{35.87}$$

Read the second line of Equation 35.87 in words: log-sum-exp is the maximum of an expected score $\mathbf{p}^\top \mathbf{x}$ plus an entropy bonus $H(\mathbf{p})$ over all distributions on n outcomes. Which distribution achieves this maximum? By the equality condition of Fenchel–Young, it is $\mathbf{p} = \nabla \text{lse}(\mathbf{x})$, whose i-th entry is $\partial \, \text{lse} / \partial x_i = e^{x_i} / \sum_j e^{x_j}$—precisely the softmax (whose Hessian we computed in Section 35.3). Put another way, among all distributions achieving the same expected score $\mathbf{p}^\top \mathbf{x}$, softmax is the one with the largest entropy, matching the maximum-entropy derivation in Section 36.4.2 (and Exercise 2 derives softmax from that constraint side). Without the entropy term, maximizing $\mathbf{p}^\top \mathbf{x}$ over Δ puts all mass on $\arg\max_i x_i$; adding entropy softens that hard selection into a smooth probability distribution. The very same conjugacy construction, applied to the convex generator of an f-divergence, produces the variational bounds used to train f-GANs (Section 37.3.4).

35.7.5 Worked Examples

We now work through three examples that highlight what duality reveals in practice. In the support vector machine, complementary slackness concentrates the solution on a handful of data points, while the dual exposes the data only through inner products. In water-filling, the Lagrange multiplier carries tangible engineering units—communication rate per unit of power. Finally, a one-dimensional nonconvex problem demonstrates the strict duality gap that convexity prevents.

35.7.5.1 The Support Vector Machine Dual

Given linearly separable training data $(\mathbf{x}_i, y_i)$ with labels $y_i \in \{\pm 1\}$, the support vector machine (Boser et al. 1992) seeks the separating hyperplane $\mathbf{w}^\top \mathbf{x} + b = 0$ with the widest possible margin. Its primal problem is

$$\min_{\mathbf{w}, b} \ \tfrac{1}{2} \|\mathbf{w}\|^2 \quad \text{subject to} \quad y_i \left(\mathbf{w}^\top \mathbf{x}_i + b \right) \geq 1, \quad i = 1, \ldots, n. \tag{35.88}$$

Because the signed distance from a point $\mathbf{x}$ to the hyperplane is $|\mathbf{w}^\top \mathbf{x} + b| / \|\mathbf{w}\|$ (Section 33.1), the two margin hyperplanes $\mathbf{w}^\top \mathbf{x} + b = \pm 1$ sit at distance $1/\|\mathbf{w}\|$ on either side of the decision boundary, or $2/\|\mathbf{w}\|$ apart. Minimizing $\frac{1}{2}\|\mathbf{w}\|^2$ therefore maximizes the margin. This is a convex quadratic program with one affine inequality per training point. Since the constraints are affine, Slater's condition requires only a feasible point—which we obtain by rescaling any separating $(\mathbf{w}, b)$ until every point has margin at least 1. Strong duality is thus guaranteed.

Introduce one nonnegative multiplier $\alpha_i \geq 0$ for each constraint and form the Lagrangian,

$$\mathcal{L}(\mathbf{w}, b, \alpha) = \tfrac{1}{2} \|\mathbf{w}\|^2 + \sum_i \alpha_i \left(1 - y_i (\mathbf{w}^\top \mathbf{x}_i + b) \right).$$

To compute the dual function $q(\alpha) = \inf_{\mathbf{w},b} \mathcal{L}$, notice first that $\mathcal{L}$ is affine in the unpenalized offset b, with coefficient $-\sum_i \alpha_i y_i$. Unless that coefficient vanishes, sending $b \to \pm\infty$ drives $\mathcal{L}$ to $-\infty$, so $q(\alpha) = -\infty$ whenever $\sum_i \alpha_i y_i \neq 0$. Here we see the effective domain of Equation 35.80 in action: the finiteness condition $\sum_i \alpha_i y_i = 0$ becomes an explicit equality constraint in the dual. On this domain the b-term disappears, leaving a strictly convex quadratic in $\mathbf{w}$ whose unique minimum occurs where $\nabla_{\mathbf{w}} \mathcal{L} = \mathbf{0}$:

$$\mathbf{w} = \sum_i \alpha_i y_i \, \mathbf{x}_i, \qquad \text{on the domain} \qquad \sum_i \alpha_i y_i = 0.$$

Thus the optimal weight vector is a linear combination of the training inputs, weighted by their multipliers. Substituting this expression back into $\mathcal{L}$ yields the **SVM dual**:

$$\max_{\alpha \succeq 0,\ \sum_i \alpha_i y_i = 0} \ \sum_i \alpha_i \ - \ \tfrac{1}{2} \sum_{i,j} \alpha_i \alpha_j \, y_i y_j \, \mathbf{x}_i^\top \mathbf{x}_j. \tag{35.89}$$

Two striking consequences follow before we even choose a solver. First, complementary slackness, $\alpha_i (1 - y_i(\mathbf{w}^\top \mathbf{x}_i + b)) = 0$, tells us that any point with $\alpha_i > 0$ must satisfy $y_i(\mathbf{w}^\top \mathbf{x}_i + b) = 1$, lying right on one of the margin hyperplanes. These points are the **support vectors**. In Figure 35.28 they can only be among the four ringed points. Their constraints are active and their normals balance ∇f, as in Figure 35.21, whereas every point with $\alpha_i = 0$ drops out of $\mathbf{w}$ completely. Second, the training inputs enter Equation 35.89 exclusively through inner products $\mathbf{x}_i^\top \mathbf{x}_j$. Replacing those inner products with a kernel function $k(\mathbf{x}_i, \mathbf{x}_j)$ immediately yields a nonlinear classifier—a structure the primal formulation hides.

Notice that the equality constraint $\sum_i \alpha_i y_i = 0$ arose solely because the unpenalized offset b left the Lagrangian affine in one direction. To keep the dual feasible set as simple as the nonnegative orthant (where projection just clips negative entries to zero), we solve a closely related variant in code by folding the offset into the weights: augment each input with a constant feature $\tilde{\mathbf{x}}_i = (\mathbf{x}_i, 1)$, set $\tilde{\mathbf{w}} = (\mathbf{w}, b)$, and minimize $\frac{1}{2}\|\tilde{\mathbf{w}}\|^2$ subject to $y_i \tilde{\mathbf{w}}^\top \tilde{\mathbf{x}}_i \geq 1$. Because this variant penalizes b as well, its decision boundary can differ slightly from Equation 35.88; in return, its Lagrangian is strictly convex in every primal variable, eliminating the equality constraint. Defining $\mathbf{Q}_{ij} = y_i y_j \tilde{\mathbf{x}}_i^\top \tilde{\mathbf{x}}_j$, the dual becomes

$$\max_{\alpha \succeq 0} \ \mathbf{1}^\top \alpha - \tfrac{1}{2} \alpha^\top \mathbf{Q} \, \alpha, \tag{35.90}$$

a concave quadratic over the nonnegative orthant. Applying projected gradient ascent—the update of Section 35.6.3 with the step sign flipped—iterates $\alpha \leftarrow \max(\mathbf{0}, \, \alpha + \eta \, (\mathbf{1} - \mathbf{Q}\alpha))$ using the safe step size $\eta = 1/\lambda_{\max}(\mathbf{Q})$. Let us run this ascent on eight points in the plane and check three predictions: does the recovered hyperplane separate the data, do all positive multipliers sit on points of margin 1, and does the dual value match the primal value?

```
X = np.array([[1.0, 1.5], [2.0, 0.5], [2.5, 2.0], [1.5, 3.0],
              [-1.0, -0.5], [-2.0, 1.0], [-1.5, -2.0], [0.5, -1.5]])
y = np.array([1.0, 1.0, 1.0, 1.0, -1.0, -1.0, -1.0, -1.0])
Xa = np.hstack([X, np.ones((len(X), 1))])     # fold the offset b into w
```

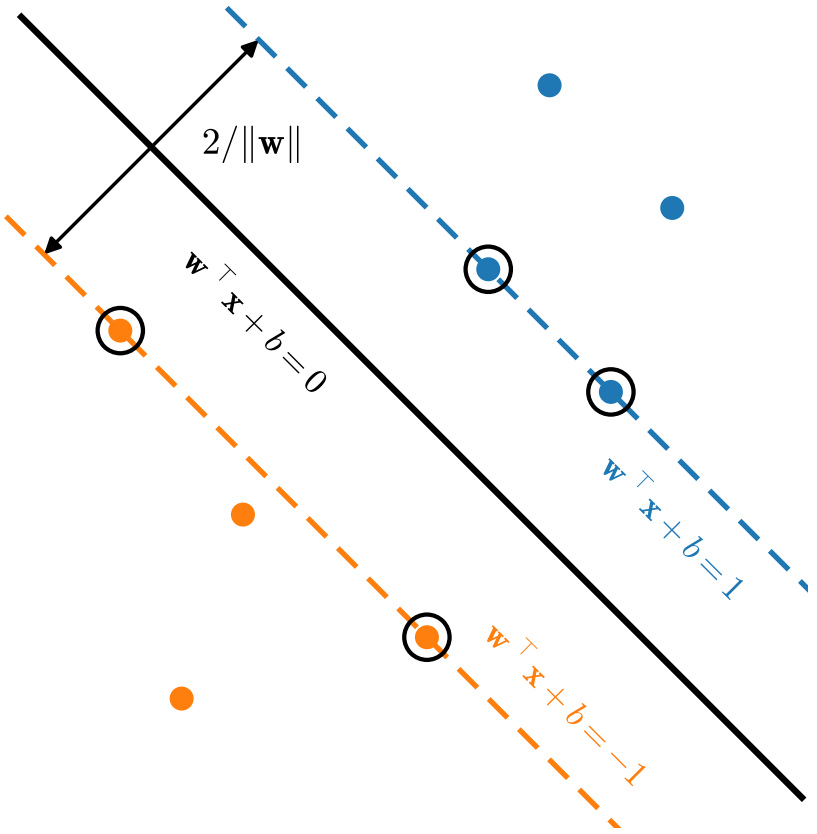


Figure 35.28: Maximum-margin separation of the eight points that the code cell below uses. The solid line is the decision boundary $\mathbf{w}^{\star\top}\mathbf{x} + b^\star = 0$ with $\mathbf{w}^\star = (\frac{4}{7}, \frac{4}{7})$ and $b^\star = -\frac{3}{7}$. The dashed lines are the margin hyperplanes $\mathbf{w}^{\star\top}\mathbf{x} + b^\star = \pm 1$, a distance $2/\|\mathbf{w}^\star\|$ apart. Only the four ringed points, which lie on a margin hyperplane, can carry $\alpha_i > 0$.

```
G = y[:, None] * Xa                              # rows y_i x~_i
Q = G @ G.T                                      # Q_ij = y_i y_j x~_i^T x~_j
alpha = np.zeros(len(X))
eta = 1.0 / np.linalg.eigvalsh(Q).max()          # 1 / lambda_max(Q)
for _ in range(5000):                            # projected gradient ascent
    alpha = np.maximum(0.0, alpha + eta * (1.0 - Q @ alpha))
w = G.T @ alpha                                  # w~* = sum_i alpha_i y_i x~_i
margins = y * (Xa @ w)
dual, primal = alpha.sum() - 0.5 * alpha @ Q @ alpha, 0.5 * w @ w
print('alpha* =', alpha.round(4))
print('rank of Q =', np.linalg.matrix_rank(Q), '  multipliers > 0:', int((alpha >
↪  1e-8).sum()))
print('w* =', w[:2].round(4), ' b* =', w[2].round(4),
      ' separates data:', bool((np.sign(Xa @ w) == y).all()))
print('margins y_i (w^T x_i + b) =', margins.round(4))
print(f'primal = {primal:.6f}, dual = {dual:.6f}, gap = {abs(primal - dual):.1e}, '
      f'comp. slack = {np.abs(alpha * (margins - 1.0)).max():.1e}')
```

```
alpha* = [0.1309 0.0732 0.     0.     0.     0.2442 0.     0.3885]
rank of Q = 3   multipliers > 0: 4
w* = [0.5714 0.5714]  b* = -0.4286  separates data: True
margins y_i (w^T x_i + b) = [1.     1.     2.1429 2.1429 1.2857 1.     2.4286 1.    ]
primal = 0.418367, dual = 0.418367, gap = 4.4e-16, comp. slack = 3.0e-16
```

All three predictions hold. Four multipliers are positive, and precisely those four points sit at margin 1; the other four have margins between 1.29 and 2.43 with $\alpha_i = 0$. The printed digits match the exact solution $\tilde{\mathbf{w}}^\star = (\frac{4}{7}, \frac{4}{7}, -\frac{3}{7})$—that is, $\mathbf{w}^\star = (\frac{4}{7}, \frac{4}{7})$ and $b^\star = -\frac{3}{7}$—with $p^\star = d^\star = \frac{41}{98} \approx 0.418367$ and a duality gap near 10^{-16}. (The equality of the primal and dual values also follows directly from the identity $\sum_i \alpha_i^\star = \|\tilde{\mathbf{w}}^\star\|^2$ at the optimum; see Exercise 4a.)

The printout also highlights a subtle point about interpreting dual solutions. Because the primal objective $\frac{1}{2}\|\tilde{\mathbf{w}}\|^2$ is strictly convex, the primal optimum $\tilde{\mathbf{w}}^\star$ is unique. The dual optimum $\alpha^\star$ is *not*. Here $\mathbf{Q} = \mathbf{G}\mathbf{G}^\top$, where $\mathbf{G}$ is the 8×3 matrix with rows $y_i \tilde{\mathbf{x}}_i^\top$, so $\mathbf{Q}$ has rank at most 3. Because the four support vectors correspond to four vectors in $\mathbb{R}^3$, they are linearly dependent: there exists a nonzero vector $\mathbf{d}$ supported on those four indices with $\mathbf{G}^\top \mathbf{d} = \mathbf{0}$. Moving $\alpha^\star$ along $\mathbf{d}$ leaves the quadratic term unchanged (since $\mathbf{Q}\mathbf{d} = \mathbf{0}$) and leaves the linear term unchanged as well, because every support vector has margin 1: $\mathbf{1}^\top \mathbf{d} = \sum_i d_i \, y_i \tilde{\mathbf{w}}^{\star\top} \tilde{\mathbf{x}}_i = \tilde{\mathbf{w}}^{\star\top} \mathbf{G}^\top \mathbf{d} = 0$. Thus an entire line segment of multiplier vectors is optimal, all yielding the same $\tilde{\mathbf{w}}^\star$ and the same dual value; projected ascent from $\mathbf{0}$ simply returns one point on that segment. At each endpoint of the segment, one of the four margin-1 points actually has $\alpha_i = 0$ (Exercise 4c). Remember that complementary slackness runs in one direction: $\alpha_i > 0$ guarantees margin 1, but margin 1 does not force $\alpha_i > 0$.

35.7.5.2 Water-Filling

Our second worked example has both a closed-form solution and an unforgettable physical picture. Suppose a transmitter must divide a total power budget P across n independent communication channels. Channel i has background noise power $N_i > 0$, and allocating power $P_i \geq 0$ to it yields a transmission rate of $\log(1 + P_i/N_i)$ nats per use (Cover and Thomas 1999). (Cover and Thomas include a factor of $\frac{1}{2}$ for a real Gaussian channel, which

rescales the Lagrange multiplier below without changing the optimal allocation.) To maximize the total rate, we solve

$$\max_{\mathbf{P}} \; U(\mathbf{P}) = \sum_{i=1}^{n} \log\left(1 + \frac{P_i}{N_i}\right) \quad \text{subject to} \quad P_i \geq 0, \;\; \sum_i P_i = P. \tag{35.91}$$

Maximizing the concave function U is equivalent to minimizing the convex function $-U$ under affine constraints. Because a strictly positive feasible allocation exists, Slater's condition holds, and the KKT conditions are both necessary and sufficient. Introducing a multiplier ν for the budget equality $\sum_i P_i - P = 0$ and multipliers $\lambda_i \geq 0$ for the nonnegativity constraints $-P_i \leq 0$, stationarity of the Lagrangian of $-U$ reads

$$\frac{1}{N_i + P_i} = \nu - \lambda_i.$$

Now read off the two cases. Whenever channel i receives positive power ($P_i > 0$), complementary slackness forces $\lambda_i = 0$, so $N_i + P_i = 1/\nu$: every active channel ends up at the exact same total level $1/\nu$. Whenever channel i receives no power ($P_i = 0$), dual feasibility $\lambda_i = \nu - 1/N_i \geq 0$ requires $N_i \geq 1/\nu$. Writing $\theta = 1/\nu$ for this common threshold, the optimal allocation takes the compact form

$$P_i^\star = \max(\theta - N_i,\, 0), \qquad \text{with } \theta \text{ set by } \sum_i \max(\theta - N_i,\, 0) = P.$$

This is the classic **water-filling** rule. Imagine each channel as a basin whose floor sits at height N_i, and pour a volume P of water into the collection of basins. As Figure 35.29 illustrates, the water settles to a uniform surface level θ, pouring the most power into the quietest channels and leaving any basin whose floor exceeds θ completely dry. Because $\sum_i \max(\theta - N_i, 0)$ is continuous, piecewise linear, zero for $\theta \leq \min_i N_i$, and strictly increasing for $\theta > \min_i N_i$, any budget $P > 0$ determines a unique water level θ, which bisection finds effortlessly. (The simplex projection in Section 35.6.3 has the same threshold structure inverted: subtract a common level from each coordinate and clip at zero.)

The shadow-price proposition gives $\nu^\star$ an immediate engineering meaning. Since maximizing U is minimizing $-U$, relaxing the budget from P to $P + v$ changes $-U^\star$ at rate $-\nu^\star$, so $\mathrm{d}U^\star/\mathrm{d}P = \nu^\star = 1/\theta$: the optimal multiplier is the extra communication rate gained per unit of additional power. Let us compute the water-filling allocation for five channels and check this shadow-price prediction against finite differences at $P \pm 10^{-4}$.

```
N = np.array([0.1, 0.4, 0.8, 1.6, 2.5])       # channel noise levels
P = 3.0                                       # total power budget

def water_fill(P, N, steps=60):
    lo, hi = N.min(), N.min() + P             # bracket for the water level
    for _ in range(steps):                    # bisection on the level theta
        mid = 0.5 * (lo + hi)
        lo, hi = (mid, hi) if np.maximum(0, mid - N).sum() < P else (lo, mid)
    theta = 0.5 * (lo + hi)
    return theta, np.maximum(0.0, theta - N)
```

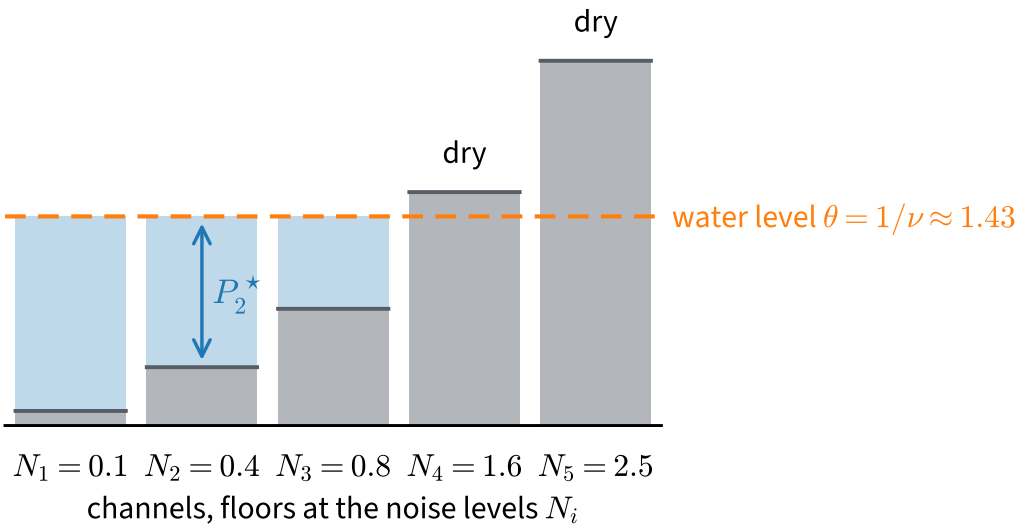


Figure 35.29: Water-filling for the noise floors and budget in the code cell below: pouring $P = 3$ units of power into basins with floors at N_i fills the three quietest channels to the common level $\theta = 1/\nu^\star \approx 1.43$, allocating $P_i^\star = \theta - N_i$ to each, while the two basins with floors above θ receive nothing, as complementary slackness requires.

```
theta, power = water_fill(P, N)
print('water level theta =', f'{theta:.6f}', ' P* =', power.round(4),
      ' total =', f'{power.sum():.6f}')
print('wet channels at the common level:', (N + power)[power > 0].round(6))
print('dry channels above the level:    ', bool((N[power == 0] >= theta).all()))
rate = lambda power: np.log(1.0 + power / N).sum()
dP = 1e-4                                       # shadow price, two ways
nu = 1.0 / theta
sens = (rate(water_fill(P + dP, N)[1]) - rate(water_fill(P - dP, N)[1])) / (2 * dP)
print(f'nu = 1/theta = {nu:.6f} vs finite-difference dU*/dP = {sens:.6f}')
```

```
water level theta = 1.433333  P* = [1.3333 1.0333 0.6333 0.     0.    ]  total = 3.000000
wet channels at the common level: [1.433333 1.433333 1.433333]
dry channels above the level:     True
nu = 1/theta = 0.697674 vs finite-difference dU*/dP = 0.697674
```

The power budget is met to machine precision, the three active channels all sit at the common level $\theta = (3 + 0.1 + 0.4 + 0.8)/3 \approx 1.4333$, and the noisy channels at 1.6 and 2.5 stay dry, just as dual feasibility requires. Moreover, the multiplier $\nu^\star = 1/\theta = 0.697674$ matches the finite-difference sensitivity $\mathrm{d}U^\star/\mathrm{d}P$ to all six printed digits.

35.7.5.3 An Explicit Duality Gap

Weak duality holds for every problem, but strong duality does not. A one-dimensional nonconvex example makes the gap concrete. Over the domain $\mathcal{D} = [0, 1]$, consider

$$\min_{x\in[0,1]} \; f(x) = -x^2 \quad \text{subject to} \quad g(x) = x - \tfrac{1}{2} \le 0.$$

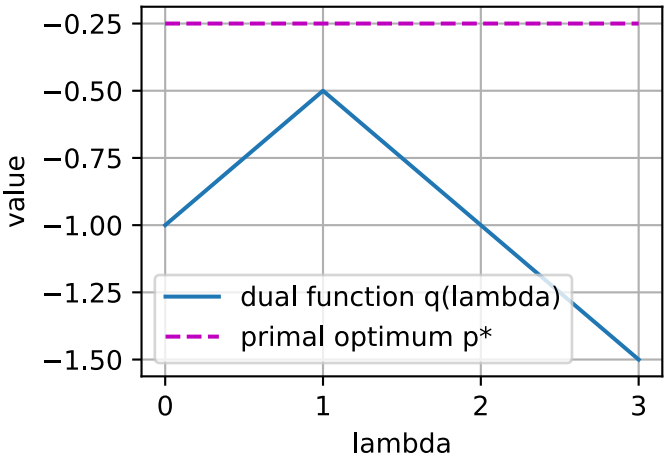
Because $f(x) = -x^2$ is concave rather than convex, the problem is nonconvex. The feasible set is $[0, \frac{1}{2}]$, and the minimum sits at the right endpoint, $p^\star = f(\frac{1}{2}) = -\frac{1}{4}$. We dualize only the constraint $g(x) \le 0$, keeping $\mathcal{D} = [0, 1]$ as a hard domain constraint so that the infimum in Equation 35.80 runs over $[0, 1]$ (weak duality applies unchanged). For any $\lambda \ge 0$, the Lagrangian $\mathcal{L}(x, \lambda) = -x^2 + \lambda(x - \frac{1}{2})$ is concave in x, so its minimum on $[0, 1]$ must occur at one of the two endpoints: $\mathcal{L}(0, \lambda) = -\lambda/2$ or $\mathcal{L}(1, \lambda) = \lambda/2 - 1$. Therefore,

$$q(\lambda) = \min\left(-\frac{\lambda}{2},\; \frac{\lambda}{2} - 1\right),$$

which is piecewise linear and concave, as guaranteed. The two lines intersect at $\lambda^\star = 1$, where $d^\star = -\frac{1}{2} < -\frac{1}{4} = p^\star$: a strict duality gap of $\frac{1}{4}$. Let us plot $q(\lambda)$ alongside $p^\star$ to compare this nonconvex dual curve with the clipping plot from the start of the section.

```
xs = np.linspace(0.0, 1.0, 4001)              # the domain D = [0, 1]
fx, gx = -xs**2, xs - 0.5                     # objective and constraint on D
p_star = fx[gx <= 0.0].min()                  # primal optimum: -1/4
lams = np.linspace(0.0, 3.0, 601)
q = np.array([(fx + lam * gx).min() for lam in lams])  # dual function on D
d_star, lam_star = q.max(), lams[q.argmax()]
print(f'p* = {p_star:.4f},  d* = {d_star:.4f} at lambda = {lam_star:.2f},'
      f'  gap = {p_star - d_star:.4f}')
d2l.plot(lams, [q, np.full_like(lams, p_star)], 'lambda', 'value',
         legend=['dual function q(lambda)', 'primal optimum p*'])
```

```
p* = -0.2500,  d* = -0.5000 at lambda = 1.00,  gap = 0.2500
```



Unlike the clipping curve, which rose smoothly to touch $p^\star$, this tent peaks at $-\frac{1}{2}$, a full quarter below the primal optimum. Notice that a strictly feasible point certainly exists—$x = \frac{1}{4}$ lies in the interior of $\mathcal{D}$ with $g(\frac{1}{4}) = -\frac{1}{4} < 0$—so the gap is caused entirely by the

lack of convexity, not by a failure of Slater's condition. Returning to panel (b) of Figure 35.26, the image curve $\{(u, -(u+\frac{1}{2})^2) : u \in [-\frac{1}{2}, \frac{1}{2}]\}$ bends downward, so no supporting line below $\mathcal{A}$ can reach $(0, p^\star)$; the highest one, of slope -1, certifies only $-\frac{1}{2}$.

Two open challenges remain. First, for a nonconvex problem, the dual can bound the optimum but need not reach it. Second, even when a saddle point exists, updating $\mathbf{x}$ and λ simultaneously can fail to converge, a central difficulty in minimax training. While training an entire deep network is nonconvex, many of its building blocks (projections, clipped updates, last-layer fits, and convex trust-region subproblems) are convex, and everything in this section applies directly to them. So far, however, every derivation in this chapter has assumed exact real arithmetic; in the next section we turn to what happens when finite floating-point precision, rather than the optimization algorithm itself, determines the answer.

35.7.6 Summary

For any constrained optimization problem, the Lagrange dual function $q(\lambda, \nu) = \inf_{\mathbf{x} \in \mathcal{D}} \mathcal{L}(\mathbf{x}, \lambda, \nu)$ is concave and satisfies weak duality, $q(\lambda, \nu) \le p^\star$, so every choice of $\lambda \succeq 0$ certifies a lower bound on the primal optimum. When the primal problem is convex and satisfies Slater's condition (a strictly feasible point for all non-affine inequalities), strong duality closes the gap ($d^\star = p^\star$) and attained primal–dual optima form a saddle point of the Lagrangian. For linear constraints, the dual reduces to the convex conjugate, $q(\lambda) = -f^*(-\mathbf{A}^\top \lambda) - \mathbf{b}^\top \lambda$, which turns conjugate pairs such as log-sum-exp and negative entropy directly into dual formulations. Under strong duality, the optimal multipliers act as shadow prices ($\lambda_i^\star = -\partial p^\star / \partial u_i$ and $\nu_j^\star = -\partial p^\star / \partial v_j$ wherever $p^\star$ is differentiable), explaining why weight decay matches a norm constraint for convex losses, why SVM multipliers vanish off the support vectors, and why the water-filling multiplier $1/\theta$ measures communication rate per unit of power. Without convexity, a strict duality gap can remain, as shown by minimizing $-x^2$ on $[0, 1]$ subject to $x \le \frac{1}{2}$.

35.7.7 Exercises

1. **Conjugate pairs.** Compute each conjugate from Equation 35.84.
    1. $f(\mathbf{x}) = \lambda \|\mathbf{x}\|_1$ with $\lambda > 0$.
    2. $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{A}\mathbf{x}$ with $\mathbf{A} \succ 0$. What does the Fenchel–Young inequality Equation 35.85 state for this pair?
    3. The softplus $f(t) = \log(1+e^t)$ on $\mathbb{R}$. The binary cross-entropy of a logit t against a soft label $p \in [0, 1]$ is $\log(1 + e^t) - pt$. Use Fenchel–Young to show that it is at least the binary entropy $H(p) = -p \log p - (1-p)\log(1-p)$, with equality exactly when $p = 1/(1+e^{-t})$, the sigmoid of t. Write the difference as a KL divergence.
    4. The double well $\phi_0(x) = (x^2-1)^2$ of Section 35.3.4 (the function ϕ there without its tilt $x/2$) is not convex. Find its convex envelope ϕ_0^{**} and the set on which $\phi_0^{**} < \phi_0$.

    Adapted from Boyd and Vandenberghe, Convex Optimization, Examples 3.22 and 3.26.

2. **Maximum entropy and temperature.** The entropy of a probability vector $\mathbf{p} \in \mathbb{R}^n$ is $H(\mathbf{p}) = -\sum_i p_i \log p_i$. This exercise derives the softmax from the constraint side.
    1. Maximize H subject to $\sum_i p_i = 1$ with one Lagrange multiplier, ignoring the

sign constraints $p_i \geq 0$. Show that the only stationary point is the uniform distribution. Why does ignoring the sign constraints not change the answer?

2. Fix scores $s_1, \dots, s_n$, not all equal, and a target $\bar{s}$ strictly between $\min_i s_i$ and $\max_i s_i$, and add the constraint $\sum_i p_i s_i = \bar{s}$. Show that the maximizer is $\mathbf{p} = \mathrm{softmax}(\beta \mathbf{s})$ for a scalar β determined by the multiplier of the score constraint. Show that the mean score under $\mathrm{softmax}(\beta \mathbf{s})$ increases strictly with β, so that exactly one β meets the target.
3. Let $H^\star(\bar{s})$ be the maximal entropy. Show that $\mathrm{d}H^\star/\mathrm{d}\bar{s} = -\beta$, so that the multiplier prices the score constraint as the multiplier of Section 35.6.1.3 prices the offset b. For which targets $\bar{s}$ is β negative? What happens to the softmax temperature $1/\beta$ as $\bar{s}$ approaches $\max_i s_i$?

3. **A dual certificate for ℓ_1 minimization.** Basis pursuit,

$$\min_{\mathbf{x}} \ \|\mathbf{x}\|_1 \quad \text{subject to} \quad \mathbf{A}\mathbf{x} = \mathbf{b}, \tag{35.92}$$

with $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $m < n$, selects a sparse solution of an underdetermined linear system. Its constraints are affine, so Slater's condition needs only a feasible point and gives $d^\star = p^\star$ whenever the system has a solution. The conjugate of $\|\cdot\|_1$ is the indicator of the box $\{\mathbf{y} : \|\mathbf{y}\|_\infty \leq 1\}$.

1. Use the equality form of Equation 35.86 to show that, after replacing ν by $-\nu$, the dual problem is $\max_\nu \mathbf{b}^\top \nu$ subject to $\|\mathbf{A}^\top \nu\|_\infty \leq 1$.
2. Solve both problems by hand for $\mathbf{A} = \begin{pmatrix} 1 & 2 \end{pmatrix}$ and $b = 2$, and check $d^\star = p^\star$.
3. Let $\mathbf{x}^\star$ and $\nu^\star$ be optimal. Show that $x_i^\star = 0$ wherever $|(\mathbf{A}^\top \nu^\star)_i| < 1$, and that $(\mathbf{A}^\top \nu^\star)_i = \mathrm{sign}(x_i^\star)$ wherever $x_i^\star \neq 0$. A candidate $\mathbf{x}$ with $\mathbf{A}\mathbf{x} = \mathbf{b}$ is given. Which conditions must a vector ν satisfy to certify that $\mathbf{x}$ is optimal without solving the problem?

4. **Support vectors and the dual value.** For the offset-folded SVM dual Equation 35.90, let $\alpha^\star$ be optimal and $\tilde{\mathbf{w}}^\star = \sum_i \alpha_i^\star y_i \tilde{\mathbf{x}}_i$.

1. Show that $\sum_i \alpha_i^\star = \|\tilde{\mathbf{w}}^\star\|^2$ by writing the KKT conditions of the dual over the orthant. Conclude that the dual optimal value equals the primal value $\frac{1}{2}\|\tilde{\mathbf{w}}^\star\|^2$.
2. Show that deleting a training point with $\alpha_j^\star = 0$ leaves the remaining multipliers optimal for the smaller problem and leaves $\tilde{\mathbf{w}}^\star$ unchanged. In leave-one-out validation, each of the n points is classified by the SVM trained on the other $n-1$ points. Conclude that the leave-one-out error rate is at most the fraction of points with $\alpha_i^\star > 0$, for every optimal $\alpha^\star$. Which optimal $\alpha^\star$ gives the best bound?
3. code The matrix $\mathbf{Q}$ of the SVM cell has rank 3. Let S be the four indices with $\alpha_i^\star > 0$ and $\mathbf{G}_S$ the corresponding rows of `G`. Find a nonzero $\mathbf{d} \in \mathbb{R}^4$ with $\mathbf{G}_S^\top \mathbf{d} = \mathbf{0}$ (use `np.linalg.svd` of $\mathbf{G}_S^\top$), and move $\alpha_S^\star + t\mathbf{d}$ in each direction until a coordinate reaches zero. Verify that both endpoints are dual feasible, have dual value $\frac{41}{98}$, and give the same $\tilde{\mathbf{w}}^\star$. Which margin-1 point has $\alpha_i = 0$ at each endpoint? What does this say about reading "support vector" off the multipliers? (You should find the endpoints $(0, 0.204, 0, 0, 0, 0.192, 0, 0.441)$ and $(0.204, 0, 0, 0, 0, 0.273, 0, 0.359)$.)

5. code **Gradient descent–ascent.** ● Primal–dual methods and GAN training update the minimizing variable by gradient descent and the maximizing variable by gradient ascent at the same time.

1. For the bilinear game $\min_x \max_y xy$, whose saddle point is the origin, write simultaneous descent–ascent, $x \leftarrow x - \eta y$ and $y \leftarrow y + \eta x$, as a linear map. Show that every nonzero start spirals away from the saddle for every $\eta > 0$. Show that the alternating version, in which the y-update uses the new x, is a linear map with determinant 1. What do its iterates do for $\eta < 2$?
2. Run simultaneous descent–ascent on the Lagrangian $\mathcal{L}(\mathbf{x}, \nu) = \frac{1}{2}\mathbf{x}^\top\mathbf{x} + \nu(\mathbf{a}^\top\mathbf{x} - b)$ of the closest-point problem of Section 35.6.1.3, here with $\mathbf{a} = (1, 2)$, $b = 1$, and the start $\mathbf{x} = \mathbf{0}$, $\nu = 0$. Find the step sizes for which the iterates converge to $(\mathbf{x}^\star, \nu^\star)$. Derive them from the eigenvalues of the update acting on the pair $(\mathbf{a}^\top\mathbf{x}/\|\mathbf{a}\|, \nu)$ and on the components of $\mathbf{x}$ orthogonal to $\mathbf{a}$.
3. Add $\frac{\epsilon}{2}x^2$ with $0 < \epsilon \leq 2$ to the objective of the bilinear game. For which step sizes does simultaneous descent–ascent converge now? Which term of the Lagrangian of the previous part plays the role of ϵ?

6. code **Penalty versus constraint without convexity.** ● For a convex loss, Section 35.7.3.2 shows that weight decay and a norm constraint trace the same solutions. Take instead the nonconvex one-dimensional loss $\ell(w) = (w-1)^2(w-3)^2 - w$, whose local minima lie near $w = 1.16$ and $w = 3.11$.

   1. On a fine grid of w, compute the penalized minimizers $w_\lambda = \operatorname{argmin}_w \ell(w) + \frac{\lambda}{2}w^2$ for λ on a logarithmic grid, and the constrained optimum $p^\star(r) = \min\{\ell(w) : w^2 \leq r^2\}$ for $r \in [0, 3.2]$. Plot the points $(w_\lambda^2, \ell(w_\lambda))$ on top of the curve $(r^2, p^\star(r))$. Which budgets r are never the norm $|w_\lambda|$ of a penalized minimizer?
   2. Show that w_λ minimizes $\ell(w) + \frac{\lambda}{2}w^2$ if and only if the line of slope $-\frac{\lambda}{2}$ through $(w_\lambda^2, \ell(w_\lambda))$ lies below every point of the set $\{(w^2, \ell(w)) : w \in \mathbb{R}\}$. Use this and the supporting-line picture of Figure 35.26 to explain the missing budgets.
   3. Pick a budget r in the missing range at which the constraint is active. Compute the KKT multiplier λ of the constrained solution, with the constraint written $\frac{1}{2}w^2 \leq \frac{1}{2}r^2$ as in the text, and find a point where the penalized objective with this λ is lower. Compute the duality gap $p^\star(r) - d^\star(r)$ of the constrained problem.

7. [extended] **Entropic optimal transport.** ●● Given probability vectors $\mathbf{p} \in \mathbb{R}^n$ and $\mathbf{r} \in \mathbb{R}^m$ and a cost matrix $\mathbf{C} \in \mathbb{R}^{n \times m}$, optimal transport finds the cheapest plan Γ that moves the mass $\mathbf{p}$ to the mass $\mathbf{r}$,

$$\mathrm{OT} = \min_{\Gamma \geq 0} \sum_{i,j} C_{ij}\Gamma_{ij} \quad \text{subject to} \quad \Gamma\mathbf{1} = \mathbf{p}, \;\; \Gamma^\top\mathbf{1} = \mathbf{r}, \tag{35.93}$$

   a linear program with nm variables. Entropic regularization (Cuturi 2013) adds $\varepsilon \sum_{i,j} \Gamma_{ij}(\log \Gamma_{ij} - 1)$ to the objective, with $\varepsilon > 0$; let Γ_ε be the regularized optimum. Its solution by Sinkhorn iterations is used to compare distributions (Peyré and Cuturi 2019).

   1. Form the Lagrangian with the terms $\phi^\top(\mathbf{p} - \Gamma\mathbf{1}) + \psi^\top(\mathbf{r} - \Gamma^\top\mathbf{1})$ for the marginal constraints. Show that $\Gamma_\varepsilon = \operatorname{diag}(\mathbf{u})\,\mathbf{K}\,\operatorname{diag}(\mathbf{v})$ with $K_{ij} = e^{-C_{ij}/\varepsilon}$, $u_i = e^{\phi_i/\varepsilon}$, and $v_j = e^{\psi_j/\varepsilon}$. Why can the sign constraints $\Gamma_{ij} \geq 0$ be left out of the Lagrangian?
   2. Show that the dual function is

$$q(\phi,\psi)=\mathbf{p}^\top\phi+\mathbf{r}^\top\psi-\varepsilon\sum_{i,j}e^{(\phi_i+\psi_j-C_{ij})/\varepsilon}, \tag{35.94}$$

and that maximizing it exactly over ϕ with ψ fixed gives the Sinkhorn update $\mathbf{u}=\mathbf{p}/(\mathbf{K}\mathbf{v})$, elementwise. Conclude that Sinkhorn's algorithm is block-coordinate ascent on a concave function and that its dual value never decreases. Why is every dual value a lower bound on OT?

3. Let $H(\Gamma)=-\sum_{i,j}\Gamma_{ij}\log\Gamma_{ij}$, and grant that every feasible plan satisfies $\max(H(\mathbf{p}),H(\mathbf{r}))\le H(\Gamma)\le H(\mathbf{p})+H(\mathbf{r})$. Show that $0\le\sum_{i,j}C_{ij}(\Gamma_\varepsilon)_{ij}-\mathrm{OT}\le\varepsilon\min(H(\mathbf{p}),H(\mathbf{r}))$.
4. Let the costs lie in $[0,1]$. In float64, e^{-x} rounds to zero for x above about 745. For which ε can entries of $\mathbf{K}$ vanish, and what then happens to the multiplicative updates $\mathbf{u}=\mathbf{p}/(\mathbf{K}\mathbf{v})$? Explain why updating ϕ and ψ directly, with a log-sum-exp in place of the products with $\mathbf{K}$, avoids the problem.

8. code **The dual of ridge regression.** Write $\min_{\mathbf{w}}\frac{1}{2}\|\mathbf{A}\mathbf{w}-\mathbf{b}\|^2+\frac{\lambda}{2}\|\mathbf{w}\|^2$, with $\mathbf{A}\in\mathbb{R}^{n\times d}$, as $\min_{\mathbf{w},\mathbf{z}}\frac{1}{2}\|\mathbf{z}\|^2+\frac{\lambda}{2}\|\mathbf{w}\|^2$ subject to $\mathbf{A}\mathbf{w}-\mathbf{b}-\mathbf{z}=\mathbf{0}$, with multiplier $\alpha\in\mathbb{R}^n$.
    1. Minimize the Lagrangian over $\mathbf{z}$ and $\mathbf{w}$ to get $\mathbf{z}=\alpha$, $\mathbf{w}=-\mathbf{A}^\top\alpha/\lambda$, and the dual function $q(\alpha)=-\frac{1}{2}\|\alpha\|^2-\frac{1}{2\lambda}\alpha^\top\mathbf{A}\mathbf{A}^\top\alpha-\mathbf{b}^\top\alpha$. Why does strong duality hold?
    2. Maximize q to get $\alpha^\star=-\lambda(\mathbf{A}\mathbf{A}^\top+\lambda\mathbf{I})^{-1}\mathbf{b}$ and $\mathbf{w}^\star=\mathbf{A}^\top(\mathbf{A}\mathbf{A}^\top+\lambda\mathbf{I})^{-1}\mathbf{b}$. Explain why the solution is a combination of the rows of $\mathbf{A}$, as the SVM weight vector is a combination of the data points, and why the data enter only through $\mathbf{A}\mathbf{A}^\top$.
    3. For $n=20$ examples and $d=50$ features, verify numerically that the primal solve $(\mathbf{A}^\top\mathbf{A}+\lambda\mathbf{I})\mathbf{w}=\mathbf{A}^\top\mathbf{b}$ and the dual formula agree to about 10^{-14}, and that the primal and dual values coincide. Which solve is cheaper when $n\ll d$? What does replacing $\mathbf{A}\mathbf{A}^\top$ by a kernel matrix give?

35.8 Numerical Stability and Conditioning

Every computation in this chapter so far ran in floating point, and we read its results as if they were exact real numbers. Most of the time, that convenient fiction holds up. Occasionally, however, a training loss turns into `NaN`, or a linear solver returns an answer that looks completely plausible and is badly wrong. In this section we ask when finite-precision arithmetic, rather than the optimization algorithm, decides the answer—and what we can do about it.

Two small examples show the two ways numerical trouble arrives. First, the three logits $(1000, 1001, 1002)$ define an entirely ordinary softmax, with probabilities $(0.09, 0.24, 0.67)$. Yet if we evaluate the textbook formula as written, it returns `NaN` three times, even in double precision (fp64). Second, `np.linalg.solve` applied to a 12×12 linear system returns an answer that is off by 30%, even though it is the exact solution of a system that differs from the stated one only in the seventeenth decimal digit. The first failure halts training with a `NaN` loss; the second slips past unnoticed.

These two failures have fundamentally different causes. The naive softmax formula passes through e^{1000}, a number too large for any standard floating-point format to hold, whereas an algebraically equivalent rewrite that never forms e^{1000} is accurate to a few units in the

last place. By contrast, the 12×12 linear system is so sensitive that a rounding error in the sixteenth digit of its data moves the solution in the first digit, and no rewriting of the solver can undo that sensitivity.

To tell those two situations apart, we begin by examining how computers store real numbers and how much error each rounding introduces. Next, we rewrite softmax and cross-entropy so they never pass through numbers the format cannot hold, and we rewrite the variance so it never subtracts nearly equal large numbers. We then introduce a sensitivity measure—the condition number of a function—that separates a bad algorithm from an inherently hard problem. Finally, we see how the condition number of a linear system governs both the digits a least-squares solve can keep and, as in Section 35.1, the number of steps gradient descent requires. While Section 5.5 shows how modern training uses low-precision formats in practice, here we derive why computations in them succeed or fail.

Three ideas matter more than any individual formula: the relative rounding model for floating-point arithmetic, the principle that an unstable algorithm routes a well-conditioned problem through an ill-conditioned intermediate step, and the bound linking a solver's solution error to the problem's sensitivity. The subsections on summation order and ridge regularization can be skipped on a first reading.

```
from fractions import Fraction
```

35.8.1 Floating-Point Arithmetic

35.8.1.1 Representation and Spacing

What does a computer actually store when we ask it to store 1.1? The mechanism is easiest to see in a toy binary format that keeps only two bits after the binary point. Between 1 and 2, this format can hold just four values: $(1.00)_2 = 1$, $(1.01)_2 = 1.25$, $(1.10)_2 = 1.5$, and $(1.11)_2 = 1.75$. Multiplying these four values by powers of two fills out the other intervals: $2, 2.5, 3, 3.5$ between 2 and 4, and $0.5, 0.625, 0.75, 0.875$ between 0.5 and 1. Each *octave*—an interval $[2^e, 2^{e+1})$ between consecutive powers of two—holds exactly four values, so the absolute gap between adjacent numbers doubles every time the magnitude doubles.

Any real number that falls between grid points is rounded to the nearest one: $1.1 \mapsto 1$ (a relative error of -0.091), $1.3 \mapsto 1.25$ (an error of -0.038), and $3.3 \mapsto 3.5$ (an error of $+0.061$). Halfway points are ties, and standard arithmetic breaks ties toward the neighbor whose last bit is 0: the midpoint 1.125 therefore rounds to 1, an error of -0.111. Notice that no relative error exceeds $1/8$. Rounding moves a number by at most half a gap, and in every octave half a gap is $1/8$ of the octave's smallest value. Figure 35.30 sketches this grid on the real line.

Real floating-point formats work in the same way, simply with more bits. A floating-point number is scientific notation in base two with a fixed budget of p binary digits after the binary point,

$$x = (-1)^s \cdot (1.b_1 b_2 \ldots b_p)_2 \cdot 2^e, \tag{35.95}$$

specified by a sign bit s, *mantissa* (or significand) bits $b_1, \ldots, b_p$, and an integer exponent e drawn from a fixed range. The exponent sets the range of magnitudes the format can

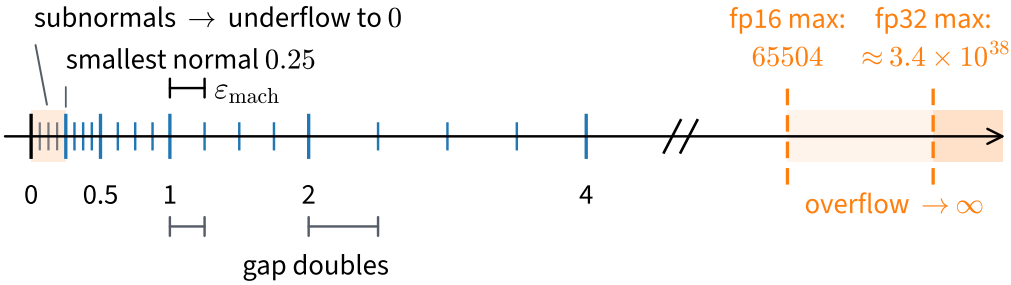


Figure 35.30: Floating-point numbers on the real line for a toy format with two mantissa bits (four values per octave, $\varepsilon_{\text{mach}} = 1/4$, smallest normal 0.25). The absolute gap between neighbors doubles at every power of two while the relative gap stays near $\varepsilon_{\text{mach}}$. Below the smallest normal, subnormal numbers continue with the same spacing down to zero; after the axis break, real formats end at their overflow thresholds (fp16 at 65504, fp32 at about 3.4×10^{38}).

represent, while the mantissa sets its relative precision. Within the octave $[2^e, 2^{e+1})$, adjacent values are spaced 2^{e-p} apart. The gap from 1 to the next representable number, $\varepsilon_{\text{mach}} = 2^{-p}$, is called *machine epsilon*; our toy format has $p = 2$ and $\varepsilon_{\text{mach}} = 1/4$.

Rounding a real number x to the nearest representable value moves it by at most half a gap, 2^{e-p-1}. Because $|x| \geq 2^e$ throughout the octave, the relative error is at most 2^{-p-1}. Calling a value *normal* when it has the form Equation 35.95 with an in-range exponent, every real x that rounds to a normal number satisfies

$$\text{fl}(x) = x\,(1+\delta), \qquad |\delta| \leq u = \tfrac{1}{2}\,\varepsilon_{\text{mach}} = 2^{-p-1}. \tag{35.96}$$

In words, each rounding perturbs a number by at most one part in 2^{p+1}: one part in 8 for our toy format, about one part in 17 million for fp32 ($p = 23$), and one part in 256 for bf16 ($p = 7$). The bound u is called the *unit roundoff*. Standard IEEE arithmetic rounds the result of every basic operation to the nearest representable value, so the computed sum $x \oplus y$ equals $(x + y)(1 + \delta)$ with $|\delta| \leq u$, and the same guarantee holds for $-$, $\times$, and $/$ (IEEE 2019; Goldberg 1991).

This clean relative model has three exceptions. Results larger than the format can hold *overflow* to ∞. Results smaller than the smallest normal number *underflow*, stepping down through *subnormal* numbers that fill the remaining gap to zero. Undefined operations, such as $0/0$ or ∞/∞, produce `NaN`. All three are absolute-scale effects rather than relative ones, and we will return to them right after putting the standard formats on the table.

35.8.1.2 Formats

Deep learning relies on five formats narrower than double precision (fp64). Let us inspect each format's machine epsilon $\varepsilon_{\text{mach}}$, smallest normal number, and largest finite value directly from the library:

```
print(f'{"dtype":>14} {"eps":>10} {"smallest normal":>16} {"max":>10}')
for dt in [torch.float16, torch.float32, torch.bfloat16,
           torch.float8_e4m3fn, torch.float8_e5m2]:
    fi = torch.finfo(dt)
    print(f'{str(dt).removeprefix("torch."):>14} {fi.eps:10.3e} '
          f'{fi.smallest_normal:16.3e} {fi.max:10.3e}')
```

```
         dtype        eps  smallest normal        max
       float16  9.766e-04        6.104e-05  6.550e+04
       float32  1.192e-07        1.175e-38  3.403e+38
      bfloat16  7.812e-03        1.175e-38  3.390e+38
  float8_e4m3fn  1.250e-01        1.562e-02  4.480e+02
    float8_e5m2  2.500e-01        6.104e-05  5.734e+04
```

fp32 keeps $p = 23$ mantissa bits (about seven decimal digits) and reaches up to 3.4×10^{38}. **fp16** keeps $p = 10$ bits (about three decimal digits) but has a narrow exponent range: it overflows above 65504, and below $2^{-14} \approx 6.1 \times 10^{-5}$ its values become subnormal. **bf16** (`bfloat16` in the libraries) makes the opposite trade-off. It keeps all eight exponent bits of fp32, preserving its vast dynamic range, at the cost of keeping only $p = 7$ mantissa bits ($\varepsilon_{\text{mach}} = 2^{-7} \approx 0.0078$, roughly two to three decimal digits). Notice that the value 2^{-8} often quoted for bf16 is its unit roundoff $u = \varepsilon_{\text{mach}}/2$. The two fp8 formats push this trade-off further still: E4M3 keeps three mantissa bits (about one decimal digit) and tops out at 448, while E5M2 keeps two mantissa bits with fp16's exponent range. With so little dynamic range, fp8 is practical only when paired with a separate scale factor per tensor or per block of values.

Because bf16 is simply the top 16 bits of an fp32 number—the same sign bit, the same eight exponent bits, and the first seven mantissa bits—converting fp32 to bf16 amounts to rounding off the low 16 bits. We can implement that rounding directly with integer arithmetic on the raw bit pattern. The midpoint of the discarded 16 bits is `0x8000`, so we add `0x7FFF` (one below the midpoint), plus one more if the lowest kept bit is 1, and then clear the low 16 bits. This sum carries into the kept bits if and only if the discarded part lies strictly above the midpoint, or sits right on the midpoint with an odd kept bit. That is round-to-nearest, ties-to-even in a single line of bit manipulation; Exercise 5 builds stochastic rounding on top of it. Let us verify it on one input above a midpoint and two exact ties:

```
def to_bf16(x):
    """Round float32 to the nearest bfloat16, ties to even."""
    bits = np.atleast_1d(np.asarray(x, np.float32)).view(np.uint32)
    bits = (bits + 0x7FFF + ((bits >> 16) & 1)) & 0xFFFF0000
    return bits.astype(np.uint32).view(np.float32)

for x, expected in [(1 + 2**-8 + 2**-10, 1 + 2**-7),   # above the midpoint
                    (1 + 2**-8, 1.0),                  # tie, 1 is even
                    (1 + 3 * 2**-8, 1 + 2**-6)]:       # tie, 1 + 2^-6 is even
    print(f'{x:.8f} -> {to_bf16(x).item():.8f}   (expected {expected:.8f})')
```

```
1.00488281 -> 1.00781250   (expected 1.00781250)
1.00390625 -> 1.00000000   (expected 1.00000000)
1.01171875 -> 1.01562500   (expected 1.01562500)
```

As expected, the first input lies above the midpoint between 1 and $1 + 2^{-7}$ and rounds up.

The second sits right on that midpoint and rounds down to the even neighbor 1, whereas the third sits on the midpoint between $1+2^{-7}$ and $1+2^{-6}$ and rounds up to the even neighbor $1+2^{-6}$. Next, let us check the same tie rule in fp32, observe how the spacing grows with magnitude, and find the largest logit each format can exponentiate without overflow:

```
eps = np.finfo(np.float32).eps
one = np.float32(1.0)
print('1 + eps   != 1 :', one + eps != one)
print('1 + eps/2 == 1 :', one + np.float32(eps / 2) == one)
print('gap between float32 neighbors at 1    :', np.spacing(one))
print('gap between float32 neighbors at 2^20 :', np.spacing(np.float32(2.0**20)))
for dt in [np.float16, np.float32, np.float64]:
    print(f'{np.dtype(dt).name}: exp(x) overflows for x >',
          f'{np.log(np.finfo(dt).max):.2f}')
```

```
1 + eps   != 1 : True
1 + eps/2 == 1 : True
gap between float32 neighbors at 1    : 1.1920929e-07
gap between float32 neighbors at 2^20 : 0.125
float16: exp(x) overflows for x > 11.09
float32: exp(x) overflows for x > 88.72
float64: exp(x) overflows for x > 709.78
```

Here $1+\varepsilon_{\text{mach}}/2$ is the midpoint between 1 and its upper neighbor, so ties-to-even sends it back to 1. The worst-case rounding error of half a gap really does occur, giving a relative error of $u/(1+u)$, just inside the bound $|\delta| \le u$. Meanwhile, the gap at 2^{20} is 2^{20} times (about a million times) the gap at 1. Finally, the last three lines pinpoint where e^x overflows: above 11.09 in fp16, 88.72 in fp32, and 709.78 in fp64.

35.8.1.3 Overflow, Underflow, and Small Updates

Because the exponential turns an additive scale into a multiplicative one, even modest logits can overflow. In fp16, the ceiling $\log 65504 \approx 11.09$ sits well inside the range of ordinary unnormalized classifier scores.

At the opposite end, e^{-x} underflows. Below the smallest normal number, a format steps through subnormal numbers that keep the same absolute spacing while shedding significant bits one by one, until values round all the way to 0. Taking $\log 0$ later produces $-\infty$, and the backward pass fills with `inf` or `NaN`. In fp16, subnormals begin below $2^{-14} \approx 6.1\times10^{-5}$, the smallest subnormal is $2^{-24} \approx 6\times10^{-8}$, and anything below $2^{-25} \approx 3\times10^{-8}$ rounds straight to zero.

Addition introduces a second absolute threshold that matters directly for training. A gradient step replaces a weight w by $w-\eta g$. If the update ηg is small compared with the spacing around w, the sum simply rounds back to w, and the step vanishes without a trace.

Proposition (small updates vanish). *Let $w > 0$ be a floating-point number with $2^e \le w < 2^{e+1}$ in a format with p mantissa bits, and let Δ be real. If $|\Delta| < 2^{e-p-1}$, then $\mathrm{fl}(w+\Delta) = w$, except when $w = 2^e$ and $\Delta < 0$; in that case the same holds for $|\Delta| < 2^{e-p-2}$.*

Proof. In $[2^e, 2^{e+1}]$ the representable numbers are 2^{e-p} apart, so the neighbors of w are $w \pm 2^{e-p}$, except that below $w = 2^e$ the lower octave $[2^{e-1}, 2^e)$ begins and the spacing halves to 2^{e-p-1}. Rounding to the nearest number returns w whenever $w+\Delta$ is closer to

w than to the adjacent grid point on its side—that is, whenever $|\Delta|$ is strictly less than half of that gap. ■

For a weight update $\Delta = -\eta g$ with $2^e < w < 2^{e+1}$, the proposition reads

$$\mathrm{fl}(w - \eta g) = w \qquad \text{whenever } |\eta g| < 2^{e-p-1} = u\,2^e. \tag{35.97}$$

In short, any update smaller than roughly $u|w|$ is lost. The relative model Equation 35.96 describes products and quotients well, but when we add two numbers of very different magnitudes, it hides this sharp absolute cutoff.

A weight sitting right on a power of two, such as $w = 1$, falls into the exceptional case of the proposition: because the spacing halves below 1, an increase vanishes below u (as Section 5.5 tests), whereas a decrease already vanishes below $u/2$. In bf16, a weight equal to 1 therefore ignores any decrease smaller than $2^{-9} \approx 0.002$, so a learning rate of 10^{-3} with an order-1 gradient leaves the weight completely frozen. The corresponding threshold is $2^{-12} \approx 2.4 \times 10^{-4}$ in fp16 and $2^{-25} \approx 3 \times 10^{-8}$ in fp32—which is why mixed-precision training keeps a master copy of the weights in fp32. Let us test steps straddling these thresholds (computing the bf16 updates in fp32 and rounding with `to_bf16`):

```
one32, one16 = np.float32(1.0), np.float16(1.0)
print('bfloat16: 1 - 1e-3 ->', to_bf16(one32 - np.float32(1e-3)).item())
print('bfloat16: 1 - 2e-3 ->', to_bf16(one32 - np.float32(2e-3)).item())
print('fp16    : 1 - 1e-4 ->', one16 - np.float16(1e-4))
print('fp16    : 1 - 3e-4 ->', one16 - np.float16(3e-4))
print('fp32    : 1 - 1e-3 ->', one32 - np.float32(1e-3))
```

```
bfloat16: 1 - 1e-3 -> 1.0
bfloat16: 1 - 2e-3 -> 0.99609375
fp16    : 1 - 1e-4 -> 1.0
fp16    : 1 - 3e-4 -> 0.9995
fp32    : 1 - 1e-3 -> 0.999
```

Sure enough, a step of 10^{-3} leaves the bf16 weight stuck at 1, whereas 2×10^{-3} clears the 0.00195 threshold and jumps to the neighbor $1 - 2^{-8} = 0.99609375$. Similarly, fp16 swallows 10^{-4} and responds to 3×10^{-4} by stepping to $1 - 2^{-11} \approx 0.9995$, while fp32 keeps the full update.

Gradients run into the underflow threshold instead. In fp16, gradients below 6.1×10^{-5} lose precision and those below 3×10^{-8} vanish to zero. Multiplying the loss by a constant factor S before the backward pass scales every gradient up by S and lifts them back into the normal range; dividing by S afterward in fp32 restores the true scale (Section 5.5.3.3 covers the practical workflow, and Exercise 1 determines the admissible window for S).

Stochastic rounding (Gupta et al. 2015) removes the lost-update cutoff in a different way. Instead of always rounding to the nearest grid point, it rounds up or down at random with probabilities chosen so that the expected stored value equals the exact result. Updates far too small for round-to-nearest then still accumulate over many steps in expectation (Exercise 5).

35.8.2 Stable Softmax and Cross-Entropy

Softmax, log-sum-exp, and cross-entropy all exponentiate logits, so all three collide head-on with the overflow and underflow thresholds we just computed. To keep everything concrete, we will carry one running example through this subsection: a two-class classifier with logits $\mathbf{z} = (0, t)$ whose true label is the unlikely class 0. Its probabilities are $\text{softmax}(\mathbf{z}) = (1/(1+e^t),\, e^t/(1+e^t))$, its log-sum-exp is $\log(1+e^t)$, and the cross-entropy loss for class 0 is $\log(1+e^t) \approx t$ for large t. None of these outputs is extreme: even at $t = 100$, the probabilities are 3.7×10^{-44} and 1, and the loss is simply 100.

35.8.2.1 The Max Shift

Given logits $\mathbf{z} \in \mathbb{R}^n$, the softmax vector is defined by

$$\text{softmax}(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}.$$

At $t = 100$, evaluating this formula directly forms e^{100}, which exceeds fp32's overflow ceiling of 88.72. The second entry turns into `inf/inf` (`NaN`), and the first collapses to $1/\infty = 0$. Notice that the true probabilities themselves fit easily in fp32; only the intermediate exponential does not.

Luckily, multiplying numerator and denominator by the same positive constant e^{-c} leaves the ratio untouched, so softmax is *shift-invariant*:

$$\text{softmax}(\mathbf{z} - c\mathbf{1}) = \text{softmax}(\mathbf{z}) \qquad \text{for every } c \in \mathbb{R}. \tag{35.98}$$

Factoring e^c out of the sum inside a logarithm gives the companion identity for **log-sum-exp**, $\text{lse}(\mathbf{z}) = \log \sum_j e^{z_j}$, which is the logarithm of the softmax denominator:

$$\text{lse}(\mathbf{z}) = c + \log \sum_{j=1}^{n} e^{z_j - c}. \tag{35.99}$$

Both identities hold for any $c \in \mathbb{R}$, and choosing $c = \max_j z_j$ makes both computations safe. Every shifted exponent $z_j - c$ is then at most 0, so no exponential can overflow. At the same time, the largest term equals $e^0 = 1$, so the sum is guaranteed to lie in $[1, n]$ and its logarithm cannot underflow to $-\infty$. Let us test this on $(0, 100)$ in fp32, on the opening's $(1000, 1001, 1002)$ in fp64, and on $(0, 10)$ where both versions stay finite—do they return the exact same bits there?

```
def softmax_naive(z):
    e = np.exp(z)
    return e / e.sum()

def softmax_stable(z):
    e = np.exp(z - z.max())        # shift by the max: largest exponent is 0
    return e / e.sum()

z = np.array([0.0, 100.0], dtype=np.float32)
```

```
    with np.errstate(over='ignore', invalid='ignore'):
        print('naive  softmax(0, 100):', softmax_naive(z))
    print('stable softmax(0, 100):', softmax_stable(z))
    z = np.array([1000.0, 1001.0, 1002.0])        # the opening's logits, fp64
    with np.errstate(over='ignore', invalid='ignore'):
        print('naive  softmax(1000, 1001, 1002):', softmax_naive(z))
    print('stable softmax(1000, 1001, 1002):', softmax_stable(z).round(4))
    z = np.array([0.0, 10.0], dtype=np.float32)
    a, b = softmax_naive(z), softmax_stable(z)
    print('softmax(0, 10), naive minus stable, in ulps:',
          (a - b) / np.spacing(b))
```

```
naive  softmax(0, 100): [ 0. nan]
stable softmax(0, 100): [3.8e-44 1.0e+00]
naive  softmax(1000, 1001, 1002): [nan nan nan]
stable softmax(1000, 1001, 1002): [0.09   0.2447 0.6652]
softmax(0, 10), naive minus stable, in ulps: [2. 0.]
```

At $t = 100$, the shifted softmax returns $(3.8 \times 10^{-44}, 1)$, the true 3.7×10^{-44} rounded onto the subnormal grid with five significant bits left, while the direct formula returns 0 and NaN. Switching to higher precision does not cure the problem: the logits $(1000, 1001, 1002)$ from the opening overflow e^{1000} even in fp64, turning the direct formula into three NaNs where the shifted version cleanly returns $(0.09, 0.24, 0.67)$. At $t = 10$, the two formulas either agree to the last bit or differ by one or two units in the last place (*ulps*), because real-number identities do not guarantee identical rounding orders. One edge case remains: the max shift cannot rescue non-finite logits—if an entire row is $-\infty$, then $c = -\infty$ and $z_j - c$ is undefined.

35.8.2.2 Bounds for Log-Sum-Exp

Log-sum-exp is the log-normalizer of every exponential-family model (Section 36.4), and up to a single subtraction it is the cross-entropy loss. The shift Equation 35.99 immediately tells us how close it sits to the largest logit:

Proposition (log-sum-exp sandwich). *For every* $\mathbf{z} \in \mathbb{R}^n$,

$$\max_j z_j \;\le\; \operatorname{lse}(\mathbf{z}) \;\le\; \max_j z_j + \log n.$$

Proof. Set $c = \max_j z_j$ in Equation 35.99. Every exponential $e^{z_j - c}$ lies in $(0, 1]$ and the maximizing term equals 1, so the sum lies in $[1, n]$ and its logarithm in $[0, \log n]$. Adding c yields both bounds. ■

Log-sum-exp is therefore a *soft maximum*. In our two-class example, $\operatorname{lse}(0, t) = \log(1 + e^t)$ stays within $\log 2 \approx 0.69$ of $\max(0, t)$, attaining the full gap at $t = 0$ where the two logits tie and hugging the maximum closely as $|t|$ grows (Figure 35.31).

Moreover, its gradient is the softmax itself, $\partial \operatorname{lse}(\mathbf{z})/\partial z_i = e^{z_i} / \sum_j e^{z_j} = \operatorname{softmax}(\mathbf{z})_i$. That is why the gradient of cross-entropy with respect to the logits, for label y, simplifies to $\operatorname{softmax}(\mathbf{z}) - \mathbf{e}_y$; Section 35.3 also shows that log-sum-exp is convex.

Subtracting the shifted log-sum-exp from the shifted logit yields the **log-softmax**, computing log-probabilities directly without ever forming the probabilities themselves:

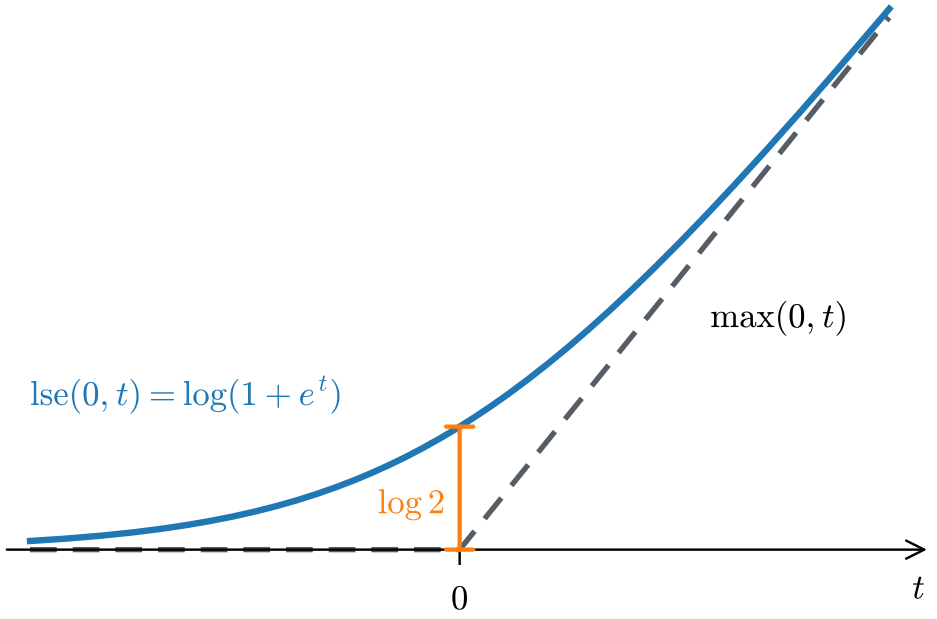


Figure 35.31: Log-sum-exp as a soft maximum: $\mathrm{lse}(0,t) = \log(1+e^t)$ stays within $\log 2$ of $\max(0,t)$, matching the sandwich bound for $n = 2$. The gap peaks at $t = 0$, where the two logits tie, and vanishes as $|t|$ grows.

$$\log \mathrm{softmax}(\mathbf{z})_i = (z_i - c) - \log \sum_{j=1}^{n} e^{z_j - c}, \qquad c = \max_j z_j. \tag{35.100}$$

Because we never exponentiate the final result, a tiny probability cannot underflow to zero before its logarithm is taken. Notice a subtle detail in Equation 35.100: the algebraically equivalent form $z_i - \mathrm{lse}(\mathbf{z})$ is noticeably worse in floating point, because it first forms the large number $\mathrm{lse} = c + \log \sum_j e^{z_j - c}$ and then subtracts it from the large number z_i. How much difference does that parenthesization make? Let us compare both forms in fp32 on the logits $(1000, 1001, 1002)$:

```
z = np.array([1000.0, 1001.0, 1002.0], dtype=np.float32)
c = z.max()
s = np.log(np.exp(z - c).sum())          # log sum exp(z - c), in [0, log 3]
with np.errstate(over='ignore'):
    print('naive log(sum(exp(z))):', np.log(np.exp(z).sum()))
lse = c + s
print('lse = c + s           :', lse, '  float32 gap at 1002:', np.spacing(c))
log_p_sub = z - lse                      # forms lse, then subtracts it
log_p = (z - c) - s                      # never forms c + s
print(f'sum of exp(z - lse)    : {np.exp(log_p_sub).sum():.8f}')
print(f'sum of exp((z - c) - s): {np.exp(log_p).sum():.8f}')
```

```
naive log(sum(exp(z))): inf
lse = c + s           : 1002.4076   float32 gap at 1002: 6.1035156e-05
sum of exp(z - lse)    : 1.00001323
sum of exp((z - c) - s): 1.00000000
```

The direct computation overflows to `inf`—as it would even in fp64, since $1000 > 709.78$, which is what turned `inf/inf` into three `NaN`s in the opening. The shifted value lse $= 1002.4076$ sits comfortably inside the sandwich $[1002, 1002 + \log 3]$. More interestingly, the two log-softmax forms disagree: exponentiating $z_i - \text{lse}$ gives probabilities that sum to 1.0000132, whereas Equation 35.100 sums to 1.0000000.

Why did $z_i - \text{lse}$ lose accuracy? Surprisingly, the subtraction itself made no error at all. Whenever two floating-point numbers lie within a factor of two of each other, their difference is exactly representable, because both are multiples of the smaller number's spacing and their difference needs no extra bits (Sterbenz's lemma, Exercise 6).

The error entered earlier, when lse was stored. Near 1002, fp32 numbers are $2^{-14} \approx 6.1 \times 10^{-5}$ apart, so the true value 1002.4076060 rounds down by 1.32×10^{-5}. Every log-probability computed as $z_i - \text{lse}$ is therefore 1.32×10^{-5} too high, every resulting probability is inflated by the factor $e^{1.32 \times 10^{-5}}$, and their sum overshoots 1 by 1.32×10^{-5}. An absolute error in an exponent turns into a relative error of the same size after exponentiation, and the subtraction merely exposed an error that was already sitting in its input—the central theme of Section 35.8.3.

35.8.2.3 Cross-Entropy from Logits

The cross-entropy loss for label y is simply the negative log-softmax, so Equation 35.100 gives it directly from the logits:

$$-\log \text{softmax}(\mathbf{z})_y = \text{lse}(\mathbf{z}) - z_y. \tag{35.101}$$

Computing the loss this way requires one stable log-sum-exp and one subtraction, without ever forming a probability. On our running example, the loss for class 0 is $\text{lse}(0, t) - 0 = \log(1 + e^t)$. If we instead compute probabilities first, the true-class probability $e^{-t}/(1 + e^{-t}) \approx e^{-t}$ drops below fp32's normal range once $t > 87.3$ and underflows completely to zero near $t = 104$, sending its negative logarithm to ∞. Let us compare the two routes using the library's built-in functions, with $t = 103$ and 104 straddling that cliff:

```
print('    t  CE from logits    CE via probabilities')
for t in [20.0, 60.0, 103.0, 104.0]:
    logits = torch.tensor([[0.0, t]])
    y = torch.tensor([0])                       # label = the unlikely class
    from_logits = F.cross_entropy(logits, y)
    probs = F.softmax(logits, dim=1)            # stable softmax, then...
    from_probs = -torch.log(probs[0, y])        # ...take the log yourself
    print(f'{t:5.0f}  {from_logits.item():15.4f}  {from_probs.item():15.4f}')
```

```
    t  CE from logits    CE via probabilities
   20          20.0000          20.0000
   60          60.0000          60.0000
  103         103.0000         103.2789
  104         104.0000              inf
```

Computed directly from logits, the loss equals t in every row, and the probability route matches it at $t = 20$ and 60. At $t = 103$, however, the probability e^{-103} falls into the subnormal range, and what happens next depends on how the library handles subnormal numbers.

PyTorch preserves subnormals: $e^{-103} \approx 1.32 \times 2^{-149}$ rounds to the smallest fp32 subnormal, 2^{-149}. Only a single significant bit survives, so the loss reads $149 \log 2 = 103.2789$—wrong in the first decimal place with no warning. At $t = 104$, $e^{-104} < 2^{-150}$ rounds to 0, and the loss becomes `inf`.

The practical takeaway is simple: keep losses, likelihoods, and posteriors in log space until you actually need a probability. The fused loss in Section 3.3.5.2 implements Equation 35.101 for this reason, Section 37.1 interprets what cross-entropy measures, and Section 36.11 sums hundreds of per-pixel log-probabilities to avoid the same underflow.

35.8.3 Catastrophic Cancellation

35.8.3.1 The Condition Number of a Function

In fp32, evaluating $\log(1 + x)$ at $x = 10^{-8}$ returns 0—a relative error of 100%—whereas the library function `log1p` returns 10^{-8}. Is $\log(1 + x)$ near $x = 0$ inherently hard to compute, or is the formula to blame? To answer questions like this systematically, we need a quantity that measures how strongly a function amplifies errors in its input, independent of any particular algorithm.

The **relative condition number** of a differentiable function f at a point x with $f(x) \neq 0$ is the factor by which a small relative perturbation in x turns into a relative perturbation in $f(x)$. Perturbing x to $x(1 + \delta)$ changes the output, to first order, to $f(x(1 + \delta)) \approx f(x) + x f'(x)\,\delta$. Dividing the change by $f(x)$ gives the amplification factor

$$\kappa_f(x) = \left| \frac{x\, f'(x)}{f(x)} \right|. \tag{35.102}$$

For example, the exponential has $\kappa_f(x) = |x|$: at $x = 88$, an input error of one unit roundoff becomes 88 units in the output. That is the exact mechanism we just saw in the log-softmax cell, where an absolute error in an exponent became a relative error of the same size in the probability. Notice that κ_f is a first-order property of the exact mathematical function. It says nothing about how f is evaluated, and it breaks down where $f(x) = 0$, since relative error is undefined there.

Subtraction takes two inputs, and the same first-order reasoning applies to both. Suppose $\widehat{a} = a(1 + \delta_a)$ and $\widehat{b} = b(1 + \delta_b)$ carry relative errors $|\delta_a|, |\delta_b| \leq u$, and suppose the subtraction itself is exact. Then $\widehat{a} - \widehat{b} = (a - b) + (a\delta_a - b\delta_b)$, so the triangle inequality bounds the relative error of the difference by

$$\frac{|(\widehat{a} - \widehat{b}) - (a - b)|}{|a - b|} = \frac{|a\delta_a - b\delta_b|}{|a - b|} \leq \frac{|a| + |b|}{|a - b|}\, u. \tag{35.103}$$

The factor $(|a| + |b|)/|a - b|$ is subtraction's condition number, and it blows up whenever $a \approx b$. Subtracting two nearly equal numbers that already carry rounding errors is called **catastrophic cancellation**: the accurate leading digits cancel out, promoting the noise in

the trailing digits into the leading digits of the answer. For reference, the table below collects the condition numbers of several common problems:

problem	κ_f	regime
$a - b$	$(\lvert a\rvert + \lvert b\rvert)/\lvert a - b\rvert$	large when $a \approx b$
$\log(1 + x)$	$x/((1 + x)\log(1 + x))$	1.000 at $x = 10^{-8}$, 0.82 at $x = 0.5$
e^x	$\lvert x\rvert$	large for large $\lvert x\rvert$
$1 - \cos x$	$x \sin x/(1 - \cos x)$	tends to 2 as $x \to 0$
$\sqrt{x}$	$1/2$	always
variance of data with mean m, spread s	at most $2(1 + \lvert m\rvert/s)$	derived below

The second row settles our opening question: $\log(1 + x)$ near $x = 0$ is completely well conditioned ($\kappa_f \approx 1$). The naive algorithm `log(1 + x)` fails because it takes a detour through an ill-conditioned intermediate step. It first forms $y = 1 + x$ and then evaluates $\log y$ at $y \approx 1$, where the logarithm has $\kappa_{\log}(y) = 1/\lvert\log y\rvert \approx 1/x = 10^8$. Rounding $1 + x$ introduces a relative error of order u in y, which log magnifies by 10^8—more than 100% in fp32. By contrast, `log1p` is written so that the rounding error of $1+x$ never enters the result; for tiny x, it simply returns x.

This gives us the organizing principle of the entire section: *an unstable algorithm routes a well-conditioned problem through an ill-conditioned intermediate step.* Whenever κ_f is small and the computed answer is bad, the algorithm is at fault and we should rewrite it. Whenever κ_f itself is large, no algorithm can rescue the answer—we must change the problem instead, which is the subject of Section 35.8.4.

35.8.3.2 Subtraction of Nearly Equal Numbers

Let us test this in code. The cell below repeats the $\log(1+x)$ experiment and then subtracts two fp32 numbers that agree in their first seven decimal digits, 1.0002344 and 1.0002341, whose true difference is 3.0×10^{-7}:

```
x = np.float32(1e-8)
print('float32 rounds 1 + x to:', np.float32(1.0) + x)
print('log(1 + x) =', np.log(np.float32(1.0) + x), '   log1p(x) =', np.log1p(x))
a, b = np.float32(1.0002344), np.float32(1.0002341)
print(f'stored a = {float(a):.10f}, b = {float(b):.10f}')
print('a - b =', a - b, '=', (a - b) / np.spacing(np.float32(1.0)), 'ulps of 1')
print('float64: log(1 + 1e-20) =', np.log(1 + 1e-20),
      '  log1p(1e-20) =', np.log1p(1e-20))
```

```
float32 rounds 1 + x to: 1.0
log(1 + x) = 0.0    log1p(x) = 1e-08
stored a = 1.0002343655, b = 1.0002341270
a - b = 2.3841858e-07 = 2.0 ulps of 1
float64: log(1 + 1e-20) = 0.0   log1p(1e-20) = 1e-20
```

Between 1 and 2, every fp32 difference is an integer multiple of $\varepsilon_{\text{mach}} = 1.19 \times 10^{-7}$, and the true difference 3.0×10^{-7} does not fall on that grid. The subtraction returned two ulps,

2.38×10^{-7} (20% too small), yet once again it committed no error of its own. The inputs were stored as 1.0002343655 and 1.0002341270, and their exact difference is what the cell printed; the error was already baked into the stored inputs. Here the cancellation factor Equation 35.103 is $2.0005/(3 \times 10^{-7}) \approx 7 \times 10^{6}$. Multiplied by $u \approx 6 \times 10^{-8}$, it allows a relative error of up to 0.4, and the observed 0.2 falls squarely inside the bound.

The same cancellation appears in $\log(1 + x)$ and $e^{x} - 1$ near 0 (where `log1p` and `expm1` avoid it), in $1 - \cos x$ near 0 (rewritten as $2\sin^{2}(x/2)$), and in the quadratic formula when $b^{2} \gg 4|ac|$ (Exercise 2). In every one of these cases, an algebraic rewrite removes the offending subtraction.

Finite differences are the one place where we cannot eliminate the subtraction and must settle for a compromise. A numerical gradient check compares automatic differentiation against the forward difference $(f(x+h) - f(x))/h$. How small should we make the step h? By Taylor's theorem, the mathematical truncation error is about $\frac{h}{2}|f''(x)|$, which shrinks as $h \to 0$. Meanwhile, each function evaluation carries a rounding error of roughly $u|f(x)|$, and dividing their difference by h magnifies that error to about $2u|f(x)|/h$, which blows up as $h \to 0$. The total error is minimized where the two terms balance, at

$$h^{\star} = 2\sqrt{\frac{u\,|f(x)|}{|f''(x)|}}, \qquad \text{with error about } 2\sqrt{u\,|f(x)f''(x)|}.$$

Because the optimal error scales as $\sqrt{u}$, the best possible forward difference keeps only about half of the format's digits. For $f = \sin$ at $x = 1$, this predicts $h^{\star} \approx 2 \times 10^{-8}$ (error 1.8×10^{-8}) in fp64 and $h^{\star} \approx 5 \times 10^{-4}$ in fp32. Let us sweep h from 10^{-2} down to 10^{-12} in both formats to see the trade-off directly:

```
print('      h   error in fp64   error in fp32')
for h in 10.0 ** -np.arange(2, 13):
    err = [abs(float((np.sin(t(1) + t(h)) - np.sin(t(1))) / t(h)) - np.cos(1.0))
           for t in (np.float64, np.float32)]  # forward difference of sin at 1
    print(f'{h:7.0e}  {err[0]:14.1e}  {err[1]:14.1e}')
```

```
      h   error in fp64   error in fp32
   1e-02         4.2e-03         4.2e-03
   1e-03         4.2e-04         4.0e-04
   1e-04         4.2e-05         2.8e-04
   1e-05         4.2e-06         3.9e-03
   1e-06         4.2e-07         6.3e-02
   1e-07         4.2e-08         5.6e-02
   1e-08         3.0e-09         5.4e-01
   1e-09         5.3e-08         5.4e-01
   1e-10         5.8e-08         5.4e-01
   1e-11         1.2e-06         5.4e-01
   1e-12         4.3e-05         5.4e-01
```

In fp64, the error drops linearly with h from 4×10^{-3} at $h = 10^{-2}$ (where truncation dominates) down to roughly 10^{-8} near $h = 10^{-8}$ (the exact value at any single h fluctuates with rounding noise), and then climbs back to 4×10^{-5} at $h = 10^{-12}$ as cancellation takes over. In fp32, the sweet spot at $h = 10^{-3}$ to 10^{-4} leaves only about three correct digits; from $h = 10^{-8}$ downward, $1 + h$ rounds straight to 1, making the difference quotient 0

and the error equal to the entire derivative, $\cos 1 = 0.54$. That is why numerical gradient checks should always run in fp64.

Notice, too, that switching to higher precision never cures cancellation—it merely pushes the failure to smaller inputs. The last line of the earlier `log1p` cell showed `log(1 + 1e-20)` returning 0 in fp64 just as fp32 did at 10^{-8}, while `log1p` stayed accurate in both. Reformulating the expression removes the ill-conditioned step itself (Higham 2002).

35.8.3.3 One-Pass Variance

Catastrophic cancellation also hides inside formulas that look completely innocent. Computing a sample variance is the classic example—and it matters directly in deep learning, where normalization layers and data-preprocessing pipelines compute variances continuously.

For data $x_1, \dots, x_N$ with mean m and standard deviation s, the textbook identity

$$\frac{1}{N}\sum_{i=1}^{N}(x_i - m)^2 = \frac{1}{N}\sum_{i=1}^{N} x_i^2 - m^2,$$

tempts us to compute the variance in a single pass by accumulating only the sums of x_i and x_i^2. In exact arithmetic the identity is flawless; in floating point it is a trap. Whenever $s \ll |m|$, both terms on the right sit near m^2 while their difference is s^2, so the cancellation factor Equation 35.103 is roughly $2m^2/s^2$. For $m = 10^9$ and $s = 1$, that factor reaches 2×10^{18}—two orders of magnitude larger than the 10^{16} that fp64 can absorb.

How much of this loss is forced by the problem itself? Let us check the condition number of the variance. Perturb each input x_i by a relative error δ_i with $|\delta_i| \le u$, replacing x_i by $x_i + x_i\delta_i$. The derivative of the variance with respect to x_i is $\frac{2}{N}(x_i - m)$, because the induced change in m multiplies $\sum_i (x_i - m) = 0$. Writing $x_i = (x_i - m) + m$, the first-order change in the variance is therefore

$$\frac{2}{N}\sum_{i=1}^{N}(x_i - m)\, x_i\delta_i = \frac{2}{N}\sum_{i=1}^{N}(x_i - m)^2\delta_i + \frac{2m}{N}\sum_{i=1}^{N}(x_i - m)\,\delta_i.$$

The first sum has magnitude at most $2s^2u$. By the Cauchy–Schwarz inequality, the second has magnitude at most $2|m|u \cdot \frac{1}{N}\sum_i |x_i - m| \le 2|m|s\,u$. Dividing by s^2 shows that the relative condition number of the variance is at most $2(1 + |m|/s) \approx 2|m|/s$, or about 2×10^9 here (choosing $\delta_i = u\,\mathrm{sign}(x_i - m)$ gives $1.6\,|m|/s$ for Gaussian data). The problem itself therefore permits an error of only $2 \times 10^9 u \approx 2 \times 10^{-7}$—about seven accurate digits. The naive one-pass formula, by contrast, suffers an error of order $(m/s)^2 u$: it *squares* the condition number, just as the normal equations will square the condition number of a matrix in Section 35.8.4.

A stable one-pass algorithm keeps all deviations at the scale s rather than m^2. Welford (1962) maintains a running mean m_k and a running sum of *centered* squares $M_k = \sum_{i \le k}(x_i - m_k)^2$, updating both from the deviation $d_k = x_k - m_{k-1}$ of each new sample from the previous mean:

$$m_k = m_{k-1} + \frac{d_k}{k}, \qquad M_k = M_{k-1} + d_k\,(x_k - m_k). \tag{35.104}$$

Notice the asymmetry in the update of M_k: it multiplies the deviation from the *old* mean by the deviation from the *new* mean. That asymmetry is precisely what makes the recursion exact:

Proposition (Welford's recursion). *With $m_0 = M_0 = 0$, the recursions Equation 35.104 satisfy, for every $k \geq 1$ and in exact arithmetic,*

$$m_k = \frac{1}{k}\sum_{i=1}^{k} x_i, \qquad M_k = \sum_{i=1}^{k} (x_i - m_k)^2.$$

Proof. The mean identity $k\,m_k = (k-1)\,m_{k-1} + x_k$ follows immediately from the first recursion. For M_k, we induct on k. Using $m_k - m_{k-1} = d_k/k$ and $x_k - m_k = d_k\,(k-1)/k$, we split off the k-th term and recenter the first $k-1$ terms around m_{k-1}:

$$\begin{aligned}\sum_{i=1}^{k}(x_i - m_k)^2 &= \sum_{i=1}^{k-1}\left((x_i - m_{k-1}) - \tfrac{d_k}{k}\right)^2 + (x_k - m_k)^2 \\ &= M_{k-1} + (k-1)\tfrac{d_k^2}{k^2} + \tfrac{(k-1)^2}{k^2} d_k^2.\end{aligned}$$

The cross term vanishes because $\sum_{i\leq k-1}(x_i - m_{k-1}) = 0$, and the inductive hypothesis identifies the first sum as M_{k-1}. Combining the remaining two terms gives $\frac{k-1}{k}\,d_k^2 = d_k \cdot d_k\frac{k-1}{k} = d_k(x_k - m_k)$, which is exactly what the recursion adds. ■

Proving that Welford's recursion also stays accurate in floating point is the harder step, and we sketch only the intuition here (see Higham (2002) for the full analysis). At first glance, Welford's update still subtracts large numbers, since x_k and m_{k-1} both sit near 10^9. By Sterbenz's lemma, however, that subtraction is exact, and x_k is exact input data, so the only error in d_k comes from the running mean—at most a few dozen ulps at $|m|$ over 10^5 steps. Every quantity added into M_k is therefore a deviation of scale s, so rounding errors in the updates enter relative to s^2 rather than m^2. Even a constant offset ε in the mean shifts a centered sum of squares only by $N\varepsilon^2$, because the first-order term multiplies $\sum_i (x_i - m) = 0$. In practice, Welford's error matches the problem's own sensitivity, $u|m|/s \approx 1.1 \times 10^{-7}$, whereas the naive one-pass formula suffers an error of $(m/s)^2 u \approx 100$.

Let us compare all three methods—naive one-pass, Welford, and a two-pass reference—on 10^5 fp64 samples with mean 10^9 and variance 1:

```
rng = np.random.default_rng()
x = 1e9 + rng.normal(0.0, 1.0, size=100_000)    # mean 1e9, variance 1

naive = (x**2).mean() - x.mean()**2             # one pass, cancels
two_pass = ((x - x.mean())**2).mean()           # subtract the mean first

m, M = 0.0, 0.0                                 # Welford: one pass
for k, xk in enumerate(x, start=1):
```

```
    d = xk - m
    m += d / k                                  # m_k
    M += d * (xk - m)                           # M_k
welford = M / len(x)

print(f'naive E[x^2] - E[x]^2 : {naive:12.6f}   (= {naive / 128:.0f} x 128)')
print(f'Welford, one pass     : {welford:12.6f}')
print(f'two-pass reference    : {two_pass:12.6f}')
print(f'Welford vs two-pass   : {abs(welford - two_pass) / two_pass:.1e} relative')
print('float64 gap near 1e18 :', np.spacing(1e18))
```

```
naive E[x^2] - E[x]^2 :  -128.000000   (= -1 x 128)
Welford, one pass     :     0.999287
two-pass reference    :     0.999287
Welford vs two-pass   : 6.8e-09 relative
float64 gap near 1e18 : 128.0
```

The naive one-pass estimate comes out as an exact multiple of 128 (typically 0, ± 128, or ± 256), nowhere near the true variance 1, and frequently negative. Why 128? Both $\frac{1}{N}\sum x_i^2$ and m^2 lie in $[2^{59}, 2^{60})$, where adjacent fp64 numbers are $2^{59-52} = 128$ apart, so their difference can only be a multiple of 128: our predicted error of about 100, quantized to the grid. Welford's recursion, on the other hand, matches the two-pass reference to at least seven digits—and often eight or nine, because rounding errors of opposite signs partly cancel across the sum.

The same caution applies whenever a variance is accumulated in one pass, such as computing streaming statistics over a dataset. (The running moments that batch normalization maintains for inference are exponential moving averages, Section 7.3, which do not subtract large running sums.)

35.8.3.4 Summation Order

Both the naive variance and the two-pass reference sum 10^5 numbers, and in floating point the order of addition matters. Each addition introduces a factor $(1+\delta_k)$, so in a plain left-to-right sum the first term passes through all $n-1$ roundings. To first order in u, a left-to-right sum satisfies

$$\Big|\mathrm{fl}\Big(\sum_{i=1}^{n} x_i\Big) - \sum_{i=1}^{n} x_i\Big| \le (n-1)\,u \sum_{i=1}^{n} |x_i|.$$

Because early terms accumulate the rounding error of every subsequent addition, the worst-case error grows linearly with n. *Pairwise* summation avoids this by recursively summing the two halves of the array, so every term passes through only $\log_2 n$ additions and the bound shrinks to $u \log_2 n \sum_i |x_i|$; NumPy's `sum` uses a blocked version of this idea. Kahan's compensated summation goes one step further, carrying each addition's rounding error in a running correction variable to achieve an error bound independent of n (Exercise 8). Let us add 10^7 copies of fp32's 0.1 (whose stored value is $0.100000001490\ldots$) both left to right and pairwise, padding the pairwise array with zeros to 2^{24} entries and repeatedly adding adjacent pairs:

```
x = np.full(10_000_000, 0.1, dtype=np.float32)
exact = len(x) * float(x[0])                 # x[0] = 0.100000001490...
s = np.concatenate([x, np.zeros(2**24 - len(x), np.float32)])  # pad to 2^24
while len(s) > 1:                            # pairwise: add neighbors, halve
    s = s[0::2] + s[1::2]
print(f'in order (cumsum): {np.cumsum(x)[-1]:16.4f}')
print(f'pairwise         : {s[0]:16.4f}')
print(f'exact            : {exact:16.4f}')
```

```
in order (cumsum):     1087937.0000
pairwise         :     1000000.0000
exact            :     1000000.0149
```

The left-to-right sum finishes at $1{,}087{,}937$—almost 9% too high, though still inside its bound $(n-1)u\sum_i |x_i| \approx 6\times 10^5$. Pairwise summation lands at $1{,}000{,}000$, within 0.015 of the exact sum and well inside its bound $u\log_2 n\sum_i |x_i| \approx 1.4$.

The culprit behind the left-to-right drift is the same grid effect that erased small weight updates: each addend is rounded to the spacing of the current running sum. Once the sum crosses $2^{19} = 524{,}288$, fp32 numbers sit $1/16$ apart, so 0.1 equals 1.6 gaps and every addition rounds up to two gaps, 0.125—25% too large. Above 2^{20} the spacing doubles to $1/8$, and each addend again rounds to 0.125; earlier, between 2^{17} and 2^{19}, it rounded down to 0.09375. Because floating-point addition is not associative, a parallel reduction across GPU threads need not produce the exact same bits from one device or run to another. Fortunately, Welford's recursion also supports pairwise reduction: the merge rule in Exercise 7 combines statistics from two partial sums, which is how running moments are merged across devices in distributed training.

35.8.4 Conditioning

The condition number of a matrix and the filter factors of ridge regression are developed in Section 33.5. New here are forward and backward error, a formula that measures the backward error, and the price in digits of the normal equations.

35.8.4.1 Two Nearly Parallel Lines

So far the answer of a computation was a number. For a linear system it is a vector, and Section 33.5.3.2 showed on the system

$$x_1 + x_2 = 2, \qquad x_1 + 1.0001\,x_2 = 2.0001,$$

how fragile such an answer can be. The solution is $(1, 1)$, and raising the second right-hand side by 10^{-4}, a relative change of 3.5×10^{-5} in the data, moves it to $(0, 2)$. Each equation is a line in the plane, and the solution is where the lines cross. These two lines have slopes -1 and -0.9999 and meet at an angle of 0.003°, so shifting one of them slightly slides the crossing far along the other (Figure 35.32).

Let us read the same numbers in the other direction. Suppose a solver returned $\hat{\mathbf{x}} = (0, 2)$ for the original system. Its **forward error**, the distance to the true solution, is 100%. Yet $(0, 2)$ is the exact solution of a nearby problem, the one with b_2 changed by 10^{-4}. The

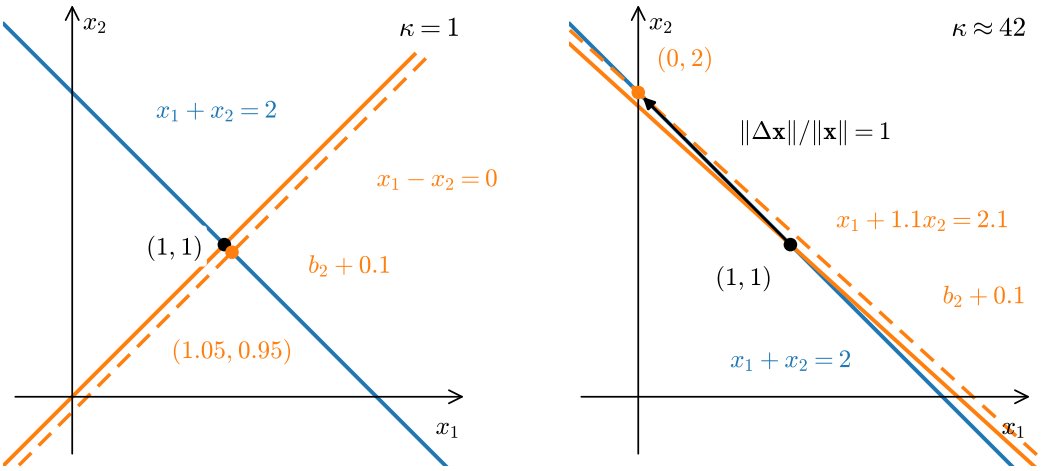


Figure 35.32: A linear system is sensitive when its lines are nearly parallel. (a) Two perpendicular lines meet at $(1, 1)$; shifting the second by 0.1 moves the crossing by 5%: amplification 1, $\kappa = 1$. (b) The lines $x_1 + x_2 = 2$ and $x_1 + 1.1x_2 = 2.1$ also meet at $(1, 1)$, but the same shift moves the crossing to $(0, 2)$, a 100% change from a 3.4% change in $\mathbf{b}$: amplification 29, bounded by $\kappa \approx 42$. The text's system, with 1.0001 in place of 1.1, has the same geometry at an angle of 0.003° and $\kappa \approx 4 \times 10^4$.

smallest such change of the data is the **backward error**. A solver with a tiny backward error has solved a nearby problem exactly, and it can still be far from the answer.

How far is a property of the problem, not of the solver. It is the condition number of Section 35.8.3.1 once more, now for the function $\mathbf{b} \mapsto \mathbf{A}^{-1}\mathbf{b}$: the largest factor by which a relative change of the input can become a relative change of the output, taken over all right-hand sides and all directions of the change. For a matrix with singular values $\sigma_1 \geq \cdots \geq \sigma_d > 0$ that factor is $\kappa(\mathbf{A}) = \sigma_1/\sigma_d$ (Section 33.5.3.2), and it is about 4×10^4 for the two lines. The shift $(0, 10^{-4})$ realizes 2.8×10^4 of it, because it is not along the most amplified direction.

35.8.4.2 Backward and Forward Error

For a linear system $\mathbf{Ax} = \mathbf{b}$ and a computed solution $\hat{\mathbf{x}}$, the forward error is $\|\hat{\mathbf{x}} - \mathbf{x}\|/\|\mathbf{x}\|$ and the backward error is the size of the smallest change of the data that makes $\hat{\mathbf{x}}$ exact. An algorithm is **backward stable** if its backward error is of the order of u, so that it returns the exact answer of a problem that differs from the given one by about one rounding of the data. The SVD and the QR decomposition are backward stable, and so in practice is the LU decomposition with pivoting that `np.linalg.solve` uses. The one-pass variance is not: the variance of any data set is nonnegative, so a negative output is the exact variance of no data at all.

The definition asks for a minimization over all changes of the data. Luckily, for changes of the matrix it has a closed form, computable from the residual.

Proposition (backward error from the residual). *Let* $\hat{\mathbf{x}} \neq \mathbf{0}$ *and* $\mathbf{r} = \mathbf{b} - \mathbf{A}\hat{\mathbf{x}}$. *The smallest* $\|\Delta\mathbf{A}\|_2$ *with* $(\mathbf{A} + \Delta\mathbf{A})\hat{\mathbf{x}} = \mathbf{b}$ *is* $\|\mathbf{r}\|/\|\hat{\mathbf{x}}\|$, *attained by* $\Delta\mathbf{A} = \mathbf{r}\hat{\mathbf{x}}^\top/\|\hat{\mathbf{x}}\|^2$.

Proof. Any such $\Delta\mathbf{A}$ satisfies $\Delta\mathbf{A}\hat{\mathbf{x}} = \mathbf{b} - \mathbf{A}\hat{\mathbf{x}} = \mathbf{r}$, so $\|\mathbf{r}\| \leq \|\Delta\mathbf{A}\|\,\|\hat{\mathbf{x}}\|$. The rank-one matrix

$\mathbf{r}\hat{\mathbf{x}}^{\top}/\|\hat{\mathbf{x}}\|^2$ maps $\hat{\mathbf{x}}$ to $\mathbf{r}$, and its norm is $\|\mathbf{r}\|\,\|\hat{\mathbf{x}}\|/\|\hat{\mathbf{x}}\|^2 = \|\mathbf{r}\|/\|\hat{\mathbf{x}}\|$. ■

Relative to the size of $\mathbf{A}$, the backward error is

$$\omega = \frac{\|\mathbf{b} - \mathbf{A}\hat{\mathbf{x}}\|}{\|\mathbf{A}\|\,\|\hat{\mathbf{x}}\|}. \tag{35.105}$$

A small residual certifies a small backward error, whatever the condition of the problem. For $\hat{\mathbf{x}} = (0, 2)$ in the two-line system, $\mathbf{r} = (0, -10^{-4})$ and $\omega = 10^{-4}/(2.00005 \cdot 2) = 2.5 \times 10^{-5}$. The version that perturbs $\mathbf{b}$ as well is due to Rigal and Gaches (see Higham (2002)). In floating point the residual is itself computed with rounding errors, so printed values of ω below about u are noise.

The condition number converts backward error into forward error. This is the result to remember, and its proof needs only the two norms $\|\mathbf{A}\| = \sigma_1$ and $\|\mathbf{A}^{-1}\| = 1/\sigma_d$ (Section 33.5).

Proposition (forward error $\le$ condition number $\times$ backward error). *Let $\mathbf{A}$ be invertible, let $\mathbf{A}\mathbf{x} = \mathbf{b}$, and suppose the computed $\hat{\mathbf{x}}$ exactly solves a nearby system, $(\mathbf{A} + \Delta\mathbf{A})\,\hat{\mathbf{x}} = \mathbf{b}$ with $\|\Delta\mathbf{A}\| \le \omega\|\mathbf{A}\|$. Then*

$$\frac{\|\hat{\mathbf{x}} - \mathbf{x}\|}{\|\hat{\mathbf{x}}\|} \;\le\; \kappa(\mathbf{A})\,\omega. \tag{35.106}$$

Proof. Subtracting $\mathbf{A}\mathbf{x} = \mathbf{b}$ from $(\mathbf{A} + \Delta\mathbf{A})\hat{\mathbf{x}} = \mathbf{b}$ gives $\mathbf{A}(\hat{\mathbf{x}} - \mathbf{x}) = -\Delta\mathbf{A}\,\hat{\mathbf{x}}$, hence $\hat{\mathbf{x}} - \mathbf{x} = -\mathbf{A}^{-1}\Delta\mathbf{A}\,\hat{\mathbf{x}}$ and

$$\|\hat{\mathbf{x}} - \mathbf{x}\| \le \|\mathbf{A}^{-1}\|\,\|\Delta\mathbf{A}\|\,\|\hat{\mathbf{x}}\| \le \frac{1}{\sigma_d}\,\sigma_1\,\omega\,\|\hat{\mathbf{x}}\| = \kappa(\mathbf{A})\,\omega\,\|\hat{\mathbf{x}}\|. \qquad \blacksquare$$

The error here is measured relative to $\hat{\mathbf{x}}$, the vector we have. Relative to the true $\mathbf{x}$, substituting $\|\hat{\mathbf{x}}\| \le \|\mathbf{x}\| + \|\hat{\mathbf{x}} - \mathbf{x}\|$ and solving for $\|\hat{\mathbf{x}} - \mathbf{x}\|$ gives the bound $\kappa\omega/(1 - \kappa\omega)$ when $\kappa\omega < 1$, the same to first order. On the two-line system, $\kappa\omega = 4 \times 10^4 \cdot 2.5 \times 10^{-5} = 1.0$, and the error relative to $\hat{\mathbf{x}}$ is $\|(-1, 1)\|/\|(0, 2)\| = 0.71$, inside the bound. For an error in the right-hand side instead of the matrix, the corresponding bound appears in Section 33.5.3.2.

A backward-stable solver has $\omega \approx u$, and taking $\log_{10}$ of the bound gives the rule of thumb

$$\text{correct digits in } \hat{\mathbf{x}} \;\approx\; \text{digits carried by the format} \;-\; \log_{10}\kappa(\mathbf{A}).$$

The rule is a worst case. A matrix with $\kappa = 10^3$ can lose three of fp32's seven digits. The chapter's running example $\frac{1}{2}(x_1^2 + 1000x_2^2)$ has that κ and loses none: its matrix is diagonal, and each unknown takes one correctly rounded division.

35.8.4.3 Example: The Hilbert Matrix

The **Hilbert matrix** $\mathbf{H}_{ij} = 1/(i + j - 1)$, $i, j = 1, \dots, n$, has a condition number that grows exponentially with n: its rows become nearly parallel in the same way as the two lines above. We solve $\mathbf{H}\mathbf{x} = \mathbf{b}$ with $\mathbf{b} = \mathbf{H}\mathbf{1}$, whose exact solution is $\mathbf{1}$, and compare the forward error with the rule of thumb and with the backward error Equation 35.105. One caveat before we judge the solver. Computing $\mathbf{b} = \mathbf{H}\mathbf{1}$ in fp64 rounds every entry of $\mathbf{b}$, a change of the data of

relative size u before the solver starts. (Rounding the entries $1/3, 1/7, \ldots$ does no harm here, because **b** is built from the matrix as stored.) How much does that alone cost? Every fp64 number is a rational number, so the stored system can also be solved without any rounding, by Gauss–Jordan elimination on `Fraction` entries. The table reports the error of that exact solve beside the error of `np.linalg.solve`.

```
def solve_exact(A, b):
    """Gauss-Jordan elimination in exact rational arithmetic."""
    M = [[Fraction(a) for a in row] + [Fraction(c)] for row, c in zip(A, b)]
    n = len(M)
    for j in range(n):              # no pivoting: A is positive definite here
        for i in range(n):
            if i != j:
                r = M[i][j] / M[j][j]
                M[i] = [a - r * c for a, c in zip(M[i], M[j])]
    return np.array([float(M[i][n] / M[i][i]) for i in range(n)])
```

For dimensions $n \in \{4, 6, 8, 10, 12\}$, we now construct the $n \times n$ Hilbert matrix `H` and right-hand side `b = H @ x_true`, comparing the floating-point solver `np.linalg.solve(H, b)` against `solve_exact(H, b)` across condition number, forward error, digits retained, and backward error:

```
print(' n     kappa  forward error  digits kept  16 - log10 kappa  backward error'
      '  exact solve error')
for n in [4, 6, 8, 10, 12]:
    i = np.arange(n)
    H = 1.0 / (1.0 + i[:, None] + i[None, :])   # Hilbert matrix
    x_true = np.ones(n)
    b = H @ x_true
    x_hat = np.linalg.solve(H, b)
    kappa = np.linalg.cond(H)
    fwd = np.linalg.norm(x_hat - x_true) / np.linalg.norm(x_true)
    omega = (np.linalg.norm(b - H @ x_hat)
             / (np.linalg.norm(H, 2) * np.linalg.norm(x_hat)))
    exact = np.linalg.norm(solve_exact(H, b) - x_true) / np.linalg.norm(x_true)
    print(f'{n:2d}  {kappa:8.1e}  {fwd:13.1e}  {-np.log10(fwd):11.1f}'
          f'  {16 - np.log10(kappa):16.1f}  {omega:14.1e}  {exact:17.1e}')
```

n	kappa	forward error	digits kept	16 - log10 kappa	backward error	exact solve error
4	1.6e+04	4.1e-14	13.4	11.8	0.0e+00	1.1e-13
6	1.5e+07	1.4e-10	9.8	8.8	1.3e-16	1.0e-10
8	1.5e+10	6.1e-08	7.2	5.8	5.7e-17	7.0e-08
10	1.6e+13	8.7e-05	4.1	2.8	1.0e-16	4.5e-05
12	1.8e+16	3.2e-01	0.5	-0.2	9.8e-17	1.4e-01

The digits kept fall from about 13 at $n = 4$ to well under one at $n = 12$, while the backward error stays at or below about 10^{-16}: each computed vector solves a system that differs from the stored one in the sixteenth or seventeenth digit. Is the solver to blame for the lost digits, then? The last column acquits it. The exact solve of the stored system, which commits no rounding error at all, misses **1** by an error of the same order in every row, about 10^{-7}, 10^{-4} and 10^{-1} for $n = 8$, 10 and 12. The digits were lost when **b** was stored, before any solver ran. The rule of thumb is pessimistic by about one digit (from about half a digit to about one

and a half, depending on the row), because the error is not aligned with the most amplified direction. At $n = 12$, $\kappa u \approx 2$ and the bound no longer constrains anything. The last decimals vary with the LAPACK build; the trend does not.

35.8.4.4 Conditioning of the Normal Equations

The least-squares problem $\min_{\mathbf{w}} \|\mathbf{A}\mathbf{w} - \mathbf{b}\|^2$ with $\mathbf{A} \in \mathbb{R}^{n \times d}$ of full column rank is solved by the normal equations $\mathbf{A}^\top \mathbf{A}\mathbf{w} = \mathbf{A}^\top \mathbf{b}$. This is an algebraically equivalent route through a worse-conditioned matrix (Section 33.5.3.2):

$$\kappa(\mathbf{A}^\top \mathbf{A}) = \kappa(\mathbf{A})^2. \tag{35.107}$$

We have seen this before, in the one-pass variance: the convenient formula squares the condition number of the problem. By the rule of thumb, the normal equations can lose about $2 \log_{10} \kappa$ digits. A solve by QR ($\mathbf{A} = \mathbf{Q}\mathbf{R}$ with orthonormal $\mathbf{Q}$ and triangular $\mathbf{R}$) or by the SVD works on $\mathbf{A}$ directly and loses about $\log_{10} \kappa$, *when the residual* $\mathbf{b} - \mathbf{A}\mathbf{w}$ *is small*. With a large residual the least-squares problem itself is more sensitive: its condition number is about $\kappa + \kappa^2 \tan\theta$, where θ is the angle between $\mathbf{b}$ and the column space of $\mathbf{A}$. QR and SVD then lose close to $2 \log_{10} \kappa$ digits too, but never more than the problem forces; the normal equations lose that many whatever the residual.

The chapter's running example is too well conditioned to show the difference: its design has $\mathbf{A}^\top \mathbf{A} = \mathrm{diag}(1, 1000)$, so $\kappa(\mathbf{A}) = \sqrt{1000} \approx 32$. The cell therefore builds harder designs, with $\kappa(\mathbf{A}) = 10^3$ and 10^6, from random orthonormal factors and a right-hand side in the column space (zero residual). We expect gaps of three and of six digits:

```
rng = np.random.default_rng()
n, d = 100, 10
for kappa in [1e3, 1e6]:
    U, _ = np.linalg.qr(rng.normal(size=(n, d)))    # orthonormal columns
    V, _ = np.linalg.qr(rng.normal(size=(d, d)))
    sigma = np.logspace(0, -np.log10(kappa), d)     # kappa(A) as chosen
    A = U * sigma @ V.T                             # A = U diag(sigma) V^T
    w_true = rng.normal(size=d)
    b = A @ w_true                                  # zero residual
    w_ne = np.linalg.solve(A.T @ A, A.T @ b)        # normal equations
    w_ls = np.linalg.lstsq(A, b, rcond=None)[0]     # SVD-based solve
    digits = [-np.log10(np.linalg.norm(w - w_true) / np.linalg.norm(w_true))
              for w in (w_ne, w_ls)]
    print(f'kappa(A) = {kappa:.0e}: normal equations keep {digits[0]:4.1f} '
          f'digits, lstsq keeps {digits[1]:4.1f}')
```

```
kappa(A) = 1e+03: normal equations keep 10.3 digits, lstsq keeps 14.3
kappa(A) = 1e+06: normal equations keep  5.7 digits, lstsq keeps 11.7
```

At $\kappa(\mathbf{A}) = 10^3$ the normal equations keep about 10 to 12 digits against about 14 for `lstsq`, and at 10^6 about 4 to 6 against about 11 to 13. The exact digits vary from run to run by a digit or two; the gaps scatter around $\log_{10} \kappa$, mostly within a digit of it. This is why `lstsq` and the pseudoinverse of Section 33.5.3.1 work from the SVD and not from $(\mathbf{A}^\top \mathbf{A})^{-1} \mathbf{A}^\top$.

35.8.4.5 Ridge Regularization and Conditioning

When $\kappa(\mathbf{A})$ itself is large, because features are nearly collinear or the design is rank deficient, no choice of algorithm helps: the problem must change. One change keeps the answer. A **preconditioner** changes variables so that the new problem is better conditioned and its solution maps back to the original one. Dividing each column of $\mathbf{A}$ by its norm is an example (Exercise 4), and the per-coordinate step sizes of Section 35.5 do the same inside an optimizer.

Ridge regularization changes the answer as well. It minimizes $\frac{1}{2}\|\mathbf{A}\mathbf{w}-\mathbf{b}\|^2+\frac{\lambda}{2}\|\mathbf{w}\|^2$, whose minimizer is

$$\mathbf{w}_\lambda = (\mathbf{A}^\top\mathbf{A}+\lambda\mathbf{I})^{-1}\mathbf{A}^\top\mathbf{b}, \tag{35.108}$$

the ridge solution of Section 33.5.3.1. Let $\sigma_1 \geq \cdots \geq \sigma_d \geq 0$ be the singular values of $\mathbf{A}\in\mathbb{R}^{n\times d}$, with $\sigma_i = 0$ beyond its rank. The matrix $\mathbf{A}^\top\mathbf{A}+\lambda\mathbf{I}$ has the eigenvectors of $\mathbf{A}^\top\mathbf{A}$ and the eigenvalues $\sigma_i^2+\lambda \geq \lambda > 0$. It is therefore positive definite, hence invertible even when $\mathbf{A}$ is rank deficient, and

$$\kappa(\mathbf{A}^\top\mathbf{A}+\lambda\mathbf{I}) = \frac{\sigma_1^2+\lambda}{\sigma_d^2+\lambda}. \tag{35.109}$$

Whenever $\sigma_1 > \sigma_d$ this ratio decreases strictly in λ (its derivative is $(\sigma_d^2-\sigma_1^2)/(\sigma_d^2+\lambda)^2 < 0$) and tends to 1 as $\lambda \to \infty$. Adding λ lifts the floor of the spectrum. While $\lambda \ll \sigma_1^2$ it barely moves the ceiling, so κ falls about in proportion to $1/\lambda$ once $\lambda \gg \sigma_d^2$. The damping term of second-order optimizers and the jitter added to a kernel matrix in Gaussian process regression (Chapter 40) are this $\lambda\mathbf{I}$ under other names.

The price is a different minimizer. In the present notation, with $\mathbf{v}_i$ the right singular vectors of $\mathbf{A}$ and $\mathbf{w}_0 = \mathbf{A}^+\mathbf{b}$ the minimum-norm least-squares solution, the filter factors of Section 33.5.3.1 read

$$\mathbf{w}_\lambda = \sum_{i=1}^{d} \frac{\sigma_i^2}{\sigma_i^2+\lambda}\,(\mathbf{v}_i^\top\mathbf{w}_0)\,\mathbf{v}_i. \tag{35.110}$$

Ridge damps most the directions with small σ_i, and these are the very directions in which the unregularized solve amplifies data errors by $1/\sigma_i$. Figure 35.33 shows the effect on the level sets; each panel is centered on its own minimizer, so the shift of the minimizer toward the origin is not drawn.

The same κ counts the steps of gradient descent on the ridge objective. At the best fixed step, gradient descent on a quadratic with condition number $\kappa > 1$ contracts the error by $\rho = (\kappa-1)/(\kappa+1)$ per step (Section 35.1). Started at the origin, where the error is $-\mathbf{w}^\star$, it has relative error $\|\mathbf{w}_k-\mathbf{w}^\star\|/\|\mathbf{w}^\star\| \leq \rho^k$ after k steps, so a tolerance $0 < \mathrm{tol} < 1$ is met within

$$k \leq \left\lceil \frac{\log(1/\mathrm{tol})}{\log\frac{\kappa+1}{\kappa-1}} \right\rceil \leq \left\lceil \frac{\kappa}{2}\log\frac{1}{\mathrm{tol}} \right\rceil \quad \text{steps.} \tag{35.111}$$

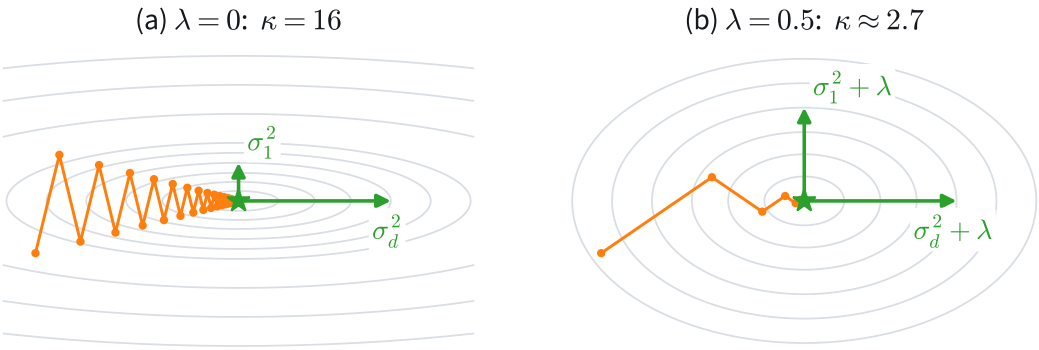


Figure 35.33: Level sets of the least-squares objective before and after adding the ridge term $\frac{\lambda}{2}\|\mathbf{w}\|^2$, drawn for $\kappa = 16$ so that both panels stay legible (the running example's $\kappa = 1000$ would draw as a line). The penalty lifts every eigenvalue of $\mathbf{A}^\top\mathbf{A}$ by λ; the semi-axes, proportional to $1/\sqrt{\sigma_i^2 + \lambda}$, are long where the curvature is small, and the valley rounds into a bowl. Orange: gradient-descent iterates from the same start at each panel's best fixed step; green arrows: the principal axes, labeled by their curvatures.

The second inequality holds because $\log\frac{\kappa+1}{\kappa-1} = 2\,\mathrm{artanh}(1/\kappa) \geq 2/\kappa$. For six digits it allows $\frac{1}{2}\log 10^6 = 6.9$ steps per unit of κ; a run in more than two dimensions can finish sooner (Exercise 4).

How many steps does a given λ save, and what does it cost? We measure both on the running example. Its objective $\frac{1}{2}(w_1^2 + 1000w_2^2) - w_1 - 1000w_2$ is least squares with $\mathbf{A}^\top\mathbf{A} = \mathrm{diag}(1, 1000)$ and $\mathbf{A}^\top\mathbf{b} = (1, 1000)$. Its minimizer is $(1, 1)$, so the ridge minimizer displays the two filter factors directly. The function `gd_iterations` runs gradient descent at the best fixed step and returns the step count and the ridge minimizer:

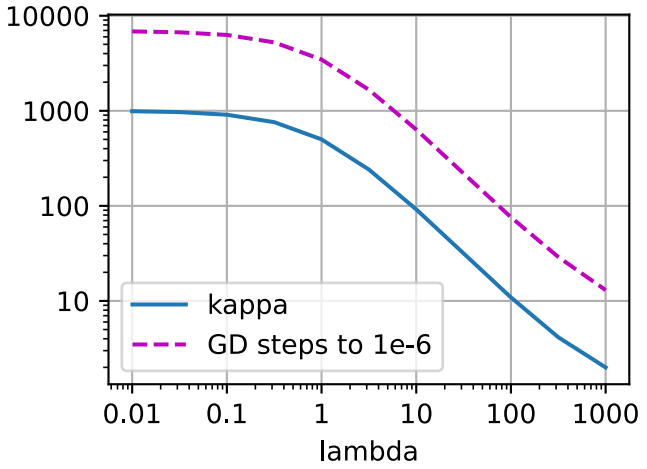
```
def gd_iterations(AtA, Atb, lam, tol=1e-6):
    """GD steps at step 2/(L+mu) to reach relative error tol on ridge."""
    M = AtA + lam * np.eye(len(Atb))
    w_star = np.linalg.solve(M, Atb)
    mu, L = np.linalg.eigvalsh(M)[[0, -1]]
    eta, w, k = 2 / (L + mu), np.zeros(len(Atb)), 0
    while np.linalg.norm(w - w_star) > tol * np.linalg.norm(w_star):
        w, k = w - eta * (M @ w - Atb), k + 1
    return k, w_star

AtA, Atb = np.diag([1.0, 1000.0]), np.array([1.0, 1000.0])
print('lambda   kappa  steps  steps/kappa  ridge minimizer')
for lam in [0, 1, 10, 100, 1000]:
    kappa = (1000 + lam) / (1 + lam)
    k, w_star = gd_iterations(AtA, Atb, lam)
    print(f'{lam:6d} {kappa:7.1f} {k:6d} {k / kappa:12.2f}  {w_star.round(4)}')
```

```
lambda   kappa  steps  steps/kappa  ridge minimizer
     0  1000.0   6908         6.91  [1. 1.]
     1   500.5   3458         6.91  [0.5   0.999]
```

```
   10    91.8    635         6.92  [0.0909 0.9901]
  100    10.9     76         6.98  [0.0099 0.9091]
 1000     2.0     13         6.51  [0.001 0.5  ]
```

The condition number falls from 1000 to 2 as λ goes from 0 to 1000, and the step count falls with it, from $6{,}908$ to 13. The ratio of steps to κ stays near 6.9 (at small κ the approximation $2/\kappa$ is loose, and rounding up to a whole step matters). The last column shows what the saved steps cost. At $\lambda = 1$ the slow direction, with $\sigma^2 = 1$, is halved to 0.5, while the stiff direction, with $\sigma^2 = 1000$, moves by 0.1%. At $\lambda = 1000$ the stiff direction is halved too. A sweep over λ shows κ and the step count together:

```
lams = np.logspace(-2, 3, 11)
kappas = (1000 + lams) / (1 + lams)
steps = [gd_iterations(AtA, Atb, lam)[0] for lam in lams]
d2l.plot(lams, [kappas, steps], 'lambda', None,
         legend=['kappa', 'GD steps to 1e-6'], xscale='log', yscale='log')
```



On log–log axes the two curves are parallel, a factor of 6.9 apart, with slope close to -1 while λ lies well between the eigenvalues 1 and 1000. Ridge thus conditions the linear solve and gradient descent alike, and it pays for both by shrinking the solution. Section 35.7.3.2 shows that the penalty $\frac{\lambda}{2}\|\mathbf{w}\|^2$ corresponds to a constraint on $\|\mathbf{w}\|$, so a λ chosen for statistical reasons also sets the conditioning of the computation.

None of these tools covers error that grows over many steps of an iteration, such as a gradient through a hundred layers; Section 4.4 takes that up.

35.8.5 Summary

A floating-point format stores numbers with about p bits of relative precision and a finite exponent range. Every rounding introduces a relative error of at most $u = 2^{-p-1}$, while overflow, underflow, and a lost small update are absolute effects that the relative model does not show. Softmax and cross-entropy fail because they pass through values the format cannot hold, and the one-pass variance because it subtracts nearly equal large numbers; none of their answers is extreme. The max shift, cross-entropy computed from logits, and

Welford's recursion avoid the detour. The relative condition number $\kappa_f = |xf'/f|$ separates a bad algorithm from a hard problem: when it is small and the computed answer is bad, the algorithm is to blame. For a linear system, $\kappa(\mathbf{A})$ bounds the forward error by κ times the backward error, which a residual certifies, so about $\log_{10}\kappa$ digits are lost, and twice that through the normal equations. Ridge adds λ to every squared singular value, which lowers κ for the solve and for gradient descent alike, at the price of shrinking the solution by the filter factors $\sigma_i^2/(\sigma_i^2+\lambda)$.

35.8.6 Exercises

1. **Loss-scale window.** Loss scaling multiplies the loss by a factor S, which multiplies every gradient by S, and divides the gradients by S after they have been converted to fp32. In fp16 the smallest normal number is $2^{-14} \approx 6.1 \times 10^{-5}$, the smallest subnormal number is $2^{-24} \approx 6.0 \times 10^{-8}$ (values below 2^{-25} round to zero), and the largest finite number is 65504. Suppose that the exact nonzero gradient entries of a network have magnitudes between 10^{-9} and 10^{-1}, and that the backward pass computes the scaled gradients in fp16.
   1. Find all powers of two S for which every scaled gradient is a normal fp16 number.
   2. The gradients of a deeper network have magnitudes between 10^{-12} and 10^{-1}. Show that no scale keeps all of them normal. Take the largest power of two S that avoids overflow. How many significant bits does a scaled gradient of magnitude 10^{-12} keep, and what is its relative rounding error?
   3. Dynamic loss scaling (Section 5.5.3.3) picks the largest scale that avoids overflow and thereby sacrifices the smallest gradients. Why is that the better end to sacrifice?
   4. How do the answers to the first two sub-items change when the gradients are computed in bf16?

2. code **Blame the problem or the algorithm.** Each expression below is evaluated in fp32. For each, compute the relative condition number Equation 35.102 of the *function* in the stated regime and the cancellation factor Equation 35.103 of the *formula as written*. Decide whether the problem or the algorithm is to blame, and rewrite the formula so that its error matches the problem's. Check both forms against fp64.
   1. The root of smaller magnitude of $ax^2+bx+c=0$ for $b>0$ and $b^2 \gg 4|ac|$, as a function of c, for example $a=c=1$ and $b=10^4$. (Hint: multiply by the conjugate.)
   2. The function $\log(1-e^{-x})$ for $x>0$, which computes $\log(1-q)$ from $\log q=-x$, at $x \in \{10^{-6}, 10^{-3}, 1, 20\}$. Find one formulation that is accurate for small x and one that is accurate for large x, and choose a switching point.
   3. The binary cross-entropy of a logit t with label 1, $\log(1+e^{-t})$, at $t \in \{-800, -50, 0, 50\}$. At which of these does the formula as written overflow, and at which does it return 0? Show that $\max(-t,0)+\operatorname{log1p}(e^{-|t|})$ equals it, never overflows, and loses no digits. Which part of the log-sum-exp sandwich is it?
   4. $\sqrt{x+1}-\sqrt{x}$ at $x=10^8$.

   Part 2 is adapted from Martin Mächler, Accurately Computing log(1 − exp(−|a|)), Section 2.

3. code **Normal equations in floating point.** The normal equations square the condition number Equation 35.107.
   1. For $\tau>0$, the $(m+1)\times m$ matrix $\mathbf{A}$ of Läuchli (1961) has a first row of ones

and $\tau\mathbf{I}$ below it. Compute $\mathbf{A}^\top\mathbf{A}$, the singular values of $\mathbf{A}$, and $\kappa(\mathbf{A})$. Show that for $\tau = 10^{-8}$ the matrix $\mathbf{A}^\top\mathbf{A}$ rounds to a singular matrix in fp64. How many correct digits does the rule of thumb of Section 35.8.4 predict for a QR-based least-squares solve with this $\mathbf{A}$ when $\mathbf{b}$ lies in the range of $\mathbf{A}$? What changes in the prediction when the residual is large?

2. Modify the `#numerical-stability-conditioning-normal-equations` cell to sweep $\kappa(\mathbf{A}) = 10^k$ for $k = 2, \dots, 10$. Plot the correct digits of the normal equations and of `lstsq` against k, together with the two rule-of-thumb lines $16 - 2k$ and $16 - k$. At which k does the rule predict that the normal equations keep no correct digit, and what do you observe there?

4. code **Feature scaling as preconditioning.** Features measured in different units give the columns of a design matrix very different norms. Let $\mathbf{X}$ be a 200×5 matrix with standard normal entries, let $\mathbf{A} = \mathbf{X}\,\mathrm{diag}(1, 3, 10, 30, 100)$, and let $\mathbf{b} = \mathbf{A}\mathbf{w}^\star$ for a random $\mathbf{w}^\star$.

   1. Apply `gd_iterations` of the `#numerical-stability-conditioning-ridge` cell, which takes $\mathbf{A}^\top\mathbf{A}$ and $\mathbf{A}^\top\mathbf{b}$, to this design at $\lambda = 0$. Compute $\kappa(\mathbf{A}^\top\mathbf{A})$ and compare the step count with the bound Equation 35.111. Explain the shortfall through the components of the start's error along the two extreme eigenvectors of $\mathbf{A}^\top\mathbf{A}$.
   2. Divide every column of $\mathbf{A}$ by its norm and repeat the measurement. How does the minimizer of the rescaled problem map back to $\mathbf{w}^\star$?
   3. Part 2 of Exercise 1 of Section 35.2 writes gradient descent in new coordinates $\mathbf{u}$ with $\mathbf{w} = \mathbf{T}\mathbf{u}$ as gradient descent on the original problem preconditioned by a positive definite matrix $\mathbf{P}$, with update $\mathbf{w} \leftarrow \mathbf{w} - \eta\,\mathbf{P}^{-1}\nabla f(\mathbf{w})$. Which $\mathbf{T}$ and which $\mathbf{P}$ correspond to column scaling? How is $\mathbf{P}$ related to the diagonal of $\mathbf{A}^\top\mathbf{A}$?

5. code **Stochastic rounding.** ● Stochastic rounding (Gupta et al. 2015) rounds a real number x between adjacent representable values $x_- < x_+$ to x_+ with probability $q = (x - x_-)/(x_+ - x_-)$ and to x_- otherwise.

   1. Show that the rounded value has expectation x and variance $q(1-q)\Delta^2 \le \Delta^2/4$, where $\Delta = x_+ - x_-$.
   2. A bf16 weight starts at $w = 1$ and receives $K = 1000$ updates of -10^{-4}, each followed by stochastic rounding. The weight stays in $[0.5, 1]$, where the bf16 spacing is $\Delta = 2^{-8}$. Show that the rounding errors of different steps are uncorrelated, and predict the mean and the standard deviation of the final weight. Which final weight does round-to-nearest produce?
   3. For a positive fp32 number, adding a uniformly random integer from $\{0, \dots, 2^{16} - 1\}$ to its bit pattern and then clearing the lower 16 bits rounds it stochastically to bf16. Explain why. Implement this rounding next to the `to_bf16` function of Section 35.8.1 and check the predictions of the previous sub-item with 1000 independent runs. Compare with an fp32 master copy that receives the same updates.

6. **Exact subtraction.** ● Subtraction can be exact and still return an inaccurate difference.

   1. Show that the bound Equation 35.103 is attained for suitable δ_a and δ_b with $|\delta_a|, |\delta_b| \le u$.
   2. Prove Sterbenz's lemma (Sterbenz 1974): if a and b are floating-point numbers in the format Equation 35.95 with $b/2 \le a \le 2b$, then $a - b$ is a floating-point number, so the computed difference is exact. Assume that the format includes

subnormal numbers.

3. In the `#numerical-stability-conditioning-log1p` cell, the fp32 difference $a - b$ misses the true value 3.0×10^{-7} by about 20 percent. Use Sterbenz's lemma and the printed number of ulps to verify that the subtraction was exact, compute the relative errors of the stored a and b, and check that the bound of the first sub-item covers the error.

7. code **Merging running moments.** ● Data-parallel training computes statistics on shards and combines them. Let shard A hold k_A values with mean m_A and centered sum of squares M_A, and let shard B hold k_B values with m_B and M_B.

   1. Derive exact formulas for the mean m and the centered sum of squares M of the union of the two shards in terms of k_A, m_A, M_A, k_B, m_B, and M_B.
   2. Show that the case $k_B = 1$ is Welford's recursion Equation 35.104.
   3. Split the 10^5 samples of the `#numerical-stability-conditioning-welford` cell into 2^j shards of nearly equal size (`np.array_split`) for $j \in \{1, 4, 8\}$. Compute each shard's statistics with Welford's recursion, merge them pairwise in a binary tree, and compare the variance with the two-pass reference. Merge the shards a second time by adding their sums $\sum_i x_i$ and $\sum_i x_i^2$ and applying the one-pass formula. Explain the difference.

8. code **Summation error growth.** ● Let $S_n = \sum_{i=1}^{n} x_i$ and let $\hat{S}_n$ be its computed value. In-order summation has the worst-case bound $|\hat{S}_n - S_n| \le (n-1)\, u \sum_i |x_i| + O(u^2)$.

   1. For $n = 2^k$, recursive pairwise summation splits the terms into two halves, sums each half recursively, and adds the two results. Prove that it satisfies $|\hat{S}_n - S_n| \le u \log_2(n) \sum_i |x_i| + O(u^2)$.
   2. For $n \in \{2^4, 2^8, 2^{12}, 2^{16}, 2^{20}\}$ uniform random numbers in $[0, 1)$ stored as fp32, measure the relative errors of in-order summation (`np.cumsum(x)[-1]`), of `np.sum` (a blocked pairwise summation), and of Kahan's compensated summation (Kahan 1965) against a float64 reference. Kahan's method keeps a correction c: after each addition $t = s + y$ it sets $c = (t - s) - y$ and subtracts c from the next term. Plot the three errors against n together with the two bounds.
   3. How does the measured error of in-order summation grow with n compared with its worst-case bound, and why? Why is the text's sum of 10^7 copies of 0.1 so much closer to the worst case?
   4. What limits the error of the two accurate methods at large n?

   Adapted from Steven G. Johnson, MIT 18.335 Introduction to Numerical Methods, Problem Set 1, Problem 4.

9. code **Mixed-precision iterative refinement.** ●● Mixed-precision training does the bulk of its arithmetic in low precision and keeps the quantities that accumulate small corrections in high precision. Iterative refinement (Moler 1967; Carson and Higham 2018) applies the same split to $\mathbf{Ax} = \mathbf{b}$: the $O(n^3)$ factorization runs in low precision and only $O(n^2)$ residual work runs in high precision. Starting from an fp32 solution $\mathbf{x}_0$, it repeats

$$\mathbf{r}_k = \mathbf{b} - \mathbf{A}\mathbf{x}_k, \qquad (\mathbf{A} + \Delta\mathbf{A}_k)\,\mathbf{d}_k = \mathbf{r}_k, \qquad \mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{d}_k,$$

   where the residual and the update are computed in fp64 and the correction $\mathbf{d}_k$ comes from an fp32 solve.

1. Model the fp32 solve as backward stable, $\|\Delta\mathbf{A}_k\| \le c\,u_{32}\,\|\mathbf{A}\|$ with $u_{32} = 2^{-24}$ and a moderate constant c, and neglect the fp64 rounding errors. Show that the error $\mathbf{e}_k = \mathbf{x}_k - \mathbf{x}$ satisfies $\mathbf{e}_{k+1} = (\mathbf{A} + \Delta\mathbf{A}_k)^{-1}\Delta\mathbf{A}_k\,\mathbf{e}_k$. Derive a contraction factor in terms of $c\,\kappa(\mathbf{A})\,u_{32}$, and state when it is below 1.
2. Implement the iteration for the Hilbert matrices of the `#numerical-stability-conditioning-hilbert` cell with $n = 3, \dots, 10$, using `np.linalg.solve` on fp32 copies of $\mathbf{H}$ and $\mathbf{r}_k$ for the correction. Record the forward error over ten iterations and compare its final value with that of an fp64 solve. For which n does refinement reach the accuracy of the fp64 solve? For which n does it still converge, but too slowly to reach that accuracy within ten iterations? For which n does the error grow?
3. Estimate the measured contraction per iteration for each n and compare it with $\kappa(\mathbf{H})\,u_{32}$ and with the spectral radius of $\mathbf{I} - \widehat{\mathbf{H}}^{-1}\mathbf{H}$, where $\widehat{\mathbf{H}}$ is the fp32 copy of $\mathbf{H}$ and the inverse is applied in fp64. Which perturbation dominates $\Delta\mathbf{A}_k$ in this experiment, and how pessimistic is the bound of the first sub-item?

35.9 Further Reading

This section points beyond the chapter, and its topics are not all equally urgent. Five of them bear directly on training neural networks, so we place them first under their respective headings: the edge of stability, iterate averaging, interpolation, the Fisher matrix, and saddle-point methods. The remaining topics come from classical optimization and numerical analysis, followed by general textbooks, courses, and surveys; Section 9.11 surveys the practical side of training. Nothing else in the book depends on this section, so feel free to browse selectively.

35.9.1 Gradient Methods

35.9.1.1 Self-Stabilization at the Edge of Stability

When you train a neural network with full-batch gradient descent, something surprising happens: the sharpness of the loss (its largest Hessian eigenvalue) climbs up to the stability limit $2/\eta$ and then hovers right around it rather than diverging (Cohen et al. 2021). The quadratic model in Section 35.1.2.4 explains the first half of this behavior, showing why the top eigenmode grows as soon as the sharpness crosses $2/\eta$, yet a purely quadratic model predicts unbounded divergence and cannot explain what pulls the sharpness back down. Damian et al. (2023) resolve the puzzle by keeping just one more term in the Taylor expansion: the cubic term along the top eigenvector. Once oscillations build along that direction, this cubic coupling pushes the iterate against the gradient of the sharpness itself. That lowers the sharpness back below $2/\eta$, letting the oscillation die down until the cycle repeats. Without the cubic term, the oscillation would simply grow unchecked.

35.9.1.2 Optimal Polynomials and Conjugate Gradients

Consider minimizing a quadratic $f = \frac{1}{2}\mathbf{x}^\top\mathbf{A}\mathbf{x}$ whose eigenvalues lie in $[\mu, L]$ and whose minimizer is $\mathbf{0}$. Any first-order method whose k-th step stays inside the span of the gradients seen so far produces an iterate of the form $\mathbf{x}_k = p_k(\mathbf{A})\mathbf{x}_0$, where p_k is a polynomial of degree at most k satisfying $p_k(0) = 1$. Viewed this way, designing a fast gradient method on quadratics is equivalent to choosing a polynomial that is as small as possible across the

interval $[\mu, L]$. The polynomial that achieves the smallest maximum over the whole interval is a scaled Chebyshev polynomial, whose peak value is $2\rho^k/(1+\rho^{2k}) \approx 2\rho^k$, with the exact same contraction factor $\rho = (\sqrt{\kappa}-1)/(\sqrt{\kappa}+1)$ that appeared for heavy ball (Section 35.1.3). When the dimension d is large compared with k, the eigenvalues can spread across the entire interval, so no method in this family can beat that worst-case rate. The conjugate gradient method does even better in practice: at every step it automatically finds the polynomial that minimizes f for the *actual* matrix $\mathbf{A}$ and starting point $\mathbf{x}_0$, without ever needing to know μ or L, and in exact arithmetic it terminates after at most d steps. Nocedal and Wright (2006) (ch. 5) derive both the standard method and its preconditioned variant.

35.9.1.3 Rates Certified by Semidefinite Programs

Proving tight convergence rates by hand is famously delicate. Our hand proof in Section 35.3.3.3 yielded a per-step contraction factor of $1-\mu/L$, whereas the section quoted the exact factor $(1-\mu/L)^2$, which cuts the required step count in half. That tighter bound comes from a technique called *performance estimation*. Instead of searching for a clever algebraic trick by hand, Drori and Teboulle (2014) cast the worst-case performance of an algorithm over all L-smooth convex functions as an optimization problem over the Gram matrix (the table of all pairwise inner products) of its iterates and gradients. Bounding that problem leads to a small semidefinite program, whose dual solution literally produces a human-checkable proof. Taylor et al. (2018) obtain the squared factor in just this way. Taking a control-theoretic perspective, Lessard et al. (2016) view the gradient as an unknown nonlinearity inside a linear dynamical system constrained only by μ and L, certifying convergence rates with a semidefinite program whose size stays fixed regardless of the number of steps. Remarkably, the same machinery also uncovers counterexamples: it is how they discovered the function on which the heavy-ball method fails to converge (Section 35.1.3).

35.9.2 Stochastic and Adaptive Methods

35.9.2.1 Averaging with a Constant Step

As we saw in Section 35.4.1, constant-step SGD settles into a noise floor proportional to the step size η. Shrinking η over time eliminates that floor, but it also slows down late progress and makes the schedule sensitive to tuning. Averaging the iterates offers a cleaner way out: the individual iterates keep bouncing around the minimum, while their running average cancels the zero-mean noise. Polyak and Juditsky (1992) show that averaging iterates obtained with step sizes decaying more slowly than $1/k$ achieves the asymptotically optimal covariance. For least squares, Bach and Moulines (2013) go one step further and show that a *constant* step size is already enough: the averaged iterate reaches an error of order $1/n$ after n examples, even without strong convexity.

35.9.2.2 Interpolation

Classical stochastic optimization assumes that individual training examples pull the parameters in conflicting directions at the optimum, creating an unavoidable noise floor. Modern overparameterized neural networks often operate in a very different regime: when a model fits every training example exactly, every per-example gradient vanishes at the minimizer. The noise floor then disappears, and on a strongly convex loss constant-step SGD converges linearly without any variance reduction (Section 35.4.4). Ma et al. (2018) analyze this interpolation regime for least squares and identify the critical batch size beyond which larger

minibatches stop speeding up training; intriguingly, that threshold is governed by the ratio of per-example curvature to full-batch curvature rather than by gradient noise. Under a related *strong growth condition*, where the gradient variance shrinks in proportion to the squared norm of the full gradient, SGD can even be accelerated to match momentum rates (Vaswani et al. 2019).

35.9.2.3 The Fisher and the Gauss–Newton Matrix

Second-order approximations such as K-FAC (Section 35.5.5) aim to approximate the Fisher information matrix $\mathbf{F} = \mathbb{E}[\nabla \log p \, \nabla \log p^\top]$, where the expectation draws labels from the model's own predictive distribution (Amari 1998; Martens and Grosse 2015). For standard losses like softmax cross-entropy and squared error, $\mathbf{F}$ coincides with the Gauss–Newton matrix $\mathbf{J}^\top \mathbf{H}_{\text{out}} \mathbf{J}$, where $\mathbf{H}_{\text{out}}$ is the Hessian of the loss with respect to the network's output. More generally, this identity holds whenever the network output parameterizes the natural parameter of an exponential-family likelihood (Section 36.4). A common pitfall in practice is substituting the *empirical* Fisher, which averages the same outer product over the ground-truth dataset labels instead of sampling from the model. It is cheaper to compute and is often labeled "Fisher" in codebases, yet in general it equals neither the true Fisher nor the Gauss–Newton matrix (Martens 2020).

35.9.2.4 Mirror Descent

In the metric view of AdaGrad (Section 35.5.1), step lengths are measured in a quadratic Euclidean norm. That geometry fits unconstrained spaces like $\mathbb{R}^d$, where all directions look alike, but it is awkward on the probability simplex, where coordinates must stay nonnegative and sum to one and multiplicative updates are far more natural. Mirror descent adapts the geometry of gradient descent by replacing the squared distance $\frac{1}{2}\|\mathbf{x} - \mathbf{y}\|^2$ in the proximal form of the gradient step with a Bregman divergence $D_\phi(\mathbf{x}, \mathbf{y}) = \phi(\mathbf{x}) - \phi(\mathbf{y}) - \nabla\phi(\mathbf{y})^\top(\mathbf{x} - \mathbf{y})$ (Nemirovski and Yudin 1983; Beck and Teboulle 2003). Choosing ϕ to be the negative entropy turns the update into the exponentiated gradient rule $x_i \leftarrow x_i e^{-\eta g_i} / \sum_j x_j e^{-\eta g_j}$. When gradients are bounded coordinatewise, this choice improves the regret defined in Section 35.5.1 on the d-dimensional simplex from the $\sqrt{d}$ scaling of projected gradient descent to a much gentler $\sqrt{\log d}$.

35.9.3 Constraints and Duality

35.9.3.1 Saddle-Point Methods

The saddle points we met in Lagrangian duality (Section 35.7.2) also arise directly as the equilibria of two-player games, including GAN training (Section 16.1). Unfortunately, the naive strategy of simultaneous gradient descent on one player and gradient ascent on the other behaves badly: on the simple bilinear game xy, it spirals outward from the saddle point for every positive step size (exercise 5). The extragradient method breaks this spiral by peeking ahead: it takes a trial step, evaluates the gradient at that lookahead point, and applies that gradient from the original iterate. On convex–concave problems with Lipschitz gradients, its averaged iterates converge at the rate $O(1/k)$ (Nemirovski 2004). Optimistic methods achieve a similar stabilizing effect at half the gradient cost by reusing the previous step's gradient as a cheap proxy for the trial gradient, an idea that Daskalakis et al. (2018) apply successfully to stabilize GAN training.

35.9.3.2 Problem Classes and Interior-Point Methods

The dual problems we solved by hand in Section 35.7.5 are quadratic programs, which sit inside a standard nested hierarchy of convex optimization classes: linear programs, quadratic programs, second-order cone programs, and semidefinite programs. Modern interior-point methods solve all of these classes to high accuracy in polynomial time. Rather than handling the inequalities $g_i(\mathbf{x}) \le 0$ directly, they add a smooth logarithmic barrier $-\frac{1}{s}\sum_i \log(-g_i(\mathbf{x}))$ to the objective and track the unconstrained minimizers as s grows, tracing out the *central path* toward the boundary. At each point along this path, the quantities $-1/(s\, g_i(\mathbf{x}))$ automatically form dual-feasible multipliers, certifying a duality gap of m/s for m constraints. This gives an automated version of the optimality certificates we built by hand in Section 35.7 (Boyd and Vandenberghe 2004; Nesterov 2018).

35.9.3.3 Operator Splitting and ADMM

The proximal gradient steps of Section 35.6.4.1 handle a smooth loss plus a single simple nonsmooth penalty. Many structured problems instead have the composite form $\min_{\mathbf{x}} f(\mathbf{x}) + h(\mathbf{A}\mathbf{x})$, where both terms are convex and possibly nonsmooth; its Fenchel dual is $\max_{\mathbf{y}} -f^*(-\mathbf{A}^\top \mathbf{y}) - h^*(\mathbf{y})$ (Rockafellar 1970). To decouple the two terms so each can be handled separately, we introduce an auxiliary variable $\mathbf{z} = \mathbf{A}\mathbf{x}$ and add a quadratic penalty $\frac{\tau}{2}\|\mathbf{A}\mathbf{x} - \mathbf{z}\|^2$ to the Lagrangian. Minimizing alternately over $\mathbf{x}$, then over $\mathbf{z}$ (which reduces to a proximal step on h), and finally taking a dual ascent step on $\mathbf{y}$ gives the alternating direction method of multipliers (ADMM), which converges for every $\tau > 0$. Boyd et al. (2011) show how this splitting naturally distributes large optimization problems across machines, and Parikh and Boyd (2014) provide a handy catalog of closed-form proximal operators for common regularizers and constraints.

35.9.3.4 The Frank–Wolfe Method

Projected gradient descent requires projecting onto the constraint set at every iteration. That is effortless for a box or an ℓ_2 ball, but on a nuclear-norm ball (matrices whose singular values sum to at most a given bound) a single projection costs a full SVD. The Frank–Wolfe method (Frank and Wolfe 1956) avoids projection altogether by minimizing a linear approximation of the objective over the constraint set instead: it finds $\mathbf{s}_k = \operatorname{argmin}_{\mathbf{s}\in C} \nabla f(\mathbf{x}_k)^\top \mathbf{s}$ and takes a convex combination $\mathbf{x}_{k+1} = (1-\gamma_k)\mathbf{x}_k + \gamma_k \mathbf{s}_k$ with step size $\gamma_k = 2/(k+2)$. On the nuclear-norm ball this linear subproblem needs only the top singular vector pair of the gradient, and on the probability simplex it reduces to finding the single smallest coordinate. The method converges at rate $O(LD^2/k)$ on a set of diameter D, and as a bonus the quantity $\nabla f(\mathbf{x}_k)^\top(\mathbf{x}_k - \mathbf{s}_k)$ provides an upper bound on $f(\mathbf{x}_k) - f^\star$ at every step with no extra computation (Jaggi 2013).

35.9.4 Floating Point

35.9.4.1 Probabilistic Rounding-Error Analysis

The worst-case summation bounds in Section 35.8.3 grow linearly as nu when n terms are added in sequence, yet measured rounding errors in practice are usually much smaller. The reason is that worst-case analysis assumes every single rounding error pushes in the same adversarial direction. Higham and Mary (2019) instead model the individual rounding errors as independent, zero-mean random variables and prove bounds of order $\sqrt{n}\,u$

that hold with high probability, matching the random-walk cancellation typically seen in experiments. Just as importantly, their analysis clarifies when the optimistic $\sqrt{n}\,u$ behavior breaks down: whenever the rounding errors are strongly correlated, as in that subsection's sum of 10^7 copies of 0.1, where long stretches of additions all round in the same direction.

35.9.4.2 Growth Factors and Iterative Refinement

In Section 35.8.4, we noted that LU factorization with partial pivoting is backward stable in practice. Strictly speaking, its backward error is bounded by a low-degree polynomial in n times $\rho_n u$, where the *growth factor* ρ_n measures the ratio of the largest intermediate entry created during elimination to the largest entry of $\mathbf{A}$. In the worst case, Wilkinson's famous pathological matrix drives ρ_n all the way to 2^{n-1}. Yet for almost every matrix encountered in real computations ρ_n remains modest, and if you need an ironclad guarantee, QR factorization eliminates exponential growth altogether at roughly twice the arithmetic cost (Higham 2002; Trefethen and Bau 1997). To recover high accuracy without paying for double precision everywhere, mixed-precision iterative refinement factors $\mathbf{A}$ cheaply in fp32, evaluates the residual in fp64, and solves for a correction using the low-precision factors; the final exercise of Section 35.8 determines just how ill conditioned $\mathbf{A}$ can get before this basic loop stalls. Using the low-precision factors instead as a preconditioner inside GMRES, an iterative solver for general linear systems, pushes mixed-precision refinement to far more severely conditioned matrices (Carson and Higham 2018).

35.9.5 General Optimization Resources

Books

- Convex Optimization — Boyd & Vandenberghe: convex sets and functions, duality, and interior-point methods, with a free PDF; the reference for Section 35.3, Section 35.6, and Section 35.7.
- Numerical Optimization — Nocedal & Wright (2nd ed., Springer): line search, trust regions, quasi-Newton and conjugate-gradient methods, and constrained optimization, in more depth than Section 35.1, Section 35.2, and Section 35.6.
- Lectures on Convex Optimization — Nesterov (2nd ed., Springer): complexity bounds and accelerated first-order methods, with the general proofs that Section 35.1 and Section 35.3 cite.
- Optimization for Data Analysis — Wright & Recht (Cambridge, 2022): gradient, accelerated, stochastic, and coordinate methods for data analysis, a companion to Section 35.1 through Section 35.5.
- Accuracy and Stability of Numerical Algorithms — Higham (2nd ed., SIAM): rounding-error analysis, summation, and the conditioning of linear systems and least squares (Section 35.8).
- Mathematics for Machine Learning — Deisenroth, Faisal & Ong: a free, self-contained chapter on gradient descent, constrained optimization, and convexity.

Courses and Video Lectures

- Stanford EE364a: Convex Optimization I — Boyd: slides, homework, and software for the Boyd & Vandenberghe text, with 2023 lecture videos (Section 35.3, Section 35.7).
- CMU 10-725: Convex Optimization — Tibshirani: ML-oriented slides and scribed notes on subgradients, proximal and stochastic methods, duality, and ADMM, with Fall 2016 lecture videos (Section 35.4, Section 35.6, Section 35.7).

Tutorials, Notes, and Surveys

- Why Momentum Really Works — Goh (Distill): interactive figures of momentum on convex quadratics (Section 35.1.3).
- Optimization Methods for Large-Scale Machine Learning — Bottou, Curtis & Nocedal: the theory of stochastic-gradient methods for large-scale learning (Section 35.4).
- An overview of gradient descent optimization algorithms — Ruder: momentum, Nesterov acceleration, AdaGrad, RMSProp, and Adam side by side (Section 35.5).
- What Every Computer Scientist Should Know About Floating-Point Arithmetic — Goldberg: the IEEE format, rounding, and cancellation (Section 35.8).
- Optimization (Chapter 5), Patterns, Predictions, and Actions — Hardt & Recht: a concise account of optimization in supervised learning (Section 35.1, Section 35.4).

Chapter 36

Probability and Statistical Learning

Throughout the main text, we train regression models with squared error and classifiers with cross-entropy (Section 2.1, Section 3.1), report accuracies on finite test sets, and pick models by comparing validation scores (Section 2.6). Every one of those steps draws an inference from a finite sample. Consider a coin that lands heads 9 times in 13 flips: its empirical bias is $9/13 \approx 0.69$, yet a fair coin produces at least nine heads more than one time in eight. A test accuracy is that same coin flipped once per test example, and this chapter develops the mathematics that explains when finite samples support reliable conclusions and when they mislead us.

Sorting out sample uncertainty rests on four distinct ideas: a *distribution* describing how outcomes vary; an *estimator* mapping a sample to a number, whose spread across hypothetical samples differs from a *posterior* conditioned on the single sample in front of us; and a *generalization bound* certifying when low error on a finite sample guarantees low error on fresh data.

We take these four ideas in order. First, we establish continuous densities (Section 36.1), joint distributions and Jacobian transformations (Section 36.2), the standard named distributions (Section 36.3), and their unification in the exponential family (Section 36.4), which explains why linear, logistic, and softmax regression all share the gradient "prediction minus observation." Next, Section 36.5 develops error bars, hypothesis tests, confidence intervals, and the bootstrap, while Section 36.6 derives our training losses from maximum likelihood and links parameter variance to Fisher curvature. Hidden variables motivate the evidence lower bound and expectation–maximization (Section 36.7), followed by Laplace, Monte Carlo, Markov chain Monte Carlo, and variational approximations when posteriors lack closed forms (Section 36.8). Finally, Section 36.9 proves finite-sample tail bounds and high-dimensional shell concentration, Section 36.10 extends them to post-selection uniform convergence (Chapter 3.5), and Section 36.11 puts distributions, smoothing, and calibration to work on image classification.

Readers familiar with Section 1.6 can skim the opening moment algebra, and Section 36.8 and Section 36.11 can be skipped on a first pass without breaking later dependencies. Full entropy and divergence theory continues in Chapter 37, while measure theory, time series, and nonparametric extensions are surveyed in Section 36.12.

36.1 Random Variables

In Section 1.6, we built probability by counting. A die has six faces, a label belongs to one of ten classes, and the probability of an event is simply the sum of the probabilities of its outcomes. Most of the numbers a neural network works with are not like that. A pixel intensity, a weight drawn at initialization, or the Gaussian noise that a diffusion model adds to an image can each take any value in an interval, and there are far too many values to list one by one. What, then, is the probability of a single one of them?

Think of a dart thrown at a round board so carelessly that it is equally likely to land anywhere on the board, and suppose it comes to rest at half the radius from the center. Asked how likely that was, we would round the distance (say, to between 0.45 and 0.55 radii), which happens about one time in ten. Asking for one more digit of precision makes the probability ten times smaller, and another digit makes it ten times smaller again. With infinitely many digits, the probability is zero, and yet the dart clearly landed somewhere.

The way out is to stop asking for the probability of a single point and ask instead how fast probability accumulates near it. Each time we narrow the interval tenfold, its probability drops tenfold, so the ratio of probability to width stays constant. This probability per unit length is the *density*. With it, every tool of discrete probability carries over to quantities that vary continuously, with sums replaced by integrals.

Some of what follows revisits Section 1.6, now extended to continuous variables and backed by proofs: we already motivated the mean and variance there, so we move straight to their continuous forms and algebra. Alongside them, we introduce the density itself, a general method for drawing samples from the cumulative distribution function, the tail bounds of Markov and Chebyshev, and a bell-shaped density whose tails are so heavy that both bounds fail. If you carry only two results out of this section, make them the relationship between the density and the cumulative distribution function Equation 36.6 and Markov's inequality Equation 36.14, from which the rest of this chapter's tail bounds descend. We follow the dart throughout, focusing on a single number, its distance from the center, and leave its two coordinates and transformations of densities for Section 36.2.

36.1.1 Densities

We develop the idea of a density in three steps: first at a single point, as the limit of ever narrower bins; next over an interval, as an area under a curve; and finally through the function that accumulates that area, which also gives us a practical way to draw random samples.

36.1.1.1 From Bins to a Density

To make "probability per unit length" precise, we first need the probability that the dart lands within a given range of distances. Let R be the dart's distance from the center of a board of radius 1. The dart lands within distance r of the center whenever it hits the disk of radius r. Because every part of the board is equally likely, this probability is simply that disk's share of the board's area:

$$P(R \le r) = \frac{\pi r^2}{\pi} = r^2, \qquad 0 \le r \le 1.$$

Half the radius holds only a quarter of the probability, because it holds only a quarter of the

area. A bin of width ϵ centered at 0.5 therefore has probability $(0.5+\epsilon/2)^2-(0.5-\epsilon/2)^2 = \epsilon$, which produces the numbers from our opening example:

bin	width ϵ	probability	probability / ϵ
$[0.45, 0.55]$	0.1	0.1	1
$[0.495, 0.505]$	0.01	0.01	1
$[0.4995, 0.5005]$	0.001	0.001	1

Each tenfold narrowing divides the probability by ten, leaving the ratio unchanged. That limiting ratio is the quantity we want to name. The *probability density* of a random variable X at x is the probability per unit length near x,

$$p(x) = \lim_{\epsilon \downarrow 0} \frac{P(x \le X < x + \epsilon)}{\epsilon},$$

wherever this limit exists. Equivalently, for a short interval, the probability is approximately its length times the density,

$$P(x \le X < x + \epsilon) \approx \epsilon\, p(x), \tag{36.1}$$

with an error that shrinks faster than ϵ. This describes the density one point at a time, which is all we need for now; we give the formal definition in terms of intervals in the next subsection. For our dart at any distance r in $(0, 1)$, the interval $[r, r + \epsilon)$ has probability $(r + \epsilon)^2 - r^2 = 2r\epsilon + \epsilon^2$. Dividing by the width ϵ gives $2r + \epsilon$, and letting $\epsilon \to 0$ yields

$$p(r) = 2r, \qquad 0 \le r \le 1. \tag{36.2}$$

The density grows linearly from the center to the rim, mirroring the circumference of the circle at radius r: there is simply more board farther out. At $r = 0.5$ the density equals 1, matching the ratio in the table above.

Notice that a density is not a probability. At $r = 0.9$, the dart's density is 1.8, and a variable that is equally likely to land anywhere in $[0, 0.1]$ has density 10 on that interval, since it packs a total probability of 1 into a length of 0.1. A density can take any non-negative value, though it can exceed 10 only on a set of total length less than $1/10$, or else the probability of that set would exceed 1. Think of a density as a rate, much like speed. A car's speed tells you how fast distance accumulates, not how far the car has traveled. In the same way, a density of 1.8 at $r = 0.9$ tells you how fast probability accumulates as r increases past 0.9, not how much probability sits at the point 0.9 itself.

Because a single point has zero length, it also carries zero probability. For every $\epsilon > 0$, the point x lies inside $[x, x + \epsilon)$, so $P(X = x)$ is at most about $\epsilon\, p(x)$, which vanishes as ϵ shrinks to zero. Thus, for any random variable with a density,

$$P(X = x) = 0 \qquad \text{for every } x. \tag{36.3}$$

This resolves our opening puzzle: the dart lands at distance 0.5 with probability zero, just as it does at every other exact distance, and yet it must land somewhere. In continuous probability, an event of probability zero is not impossible.

A single point that does carry positive probability is called an *atom*. A variable with a density has no atoms, whereas a discrete variable such as a die roll consists entirely of atoms. Some variables have both. For example, the output of a ReLU unit fed a standard normal input equals 0 with probability $\frac{1}{2}$ (an atom) and spreads the remaining half of its probability over $(0, \infty)$ with a density (Exercise 2 works out the details).

The table above came from an exact area formula, but a simulation needs no formula at all. To throw darts uniformly at the circular board, a computer can throw them uniformly at the surrounding square $[-1, 1]^2$ and keep only those that land inside the circle, which leaves every part of the board equally likely. Below, we count the fraction of darts falling in each of our three bins and divide by the bin width.

```
rng = np.random.default_rng()
xy = rng.uniform(-1, 1, size=(4 * 10**6, 2))          # darts on the square
on_board = (xy**2).sum(axis=1) <= 1                    # keep those on the board
r_dart = np.linalg.norm(xy[on_board], axis=1)          # distances R
for width in [0.1, 0.01, 0.001]:
    prob = np.mean(np.abs(r_dart - 0.5) <= width / 2)  # fraction in the bin
    print(f'width {width:<5}  probability {prob:.5f}  ratio {prob / width:.3f}')
```

```
width 0.1    probability 0.09983  ratio 0.998
width 0.01   probability 0.01002  ratio 1.002
width 0.001  probability 0.00099  ratio 0.990
```

As expected, each tenfold narrowing reduces the fraction tenfold while keeping the ratio close to $1 = p(0.5)$. What the exact table hid, however, is how noisy a very narrow bin becomes in practice. Roughly 3.1 million of the four million darts land on the board, and only about $3{,}000$ of those fall into the bin of width 0.001, so the final ratio fluctuates by a few percent from run to run.

36.1.1.2 Probability as Area

So far we have read the density off from the probabilities of tiny intervals. Now let us reverse the direction: given a density, how do we find the probability of a full interval? Partition the real line into bins $[\epsilon i, \epsilon(i+1))$ of width ϵ. By Equation 36.1, each bin has probability roughly $\epsilon\, p(\epsilon i)$, and because the bins are disjoint, their probabilities add. Summing widths times heights forms a Riemann sum, which approaches the area under the curve, namely the integral (Section 34.6), as ϵ shrinks to zero. Since X must take some real value, all bins together have probability one:

$$1 = P(X \in \mathbb{R}) \approx \sum_i \epsilon\, p(\epsilon i) \;\xrightarrow[\epsilon \to 0]{}\; \int_{-\infty}^{\infty} p(x)\, dx.$$

Summing only the bins between a and b suggests that the probability of an interval is simply the area under the density above it. Strictly speaking, this argument is motivation rather than a proof: there are roughly $1/\epsilon$ bins, and their individual errors, each small compared with ϵ, sum to a negligible total only if they shrink uniformly across x. To put things on solid footing, we turn this area formula into our formal definition.

Definition. A random variable X *has the density* p when $p \geq 0$ and, for all $a < b$,

$$P(X \in (a, b]) = \int_a^b p(x)\, dx. \tag{36.4}$$

Letting $a \to -\infty$ and $b \to \infty$ confirms that the total area under any density is 1. In fact, these two conditions, non-negativity ($p \geq 0$) and unit total area, are all a function needs to qualify as a valid density. Figure 36.1 illustrates this definition for a density with two peaks. Our dart passes the test as well: for $0 \leq a < b \leq 1$, the integral $\int_a^b 2r\, dr = b^2 - a^2$ recovers the area-based probability $P(a < R \leq b)$, confirming that $2r$ is indeed its density. In particular, the probability of landing in the outer half of the radius is $\int_{0.5}^1 2r\, dr = 1 - 0.25 = 0.75$: fully three quarters of the board lies more than half the radius from the center.

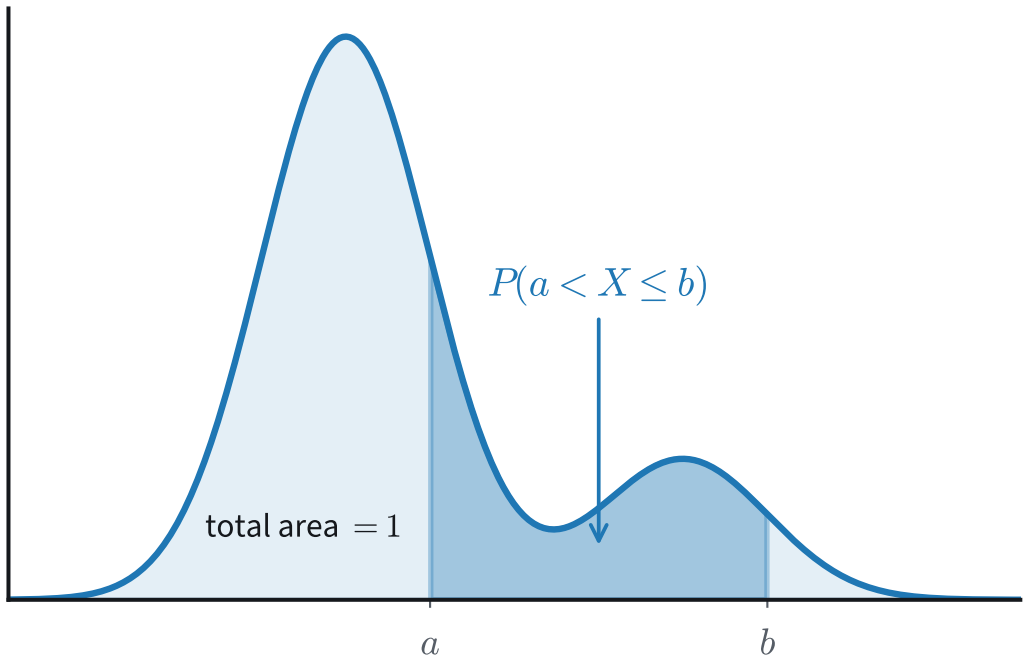


Figure 36.1: A probability density $p(x)$. The total area under the curve is 1; the probability that X lands in the interval from a to b is the area of the shaded slab above it, $\int_a^b p(x)\, dx$. The density itself is not a probability (it may exceed 1); only its integral over an interval is.

36.1.1.3 The Cumulative Distribution Function

Often it is convenient to work with a function whose values are genuine probabilities, and one that handles discrete and continuous variables alike. The *cumulative distribution function* (c.d.f.) of X gives the probability that X lands at or below x. When X has a density, this is simply the area under the density to the left of x:

$$F(x) = P(X \leq x) = \int_{-\infty}^x p(t)\, dt. \tag{36.5}$$

In other words, F sweeps across the real line from left to right, accumulating probability as it goes. It starts at 0 as $x \to -\infty$ and rises to 1 as $x \to \infty$. It can never decrease, because the non-negative density p only adds area, and whenever X has a density, F is continuous. For our dart, we computed it at the very beginning: $F(r) = 0$ for $r < 0$, $F(r) = r^2$ for $0 \le r \le 1$, and $F(r) = 1$ for $r > 1$. That function is where we started, and the density $2r$ came directly from it.

Unlike a density, the c.d.f. is defined for every random variable. A fair coin that reports 0 or 1 has

$$F(x) = \begin{cases} 0 & x < 0, \\ \frac{1}{2} & 0 \le x < 1, \\ 1 & x \ge 1, \end{cases}$$

which is a *staircase* that jumps upward by the probability of each atom. Because $F(x) = P(X \le x)$ includes any atom sitting right at x, the function takes the upper value at every jump. More generally, every c.d.f. is *right-continuous*, meaning $\lim_{y \downarrow x} F(y) = F(x)$. A mixed variable combines both behaviors: the ReLU output mentioned earlier has a c.d.f. that is 0 for $x < 0$, jumps to $\frac{1}{2}$ at $x = 0$, and then climbs smoothly toward 1. A single function therefore unifies discrete, continuous, and mixed random variables; Figure 36.2 shows one of each.

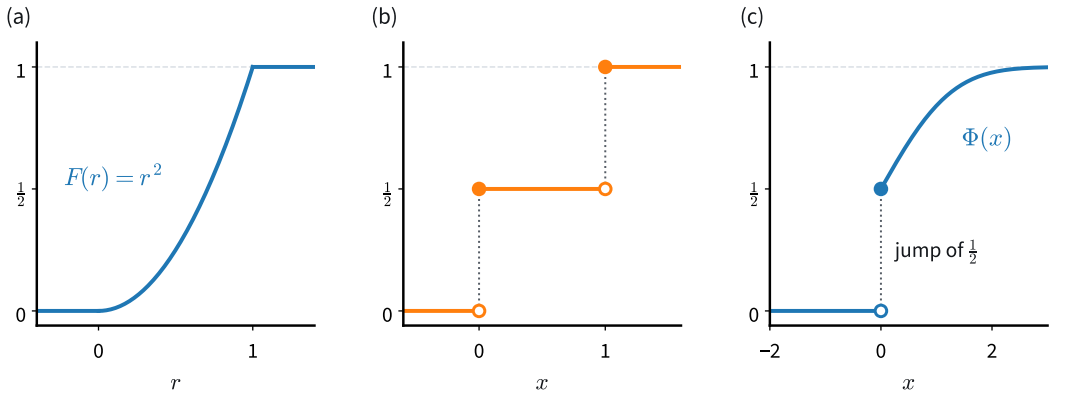


Figure 36.2: Three cumulative distribution functions. (a) The dart's distance from the center: $F(r) = r^2$ rises continuously from 0 to 1. (b) A fair coin reporting 0 or 1: a staircase that jumps by $\frac{1}{2}$ at each atom and takes the upper value there (filled dots). (c) The ReLU of a standard Gaussian: F is 0 for $x < 0$, jumps to $\frac{1}{2}$ at $x = 0$, and from there follows the Gaussian c.d.f. $\Phi(x)$.

When a density exists, the c.d.f. and the density carry the same information: integrating p produces F, and differentiating F recovers p. This brings back our opening limit of shrinking bins, now established as a theorem.

Proposition (density is the derivative of the c.d.f.). *If X has a density p that is continuous at x, then F is differentiable at x and*

$$F'(x) = p(x). \tag{36.6}$$

In particular, the probability per unit length $P(x \le X < x+\epsilon)/\epsilon$ tends to $p(x)$ as $\epsilon \downarrow 0$.

Proof. By Equation 36.5, F is the area-so-far function of p. The fundamental theorem of calculus says that the derivative of an area-so-far function is the integrand at every point where the integrand is continuous (Section 34.6). For the second statement, the single point x has probability at most $\int_{x-\delta}^{x} p(t)\,dt$ for every $\delta > 0$, and these integrals tend to zero with δ. Including or excluding an endpoint therefore changes nothing, so $P(x \le X < x+\epsilon) = F(x+\epsilon) - F(x)$, and the ratio is simply a difference quotient of F. ■

For the dart, $F'(r) = 2r$ on $(0,1)$, recovering the density we found from the bins. Figure 36.3 places the two functions side by side: the shaded area $\int_a^b p$ on the left equals the vertical rise $F(b) - F(a)$ on the right.

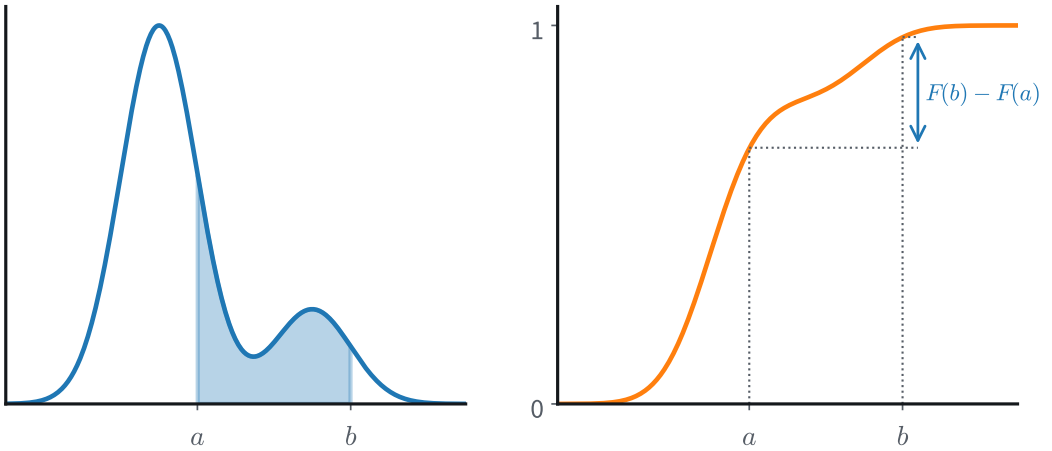


Figure 36.3: Left: a density p with the area over the interval from a to b shaded. Right: the matching c.d.f. $F(x) = \int_{-\infty}^{x} p$, an increasing curve from 0 to 1; the rise $F(b) - F(a)$ equals the shaded area on the left. The density is the slope of the c.d.f. $(F' = p)$ and the c.d.f. is the accumulated area under the density.

36.1.1.4 Quantiles and Inverse-Transform Sampling

Computers give us ready access to random numbers that are equally likely to fall anywhere between 0 and 1. To simulate the dart, or any other continuous distribution, we need a way to turn those uniform draws into samples from our target distribution.

Our building block is the simplest density of all. A random variable U is *uniform on* $[0,1]$ if its density is 1 on $[0,1]$ and 0 elsewhere. Every subinterval of $[0,1]$ then has probability equal to its length, and the c.d.f. is simply $F_U(u) = u$ for $0 \le u \le 1$.

The c.d.f. maps a value x to a probability level u. Reversing that map, asking for the value below which a given fraction u of the probability lies, yields the *quantile function*. Whenever F is continuous and strictly increasing on the interval I where $0 < F < 1$, it maps I one-to-one onto $(0,1)$, and its inverse $F^{-1} : (0,1) \to I$ is the quantile function. Its values at $u = 0.25, 0.5, 0.75$ are the *quartiles*, with the middle quartile $F^{-1}(0.5)$ called the *median*. For our dart, $F(r) = r^2$ inverts to $F^{-1}(u) = \sqrt{u}$, giving quartiles of 0.5, 0.707, and 0.866.

Notice that the median dart lands at 0.707, well past half the radius, because three quarters of the board lies beyond 0.5.

Remarkably, the quantile function is also a universal sampler.

Proposition (inverse-transform sampling). *Let F be a continuous c.d.f. that is strictly increasing on the interval I where $0 < F < 1$, and let U be uniform on $[0, 1]$. Then $X = F^{-1}(U)$ has c.d.f. F.*

Proof. With probability one, U lies in $(0, 1)$, so X is defined and lies in I. For x in I, the events $F^{-1}(U) \le x$ and $U \le F(x)$ coincide, because F is increasing on I. Since U is uniform on $[0, 1]$, the second event has probability equal to the length of $[0, F(x)]$:

$$P(F^{-1}(U) \le x) = P(U \le F(x)) = F(x).$$

For x to the left of I, $F(x) = 0$ and $X > x$ always; for x to the right of I, $F(x) = 1$ and $X \le x$ always. ■

Notice that this hypothesis allows F to be flat outside an interval, just as the dart's c.d.f. is ($F = 0$ for $r < 0$ and $F = 1$ for $r > 1$). Replacing F^{-1} with the *generalized inverse* $F^{-}(u) = \min\{x : F(x) \ge u\}$ removes the hypothesis altogether and samples staircase c.d.f.s as well.

For the dart, inverse-transform sampling gives $R = \sqrt{U}$. It is tempting to take the shortcut $R = U$, and we can see immediately why that fails without running a single line of code: $R = U$ places half the darts inside $r < 0.5$, where only a quarter belong. Figure 36.4 shows the geometry of the construction on our two-bump density. Wherever the density is high, the c.d.f. rises steeply, so an evenly spaced set of levels on the vertical axis is compressed into a tight cluster of samples on the horizontal axis.

We now have two completely different ways to generate the dart's distance: throwing darts at the square and keeping those on the board, or simply computing $\sqrt{U}$. The cell below compares the two, alongside the naive shortcut $R = U$.

```
u = rng.uniform(size=10**6)
r = np.sqrt(u)                                          # R = F^{-1}(U)
print(f'P(R < 0.5): darts {np.mean(r_dart < 0.5):.3f}, sqrt(U) '
      f'{np.mean(r < 0.5):.3f}, naive U {np.mean(u < 0.5):.3f}')
print(f'mean distance: darts {r_dart.mean():.3f}, sqrt(U) {r.mean():.3f}')
```

```
P(R < 0.5): darts 0.250, sqrt(U) 0.250, naive U 0.500
mean distance: darts 0.667, sqrt(U) 0.667
```

The two valid methods agree. Both place a quarter of their distances below 0.5, and both have a mean distance close to 0.667, a number we will derive in the next subsection. The naive shortcut U puts half its samples below 0.5 because it spreads distances evenly across $[0, 1]$, ignoring the fact that the board has far more area near the rim than near the center.

36.1.2 Mean and Variance

A density specifies an entire distribution, yet two summary numbers often tell us most of what we need to know: where the distribution is centered, and how widely it spreads. Sec-

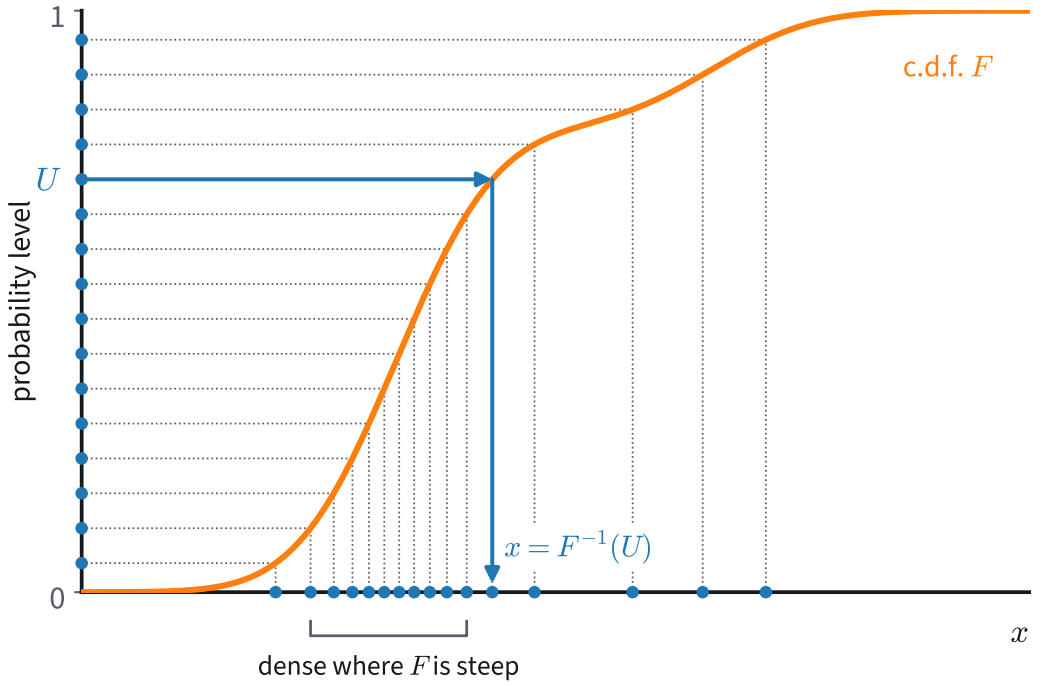


Figure 36.4: Inverse-transform sampling. Uniform levels U on the vertical axis (left) reflect through the c.d.f. F down to values $x = F^{-1}(U)$ on the horizontal axis. Where F is steep, that is, where the density $p = F'$ is large, a uniform spread of levels is compressed into a dense cluster of samples, so the histogram of reflected points reproduces p.

tion 1.6 introduced both for discrete variables and motivated their use; here we extend them to densities and prove the algebraic rules we rely on.

36.1.2.1 The Mean

The mean averages the values of X, weighting each by its probability. For a variable with a density, slicing the real line into bins of width ϵ as before shows that the bin at ϵi contributes its value ϵi times its probability, roughly $\epsilon\, p(\epsilon i)$, and the resulting sum tends to an integral. The *mean*, or *expectation*, of X is therefore

$$\mu_X = \mathbb{E}[X] = \sum_i x_i\, p_i \quad \text{(discrete)}, \qquad \mu_X = \mathbb{E}[X] = \int_{-\infty}^{\infty} x\, p(x)\, dx \quad \text{(density)}. \tag{36.7}$$

Just as in the discrete case, the mean acts as a balance point: a thin rod with mass density p along its length balances at μ_X. For this definition to make sense, we require $\sum_i |x_i|\, p_i < \infty$ in the discrete case and $\int |x|\, p(x)\, dx < \infty$ for a density. When that condition fails, the mean does not exist, and we will soon meet a concrete density where this happens. Using the same probability weights lets us average any function $g(X)$ directly,

$$\mathbb{E}[g(X)] = \sum_i g(x_i)\, p_i, \qquad \mathbb{E}[g(X)] = \int_{-\infty}^{\infty} g(x)\, p(x)\, dx,$$

again provided the sum or integral of $|g|$ is finite. Notice how convenient this is: we weight the values of g by the probabilities of X, without first having to work out the distribution of the transformed variable $g(X)$. This rule is famously called the *law of the unconscious statistician*, because it is so natural to apply that one easily forgets it is a theorem rather than the definition of $\mathbb{E}[g(X)]$. Keep in mind that $\mathbb{E}[g(X)]$ generally differs from $g(\mathbb{E}[X])$; when g is convex, Jensen's inequality (Section 35.3.3.1) tells us which of the two is larger.

For our dart,

$$\mathbb{E}[R] = \int_0^1 r \cdot 2r\, dr = \tfrac{2}{3}.$$

The mean distance is 0.667, not 0.5, again because the board has more area near the rim than near the center. Notice also that the mean sits slightly below the median 0.707: the relatively few darts that land near the center pull the average downward more than they shift the middle value.

The property of expectation we use most often is linearity. We relied on it in Section 1.6; here we prove it.

Proposition (linearity of expectation). *For random variables X, Y whose means exist and constants c, d,*

$$\mathbb{E}[cX + d] = c\,\mathbb{E}[X] + d, \qquad \mathbb{E}[X + Y] = \mathbb{E}[X] + \mathbb{E}[Y]. \tag{36.8}$$

Proof. Both identities follow from the linearity of the sum (or integral) that defines the mean. For the first, $\mathbb{E}[cX+d] = \sum_i (cx_i + d)p_i = c\sum_i x_i p_i + d\sum_i p_i = c\,\mathbb{E}[X] + d$, using $\sum_i p_i = 1$. For the second, let p_{ij} be the probability that $(X, Y) = (x_i, y_j)$, the joint distribution of the pair as in Section 1.6. Splitting the sum and recognizing the marginals $\sum_j p_{ij} = P(X = x_i)$ and $\sum_i p_{ij} = P(Y = y_j)$ gives

$$\mathbb{E}[X+Y] = \sum_{i,j} (x_i + y_j)\, p_{ij} = \sum_{i,j} x_i\, p_{ij} + \sum_{i,j} y_j\, p_{ij} = \mathbb{E}[X] + \mathbb{E}[Y].$$

For densities, the exact same computation runs with integrals over a joint density, the continuous counterpart of p_{ij}, which Section 36.2 defines. ■

Crucially, the second identity requires no independence at all: the mean of a sum equals the sum of the means no matter how tightly X and Y depend on each other. Every time we take the expected loss as an average over a dataset, we are relying on this fact.

36.1.2.2 Variance and Standard Deviation

The mean tells us where a distribution sits, but nothing about how tightly it clusters there. As in Section 1.6, we measure spread by the average squared distance from the mean, called the *variance*,

$$\sigma_X^2 = \operatorname{Var}(X) = \mathbb{E}\big[(X - \mu_X)^2\big], \tag{36.9}$$

which is defined whenever $\mathbb{E}[X^2] < \infty$. Expanding the square and applying linearity Equation 36.8, treating μ_X as a constant, yields a form that is usually much easier to compute: the mean of the square minus the square of the mean,

$$\operatorname{Var}(X) = \mathbb{E}\big[X^2 - 2\mu_X X + \mu_X^2\big] = \mathbb{E}[X^2] - 2\mu_X\,\mathbb{E}[X] + \mu_X^2 = \mathbb{E}[X^2] - \mu_X^2. \tag{36.10}$$

For a variable with a density, the mean and variance are therefore given by the two integrals

$$\mu_X = \int_{-\infty}^{\infty} x\, p(x)\, dx, \qquad \sigma_X^2 = \int_{-\infty}^{\infty} x^2 p(x)\, dx - \mu_X^2. \tag{36.11}$$

For the dart, $\mathbb{E}[R^2] = \int_0^1 r^2 \cdot 2r\, dr = \frac{1}{2}$, so $\operatorname{Var}(R) = \frac{1}{2} - \frac{4}{9} = \frac{1}{18} \approx 0.056$. For the uniform variable U on $[0,1]$, $\mu_U = \int_0^1 u\, du = \frac{1}{2}$ and $\operatorname{Var}(U) = \frac{1}{3} - \frac{1}{4} = \frac{1}{12}$.

Rescaling units or centering data are affine transformations. Under an affine map $cX + d$, the shift d drops out of the variance while the scale factor c comes out squared. Indeed, linearity gives $\mu_{cX+d} = c\mu_X + d$, so the deviation from the mean is $(cX + d) - \mu_{cX+d} = c(X - \mu_X)$; squaring and taking the expectation gives

$$\operatorname{Var}(cX + d) = c^2 \operatorname{Var}(X). \tag{36.12}$$

Two further properties are worth keeping in mind. First, variance is never negative, and it equals zero if and only if X is constant with probability one, since the expectation of the

non-negative quantity $(X-\mu_X)^2$ can vanish only when that quantity is zero with probability one. Second, variances of independent random variables add (Section 1.6); we will prove this in Section 36.2.2.1, along with the covariance term that replaces independence when variables interact.

Because variance squares the deviations, it carries squared units, such as dollars squared for a profit measured in dollars. Taking its square root restores the original units and gives the *standard deviation*,

$$\sigma_X = \sqrt{\mathrm{Var}(X)}, \tag{36.13}$$

which scales linearly under affine maps as $\sigma_{cX+d} = |c|\,\sigma_X$. For our dart, $\sigma_R = \sqrt{1/18} \approx 0.236$, measured in units of the board's radius.

36.1.3 Tails

The standard deviation sets a natural scale for typical fluctuations around the mean. A natural next question is how much probability can sit far out in the tails when all we know is the mean, or the mean and variance. Two classic inequalities answer this for every distribution at once. Both bounds are quite crude, yet surprisingly they are the sharpest possible without extra assumptions, as a simple three-point distribution will demonstrate. We then turn to a bell-shaped density that has neither a mean nor a variance, showing what those two assumptions were protecting us from.

36.1.3.1 Markov's and Chebyshev's Inequalities

The first bound uses only the mean and applies to any non-negative random variable. Its proof takes a single line, yet nearly every tail bound in this chapter comes from applying it to a well-chosen function of the variable.

Proposition (Markov's inequality). *If $X \geq 0$ has a mean and $t > 0$, then*

$$P(X \geq t) \leq \frac{\mathbb{E}[X]}{t}. \tag{36.14}$$

Proof. Let $\mathbf{1}_{X\geq t}$ be the indicator variable that equals 1 when $X \geq t$ and 0 otherwise. Then $X \geq t\,\mathbf{1}_{X\geq t}$ for every outcome: when $X \geq t$ the right-hand side is $t \leq X$, and otherwise it is $0 \leq X$. Taking expectations of both sides and using $\mathbb{E}[\mathbf{1}_{X\geq t}] = P(X \geq t)$ gives $\mathbb{E}[X] \geq t\,P(X \geq t)$. ■

Geometrically, on $[0, \infty)$ the step function that is 0 below t and t from t onward stays underneath the line $y = x$ (Figure 36.5), so its average value $t\,P(X \geq t)$ must stay underneath the average value of X. Setting t to ten times the mean gives an immediate rule of thumb: at most a tenth of any non-negative population can exceed ten times its average, whether we are looking at incomes, file sizes, or gradient norms.

Because Markov's inequality uses only the mean, it is necessarily crude. Applying it to the squared deviation $(X - \mu_X)^2$, whose mean is the variance, bounds both tails at once. This result is due to Bienaymé and Chebyshev (Chebyshev 1867).

Proposition (Chebyshev's inequality). *For any X with finite, nonzero variance and any $\alpha > 0$,*

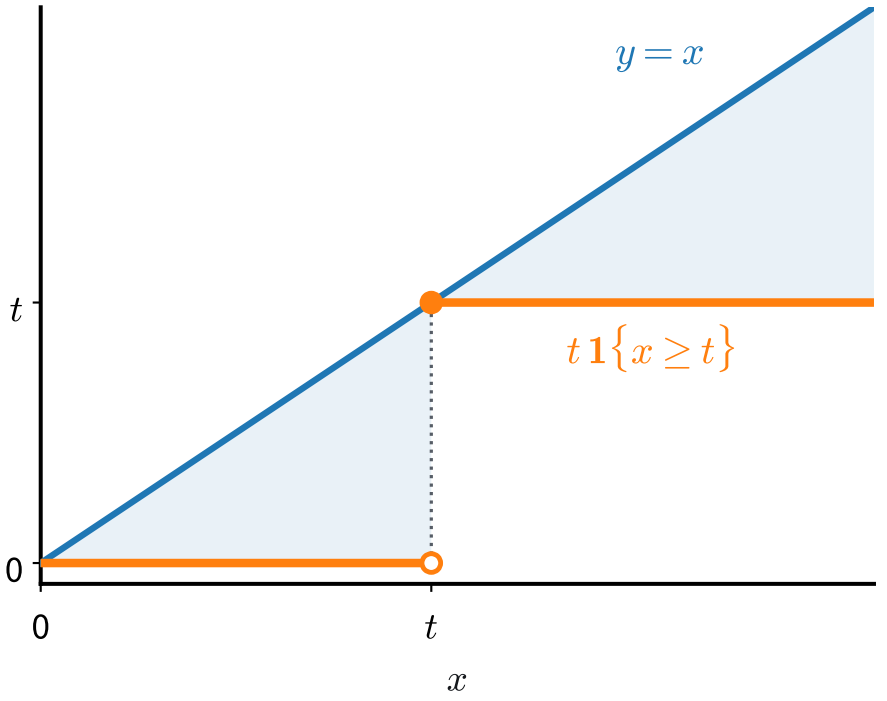


Figure 36.5: Geometry of Markov's inequality. For $x \geq 0$ the step function $t\,\mathbf{1}\{x \geq t\}$ (orange) lies under the line $y = x$ (blue), so its average $t\,P(X \geq t)$ cannot exceed $\mathbb{E}[X]$. The shaded gap is what the bound gives away.

$$P(|X - \mu_X| \geq \alpha\sigma_X) \leq \frac{1}{\alpha^2}. \tag{36.15}$$

Proof. The variable $Z = (X - \mu_X)^2$ is non-negative with mean $\text{Var}(X) = \sigma_X^2$, and the event $|X - \mu_X| \geq \alpha\sigma_X$ is identical to the event $Z \geq \alpha^2\sigma_X^2$. Since $\sigma_X > 0$, the threshold $t = \alpha^2\sigma_X^2$ is positive, so Markov's inequality Equation 36.14 gives $P(|X - \mu_X| \geq \alpha\sigma_X) \leq \sigma_X^2/(\alpha^2\sigma_X^2) = 1/\alpha^2$. ■

In particular, at most 1% of any distribution can lie ten or more standard deviations from its mean. Running the same proof with an absolute threshold $\epsilon > 0$ in place of $\alpha\sigma_X$ gives an equivalent form that requires only finite variance and also covers the degenerate case $\sigma_X = 0$:

$$P(|X - \mu_X| \geq \epsilon) \leq \frac{\text{Var}(X)}{\epsilon^2}.$$

For our dart, $\mu_R = 0.667$ and $\sigma_R = 0.236$. Two standard deviations below the mean is 0.195, while two above lies outside the board altogether, so $P(|R - \mu_R| \geq 2\sigma_R) = P(R \leq 0.195) = 0.195^2 \approx 0.038$, far smaller than Chebyshev's bound of $\frac{1}{4}$.

Given how loose $\frac{1}{4}$ is for the dart, can a bound valid for every distribution be sharpened? Not without further assumptions, and a distribution supported on just three points proves it. Let X take the values $a - 2$ and $a + 2$ with probability q each, and a with the remaining probability $1 - 2q$, where $0 < q \leq \frac{1}{2}$. We call this the *three-atom example*. By symmetry its mean is a, and its variance is $\text{Var}(X) = \mathbb{E}[(X - a)^2] = 4q + 4q = 8q$, so $\sigma_X = 2\sqrt{2q}$. Take $\alpha = 2$. As q shrinks, the threshold $2\sigma_X = 4\sqrt{2q}$ shrinks with it, while the two outer atoms stay fixed at distance 2 from a. Figure 36.6 shows the three resulting regimes. When $q > \frac{1}{8}$, the threshold exceeds 2, so neither outer atom falls in the tail and the probability is 0. At $q = \frac{1}{8}$, the threshold lands right at 2, giving $\sigma_X = 1$ and placing $a \pm 2$ two standard deviations from the mean; both outer atoms now count, their total probability is $2q = \frac{1}{4}$, and Chebyshev's bound holds with equality. When $q < \frac{1}{8}$, both outer atoms still lie beyond the threshold, and their combined mass $2q$ falls below $\frac{1}{4}$.

The same construction settles every $\alpha \geq 1$. Choosing $q = 1/(2\alpha^2)$, which is at most $\frac{1}{2}$, gives $\sigma_X = 2\sqrt{2q} = 2/\alpha$, so the threshold $\alpha\sigma_X = 2$ lands squarely on the outer atoms and $P(|X - a| \geq \alpha\sigma_X) = 2q = 1/\alpha^2$. Equality holds, proving that no constant smaller than $1/\alpha^2$ can work for all distributions. (For $\alpha < 1$, the bound exceeds 1 and says nothing.)

Chebyshev's inequality therefore captures the worst case over all distributions with a given variance. For the distributions we usually encounter, it is quite loose. A Gaussian with mean μ and standard deviation σ, the bell curve with density $e^{-(x-\mu)^2/(2\sigma^2)}/(\sigma\sqrt{2\pi})$ studied in full in Section 36.3.2.3, puts far less probability beyond two and three standard deviations than Chebyshev allows:

distance from the mean	Chebyshev's bound	Gaussian
2σ or more	25%	4.6%
3σ or more	11%	0.27%

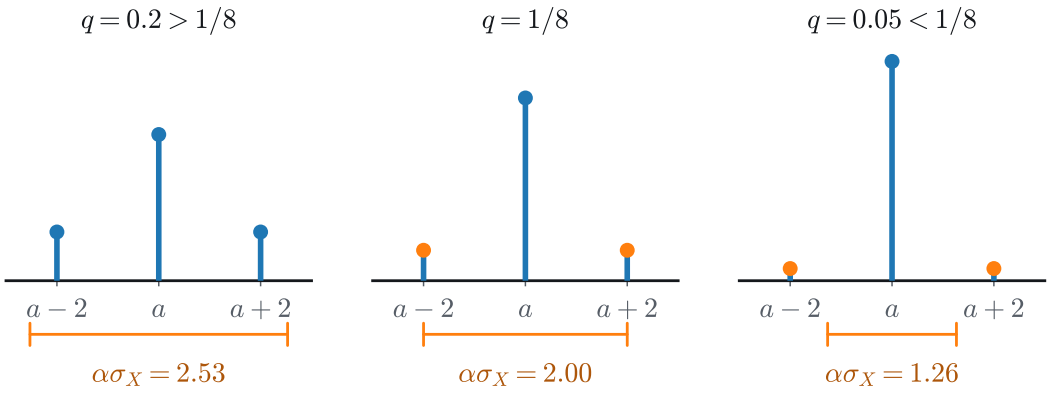


Figure 36.6: Chebyshev's bound on the three-atom example: masses q, $1-2q$, q drawn as vertical bars at $a-2$, a, $a+2$, with the threshold distance $\alpha\sigma_X = 4\sqrt{2q}$ for $\alpha = 2$ marked as a horizontal bracket from $a - \alpha\sigma_X$ to $a + \alpha\sigma_X$. The three panels show the regimes $q > 1/8$, $q = 1/8$ (where the bound is attained with equality), and $q < 1/8$. Orange atoms lie at or beyond the threshold and count toward the event $|X - a| \geq \alpha\sigma_X$.

The worst-case distribution concentrates all of its tail mass at a single distance, just as our three-atom example does. Typical distributions spread their tails out and thin them rapidly.

36.1.3.2 Heavy Tails

Chebyshev's inequality requires a finite variance, and the mean itself requires $\int |x|\, p(x)\, dx < \infty$. Surprisingly, both conditions can fail for a smooth, harmless-looking bell curve. The *Cauchy distribution* has the density

$$p(x) = \frac{1}{\pi(1 + x^2)}.$$

This is a valid density, since $\int_{-\infty}^{\infty} \frac{dx}{1+x^2} = [\arctan x]_{-\infty}^{\infty} = \pi$. Unlike a Gaussian, whose tails plummet like $e^{-x^2/2}$, the Cauchy tails decay only algebraically, like $1/(\pi x^2)$. The second moment fails first. Integrating over $[-L, L]$ gives

$$\int_{-L}^{L} x^2 p(x)\, dx = \frac{2}{\pi}(L - \arctan L),$$

which evaluates to 5.43, 62.7, and 636 at $L = 10$, 100, and 1000, growing without bound because the integrand $x^2/(\pi(1 + x^2))$ approaches the nonzero constant $1/\pi$. Hence $\mathbb{E}[X^2] = \infty$, and the Cauchy distribution has no finite variance.

The mean fails in a subtler way. Because the integrand $x\, p(x)$ is odd, it is tempting to declare the mean to be 0 by symmetry, and indeed $\int_{-L}^{L} x\, p(x)\, dx = 0$ for every L. However, the definition of expectation requires absolute convergence, and

$$\int_{-L}^{L} |x|\, p(x)\, dx = \frac{1}{\pi} \log(1 + L^2)$$

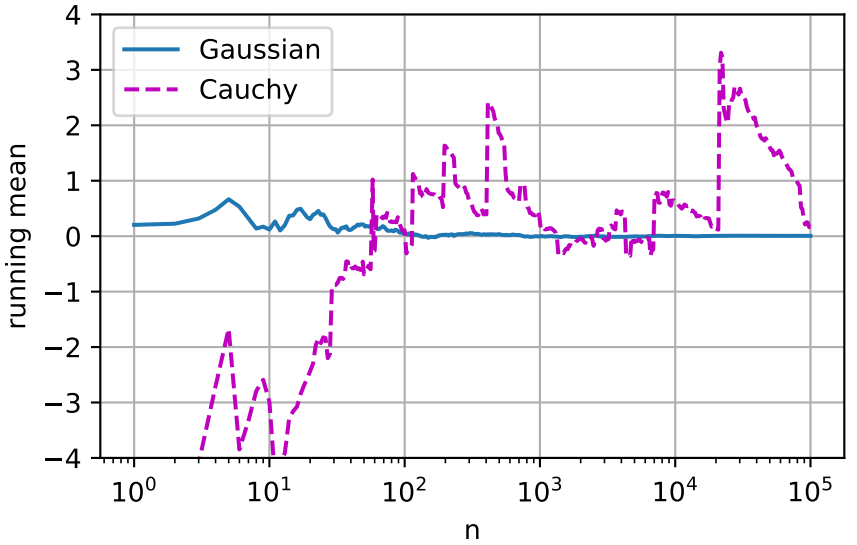
grows without bound as $L \to \infty$. Consequently, the Cauchy mean does not exist. Without absolute convergence, the value of the integral depends on how the limits are taken: over the lopsided interval $[-L, 2L]$, for example,

$$\int_{-L}^{2L} x\, p(x)\, dx = \frac{1}{2\pi} \log \frac{1 + 4L^2}{1 + L^2} \xrightarrow[L \to \infty]{} \frac{\log 2}{\pi} \approx 0.221.$$

With no mean to settle toward, what happens when we average independent Cauchy draws? One might still hope that averaging 1000 draws lands closer to 0 than a single draw, which falls outside $[-1, 1]$ with probability $P(|X| > 1) = 1 - \frac{2}{\pi} \arctan 1 = \frac{1}{2}$. The cell below plots the running mean of 10^5 Cauchy draws alongside that of 10^5 standard Gaussian draws, and then counts how many of 1000 separate averages (each over 1000 Cauchy draws) still lie farther than 1 from 0.

```
n = 10**5
idx = np.unique(np.logspace(0, 5, 500).astype(int)) - 1  # log-spaced n
running_mean = lambda x: (np.cumsum(x) / np.arange(1, n + 1))[idx]
d2l.plot(idx + 1, [running_mean(rng.standard_normal(n)),
                   running_mean(rng.standard_cauchy(n))],
         'n', 'running mean', legend=['Gaussian', 'Cauchy'],
         xscale='log', ylim=[-4, 4], figsize=(5, 3))
averages = rng.standard_cauchy((1000, 1000)).mean(axis=1)
print(f'averages of 1000 Cauchy draws with |average| > 1: '
      f'{np.mean(np.abs(averages) > 1):.2f}')
```

```
averages of 1000 Cauchy draws with |average| > 1: 0.49
```



The Gaussian running mean settles smoothly toward 0. The Cauchy running mean never settles: it lurches in sudden jumps, each triggered by a single extreme draw, and an order-one jump can strike at any n. Averaging brings no improvement at all. Roughly half of the averages of 1000 draws still lie farther than 1 from 0, matching the fraction for a single draw. In fact, as we state without proof, the average of n independent standard Cauchy variables has the same distribution as a single standard Cauchy variable.

A distribution is called *heavy-tailed* when its tails decay more slowly than every exponential $e^{-\lambda t}$ with $\lambda > 0$. Power-law tails, where $P(|X| > t)$ decays like $t^{-\alpha}$, are the canonical example: they force every moment $\mathbb{E}[|X|^k]$ with $k \geq \alpha$ to be infinite, with the Cauchy distribution corresponding to $\alpha = 1$. Heavy tails do not automatically imply infinite moments, however; the log-normal distribution, which we derive in Section 36.2.3.1, is heavy-tailed yet has all moments finite. Heavy tails matter in deep learning because sample averages over them converge slowly or fail to converge altogether. Minibatch gradient noise, for example, has been argued to be heavy-tailed (Simsekli et al. 2019), a hypothesis that Exercise 5 lets you test.

Everything in this section dealt with one number at a time, whereas our dart lands in a plane and has two coordinates. Those two coordinates turn out to be uncorrelated yet far from independent, and making sense of that requires densities over the plane, which we take up in Section 36.2.

36.1.4 Summary

A continuous random variable assigns probability zero to every individual value, so we describe its distribution through a density, the probability per unit length. Integrating the density over an interval gives that interval's probability, the c.d.f. accumulates this area from left to right with $F' = p$, and inverting the c.d.f. turns uniform noise into samples from the distribution. Markov's and Chebyshev's inequalities bound the tails of any distribution using only its mean or variance; Chebyshev's bound is tight in the worst case yet loose for typical distributions. Heavy-tailed distributions like the Cauchy have neither a mean nor a variance, and averaging their draws does nothing to reduce their spread. Above all, carry Markov's inequality forward: every tail bound in Section 36.9 descends from it.

36.1.5 Exercises

1. **Power-law tail.** Let X have the density $p(x) = 1/x^2$ for $x \geq 1$ and $p(x) = 0$ otherwise.
    1. Verify that p is a density, and compute $P(X > 2)$ and the c.d.f. $F(x)$ for every real x.
    2. Find the quantile function $F^{-1}(u)$ for $u \in (0, 1)$, the median, and the other two quartiles. Write down the inverse-transform sampler $X = F^{-1}(U)$ of Figure 36.4 for this distribution.
    3. Show that $\mathbb{E}[X] = \infty$. Clipping X at a level $c > 1$, as gradient clipping does for a gradient norm, gives $\min(X, c)$. Compute $\mathbb{E}[\min(X, c)]$, and describe how it grows with c.
    4. For the Cauchy density $p_C(x) = 1/(\pi(1 + x^2))$, derive the values of $\int_{-L}^{L} x\, p_C(x)\, dx$ and $\int_{-L}^{2L} x\, p_C(x)\, dx$ quoted in the text, and their limits as $L \to \infty$. What do the two limits say about the mean of the Cauchy distribution? Why

can no such ambiguity arise for the mean of the non-negative variable X of part (a)?

2. **Rectified Gaussian.** Let $X \sim \mathcal{N}(0,1)$ have the standard Gaussian density $p_X(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$ and c.d.f. $\Phi(x) = \int_{-\infty}^{x} p_X(t)\,dt$, and let $Y = \max(0, X)$. This is the output of a ReLU unit with a standard normal pre-activation, a common model of a unit at initialization.

    1. Express the c.d.f. F_Y through Φ and sketch it. Where does Y have an atom, and what is its mass? Is Y discrete, does it have a density, or is it mixed?
    2. Explain why Y has no density. For a strictly monotone, continuously differentiable transformation $Y = g(X)$, the change-of-variables rule of Section 36.2.3.1 gives the density $p_Y(y) = p_X(g^{-1}(y))\,|(g^{-1})'(y)|$. Which of its hypotheses fails for the ReLU?
    3. Clipping to $[-c, c]$ for a threshold $c > 0$, as the hard-tanh activation does for $c = 1$, gives $C = \min(\max(X, -c), c)$. Write down the c.d.f. of C, and find the threshold c at which the atoms of C together hold 10% of the probability. The equation has no closed-form solution; use tabulated values of Φ, or `math.erf` with $\Phi(z) = \frac{1}{2}(1 + \mathrm{erf}(z/\sqrt{2}))$.
    4. Show that $\mathbb{E}[Y^2] = \frac{1}{2}$ and $\mathbb{E}[Y] = 1/\sqrt{2\pi}$. Hint: write $X^2 = Y^2 + \min(0, X)^2$ and use the symmetry of X. The He initialization of Section 4.4.2.2 multiplies the weight variance by 2; which of these two moments does the factor compensate for?

3. code **Logistic distribution and calibration.** The logistic distribution has the c.d.f. $F(x) = \mathrm{sigmoid}(x) = 1/(1 + e^{-x})$.

    1. Find its density with Equation 36.6 and its quantile function. What are its quartiles?
    2. Let X have a continuous, strictly increasing c.d.f. F. Show that $F(X)$ is uniform on $[0, 1]$. This is the converse of inverse-transform sampling.
    3. A probabilistic regression model predicts for each input i a logistic distribution for the target y_i, centered at the correct location μ_i but with a scale s, that is, the c.d.f. $\mathrm{sigmoid}\big((y - \mu_i)/s\big)$. The actual errors $y_i - \mu_i$ follow the standard logistic distribution. Draw 10^4 errors with the quantile function from (a), compute $u_i = \mathrm{sigmoid}\big((y_i - \mu_i)/s\big)$ for $s \in \{0.5, 1, 2\}$, and plot the three histograms of the u_i. Which shape reveals predicted distributions that are too narrow, and which reveals distributions that are too wide? Such histograms are a standard check of the calibration of probabilistic forecasts (Gneiting et al. 2007).

4. **One-sided Chebyshev.** ● Chebyshev's inequality Equation 36.15 bounds both tails at once. Cantelli's inequality bounds one tail: for X with mean μ and variance σ^2, and for every $t > 0$,

$$P(X - \mu \geq t) \leq \frac{\sigma^2}{\sigma^2 + t^2}. \tag{36.16}$$

    1. For every $u \geq 0$, apply Markov's inequality Equation 36.14 to $(X - \mu + u)^2$ to show that $P(X - \mu \geq t) \leq (\sigma^2 + u^2)/(t + u)^2$. Choose u to prove Cantelli's inequality.
    2. Write the bound for $t = \alpha\sigma$, and compare it with Equation 36.15 at $\alpha = 1$ and $\alpha = 2$.

3. For given μ, σ and t, find a distribution on two points with mean μ and variance σ^2 that attains the bound.

5. [extended] **Averages of heavy tails. ●●** Simsekli et al. (2019) argued that the gradient noise of SGD in deep networks is heavy-tailed and better described by an α-stable distribution, whose variance is infinite for $\alpha < 2$, than by a Gaussian. This project measures how a Monte Carlo average behaves in that regime and then tests the premise on a small model. The Pareto density $p(x) = \alpha x^{-\alpha-1}$ on $x \geq 1$, with $\alpha > 0$, contains the density of problem 1 as the case $\alpha = 1$.

   1. Generalize problem 1b and c to this family: show that $\mathbb{E}[X^k] < \infty$ exactly when $k < \alpha$, compute the mean μ for $\alpha > 1$, and write down an inverse-transform sampler.

   2. Let $1 < \alpha < 2$, so that the mean is finite but the variance is not. Show that the largest of n independent draws exceeds t with probability close to $nt^{-\alpha}$ when $nt^{-\alpha}$ is small, and find the scale t_n that the largest draw typically reaches. Argue heuristically that the error $|\bar{X}_n - \mu|$ of the sample mean is of order $n^{1/\alpha-1}$ rather than $n^{-1/2}$. The generalized central limit theorem that Simsekli et al. invoke makes this argument precise.

   3. For $\alpha \in \{1.5, 2.5, 3\}$ and $n \in \{10, 10^2, \dots, 10^5\}$, estimate the median of $|\bar{X}_n - \mu|$ over 400 repetitions, and fit the slope of the logarithm of the median error against $\log n$. Compare the slope at $\alpha = 1.5$ with the rate $n^{1/\alpha-1}$ from (b). Compare the slopes at $\alpha = 2.5$ and $\alpha = 3$ with the rate $n^{-1/2}$ that holds when the variance is finite.

   4. In each repetition, form the interval $\bar{X}_n \pm 1.96\,\hat{\sigma}_n/\sqrt{n}$, where $\hat{\sigma}_n$ is the sample standard deviation; for Gaussian averages it covers μ with probability about 95%. Record how often it covers μ. Explain the coverage at $\alpha = 1.5$ and $n = 10^5$, given that $\hat{\sigma}_n$ is finite in every run. What does this imply for error bars computed from heavy-tailed samples, such as minibatch gradients?

   5. The Hill estimator (Hill 1975) estimates the tail index from the k largest of m positive draws $X_{(1)} \geq X_{(2)} \geq \cdots \geq X_{(m)}$ as

$$\hat{\alpha}_k = \Big(\frac{1}{k} \sum_{i=1}^{k} \log \frac{X_{(i)}}{X_{(k+1)}} \Big)^{-1}. \tag{36.17}$$

      Check it on 10^5 Pareto draws for $\alpha \in \{1.5, 3\}$ and $k \in \{10^2, 10^3, 10^4\}$. Then train the softmax regression of Section 3.3.5. At the trained parameters, collect the coordinates of the difference between the minibatch gradient and the full-batch gradient over 1000 minibatches, for batch sizes 32 and 256. Plot $\hat{\alpha}_k$ of their absolute values against k for both batch sizes. Is the estimate below 2? How much does the conclusion depend on k?

36.2 Joint Distributions and Transformations

In Section 36.1, we reduced our dart on a board of radius 1 to a single number: its distance R from the center, which has density $2r$. A dart actually lands at a point, however, with both a horizontal coordinate X and a vertical coordinate Y. Most quantities we model in

machine learning arrive in groups like this. An image consists of thousands of pixels whose values depend on one another, and every supervised learning problem pairs an input with a target. Knowing the density of each variable on its own is not enough, because separate densities say nothing about how the variables move together.

Two questions guide us in this section. The first is what one variable tells us about another. If the dart lands near the top of the board, at $Y = 0.9$, then X must lie within 0.44 of the center line, so observing Y certainly tells us something about X. Yet, surprisingly, we will find that our best single guess for X is still 0 no matter what Y is, and that the covariance between the two coordinates is zero. Dependence, predictability, and correlation are three distinct ideas, and we need the joint density of the pair to tell them apart. The second question is what happens to a density when we transform a random variable, as when a neural network layer acts on its input or a sampler reshapes uniform noise. We already saw that the dart's distance can be drawn as $R = \sqrt{U}$ from a uniform variable U, and taking the square root turns a flat density into the linear density $2r$. Clearly, a transformation does more to a density than simply relabel its axis.

Three results from this section will carry most of the weight later in the book: the law of total variance Equation 36.22, the conditional mean as the best squared-error predictor Equation 36.23, and the change-of-variables formula Equation 36.31. The covariance matrix and the geometric remark that a correlation is a cosine can safely be skipped on a first reading.

```
rng = np.random.default_rng()
```

36.2.1 Joint and Conditional Densities

Everything we might want to ask about a pair of continuous random variables is captured by a single function of two arguments. Let us define that function, and then see how to extract three essential quantities from it: the distribution of one coordinate on its own, the distribution of one coordinate once the other is observed, and the average of one given the other, which is what a regression model estimates.

36.2.1.1 Joint and Marginal Densities

Just as a one-dimensional density measures probability per unit length over short intervals, its two-dimensional counterpart measures probability per unit area over small squares. The *joint density* $p_{X,Y}$ of a pair (X, Y) gives the probability of landing in a small square of side ϵ at (x, y) as roughly its area times the density,

$$P\big(X \in [x, x+\epsilon],\ Y \in [y, y+\epsilon]\big) \approx \epsilon^2\, p_{X,Y}(x, y).$$

As in one dimension, tiling a region $\mathcal{D}$ of the plane with such squares leads to the property we take as the formal definition: $p_{X,Y} \geq 0$, and $P((X, Y) \in \mathcal{D}) = \iint_{\mathcal{D}} p_{X,Y}(x, y)\, dx\, dy$, so that the total volume under the joint density over the whole plane equals 1. For n variables, the same definition holds with n-dimensional boxes and n-fold integrals.

For our dart, every part of the unit disk is equally likely and the board has area π, so the joint density is $p_{X,Y}(x, y) = 1/\pi$ inside the disk and 0 outside. Notice that even though this density is flat on the disk, it cannot be written as a function of x times a function of y: the circular edge of the board ties the two coordinates together.

Often we know a joint density, yet we only care about the distribution of one coordinate on its own, called its *marginal distribution*. A thin vertical strip $X \in [x, x+\epsilon]$ has probability about $\epsilon\, p_X(x)$. Slicing that strip into small squares along y and adding their probabilities gives

$$\epsilon\, p_X(x) \approx \sum_i P\big(X \in [x, x+\epsilon],\ Y \in [\epsilon i, \epsilon(i+1)]\big) \approx \sum_i \epsilon^2\, p_{X,Y}(x, \epsilon i).$$

Canceling one factor of ϵ and recognizing the remaining sum as a Riemann sum in y, we obtain

$$p_X(x) = \int_{-\infty}^{\infty} p_{X,Y}(x, y)\, dy. \tag{36.18}$$

In words, the marginal density at x is the joint density integrated up the vertical strip at x, as illustrated in Figure 36.7: we simply *integrate out* the variable we do not need. For the dart, the vertical strip at x crosses the board from $-\sqrt{1-x^2}$ to $\sqrt{1-x^2}$, giving

$$p_X(x) = \int_{-\sqrt{1-x^2}}^{\sqrt{1-x^2}} \frac{1}{\pi}\, dy = \frac{2}{\pi}\sqrt{1-x^2}, \qquad -1 \le x \le 1,$$

which traces out a semicircle. As a quick check, $\int_{-1}^{1} \sqrt{1-x^2}\, dx$ is the area of half the unit disk, $\pi/2$, so p_X indeed integrates to 1. Even though the joint density is flat, the marginal density is curved: a vertical strip through the middle of the board is longer than one near the rim.

36.2.1.2 Conditional Densities and Independence

Once we observe that $Y = y$, how should our density for X change? For discrete variables, the answer is simply $P(X = x \mid Y = y) = P(X = x, Y = y)/P(Y = y)$. With continuous densities, however, that formula divides zero by zero because $P(Y = y) = 0$. The way out is the same limiting argument that defined the density in the first place: we condition on a thin horizontal strip $Y \in [y, y+\epsilon]$, which has positive probability, and then let the strip shrink. Applying our square approximation to the numerator and the one-variable rule to the denominator gives

$$P\big(X \in [x, x+\epsilon] \,\big|\, Y \in [y, y+\epsilon]\big) \approx \frac{\epsilon^2\, p_{X,Y}(x, y)}{\epsilon\, p_Y(y)} = \epsilon\, \frac{p_{X,Y}(x, y)}{p_Y(y)}.$$

Inside the strip, the probability of a short x-interval is its length ϵ times $p_{X,Y}(x, y)/p_Y(y)$. That ratio is the probability per unit length of X given $Y = y$, so we define the *conditional density* as

$$p_{X|Y}(x \mid y) = \frac{p_{X,Y}(x, y)}{p_Y(y)}, \qquad p_Y(y) > 0. \tag{36.19}$$

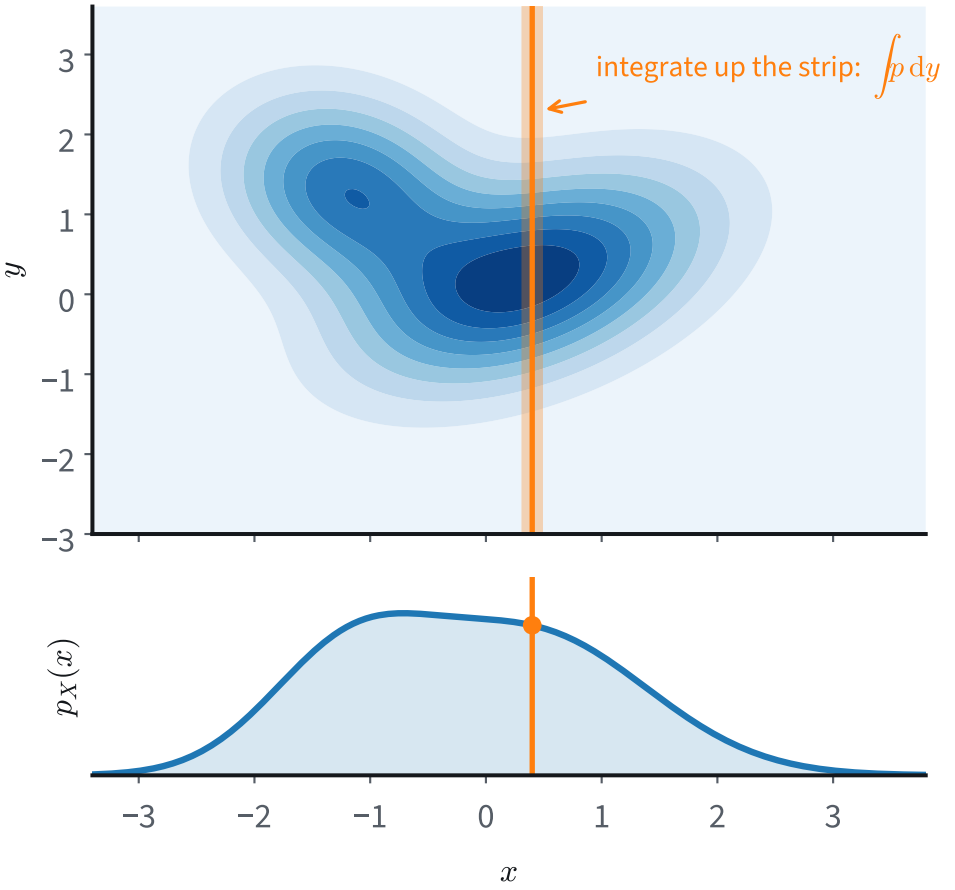


Figure 36.7: A continuous joint density $p_{X,Y}(x, y)$ shown by its contours, with one vertical strip at x highlighted. Integrating the joint density up that strip, over all y, gives the height of the marginal density $p_X(x)$ plotted along the x-axis below. Sweeping the strip across x traces out the whole marginal.

For every fixed y with $p_Y(y) > 0$, this is a valid density in x: it is non-negative, and by the marginal rule Equation 36.18 it integrates to $p_Y(y)/p_Y(y) = 1$. Geometrically, we slice the joint density along the horizontal line $Y = y$ and rescale that cross-section to unit area. For the dart, the horizontal line $Y = y$ cuts across the board on the chord $[-\sqrt{1-y^2}, \sqrt{1-y^2}]$, so

$$p_{X|Y}(x \mid y) = \frac{1/\pi}{(2/\pi)\sqrt{1-y^2}} = \frac{1}{2\sqrt{1-y^2}} \qquad \text{for } |x| \le \sqrt{1-y^2}.$$

Given $Y = y$, the horizontal position X is uniform along the chord at height y. Because that chord shrinks as $|y|$ grows, observing Y changes the distribution of X.

Rearranging Equation 36.19 gives the *chain rule* $p_{X,Y}(x, y) = p_{X|Y}(x \mid y)\, p_Y(y)$. Writing the joint density both ways, $p_{X|Y}\, p_Y = p_{Y|X}\, p_X$, and solving for $p_{X|Y}$ yields *Bayes' rule for densities*,

$$p_{X|Y}(x \mid y) = \frac{p_{Y|X}(y \mid x)\, p_X(x)}{p_Y(y)}. \tag{36.20}$$

Just as Bayes' theorem did for discrete variables in Section 1.6, this identity turns a model of how Y is generated from X into a distribution over X after seeing Y.

In the special case where observing Y tells us nothing new about X, so that $p_{X|Y}(x \mid y) = p_X(x)$ whenever $p_Y(y) > 0$, we call the two variables *independent* and write $X \perp Y$. By the chain rule, independence is equivalent to saying that the joint density factorizes, $p_{X,Y}(x, y) = p_X(x)\, p_Y(y)$ (up to sets of probability zero): multiply by $p_Y(y)$ to go one way and divide by it to go the other.

Intuitively, independence means that the conditional density of X is the same function of x, with the same location and the same shape, at every height y. Consider the joint density $p_{X,Y}(x, y) = 4xy$ on the unit square $[0, 1]^2$. Its marginal is $p_Y(y) = \int_0^1 4xy\, dx = 2y$, which gives $p_{X|Y}(x \mid y) = 4xy/(2y) = 2x$ regardless of y, so $X \perp Y$. The dart is the opposite case: its conditional density is uniform at every y, yet it lives on a chord whose length depends on y.

As Figure 36.8 shows, having the same shape is not enough for independence. For a correlated pair, the conditional densities at three heights share the same shape but have shifted centers, so the pair is dependent; only when the joint density factorizes do all three slices coincide.

Independence also gives a clean formula for the density of a sum. When $X \perp Y$, the pair $(X, X + Y)$ has joint density $p_X(x)\, p_Y(s - x)$ at (x, s), and integrating out x via Equation 36.18 leaves the *convolution* $p_{X+Y}(s) = \int p_X(x)\, p_Y(s - x)\, dx$.

36.2.1.3 Conditional Expectation

A regression model predicts a target from an input. To stay consistent with the previous subsection, we keep X as the target and Y as the observed input (whereas the main text writes the pair the other way around). Training with squared loss searches for a function h that makes $\mathbb{E}[(X - h(Y))^2]$ as small as possible. Which function is best? Within any thin

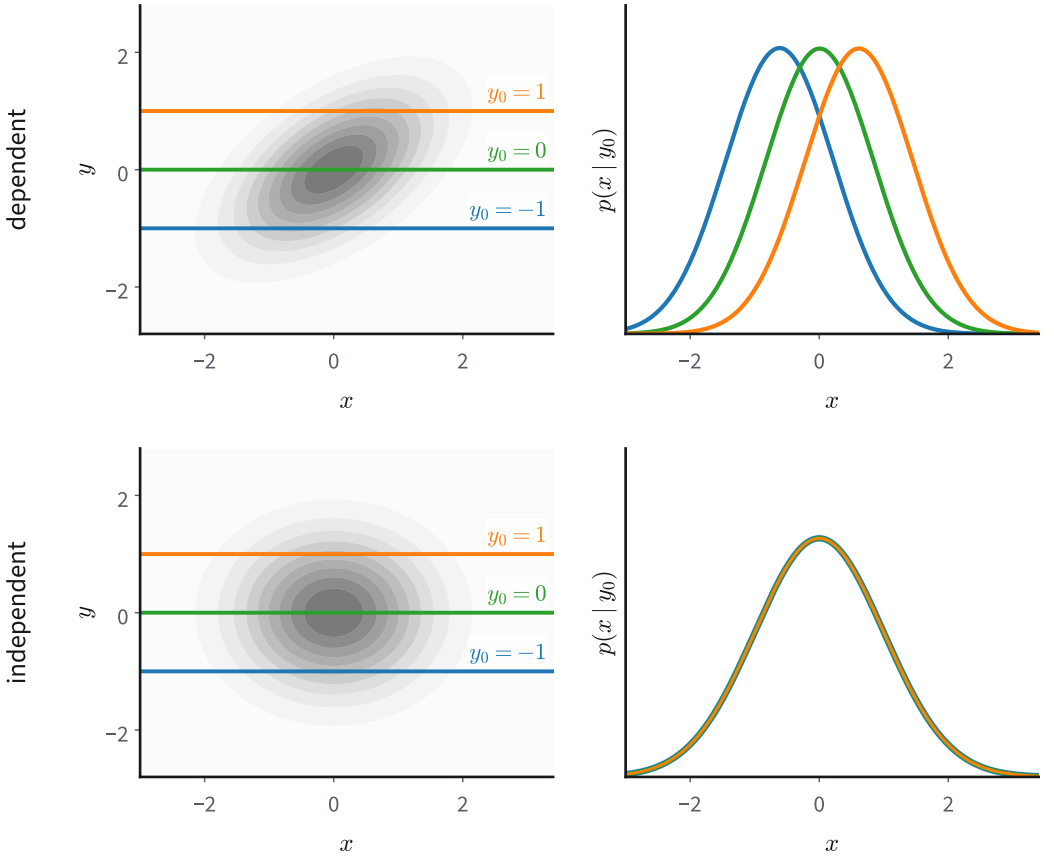


Figure 36.8: Conditioning as slicing. Left: a joint density with three horizontal slices at $y_0 = -1, 0, 1$. Right: each slice renormalized to unit area is the conditional density $p(x \mid y_0)$. Top: a correlated pair; the three conditionals have the same shape but shifted centers, so the conditional depends on y_0 and the pair is dependent. Bottom: a pair whose joint factorizes; the three conditionals coincide, which is independence.

strip $Y \approx y$, the model must predict a single number for an entire spread of possible targets. For any random variable X, the constant c closest to it in mean squared distance is its mean μ_X, since expanding the square around μ_X gives $\mathbb{E}[(X-c)^2] = \operatorname{Var}(X) + (\mu_X - c)^2$. That suggests predicting the average of X inside each strip. Accordingly, the *conditional expectation* of X given $Y = y$ is the mean of the conditional density,

$$\mathbb{E}[X \mid Y{=}y] = \int x\, p_{X|Y}(x \mid y)\, dx. \tag{36.21}$$

It is simply the average of X among all outcomes with $Y = y$. As y varies, $\mathbb{E}[X \mid Y{=}y]$ traces out a function of y; evaluated at the random variable Y, it becomes a random variable $\mathbb{E}[X \mid Y]$ that depends on Y alone. The same strip average defines $\mathbb{E}[g(X,Y) \mid Y{=}y] = \int g(x,y)\, p_{X|Y}(x \mid y)\, dx$ for any function g. Because a factor $h(Y)$ is constant within each strip, it pulls straight out of the conditional expectation:

$$\mathbb{E}[h(Y)\, g(X,Y) \mid Y] = h(Y)\, \mathbb{E}[g(X,Y) \mid Y].$$

Similarly, the spread of X within a strip is the *conditional variance* $\operatorname{Var}(X \mid Y) = \mathbb{E}[X^2 \mid Y] - (\mathbb{E}[X \mid Y])^2$, which is just the computational variance identity of Section 36.1 applied inside each strip.

Intuitively, averaging the strip averages should recover the overall average, and the overall variance should split into the spread within strips plus the spread between them. Both statements hold, and the second turns out to be especially useful.

Proposition (laws of total expectation and variance). *Let X, Y have a joint density, and let X have finite variance (for discrete pairs, the same identities hold with integrals replaced by sums). Then*

$$\begin{aligned} \mathbb{E}[X] &= \mathbb{E}\big[\mathbb{E}[X \mid Y]\big], \\ \operatorname{Var}(X) &= \mathbb{E}\big[\operatorname{Var}(X \mid Y)\big] + \operatorname{Var}\big(\mathbb{E}[X \mid Y]\big). \end{aligned} \tag{36.22}$$

Proof. For the first identity, known as the *tower property*, write out the inner expectation and apply the chain rule $p_{X|Y}(x \mid y)\, p_Y(y) = p_{X,Y}(x,y)$ to reassemble the strips into the joint density:

$$\mathbb{E}[\mathbb{E}[X \mid Y]] = \int \Big(\int x\, p_{X|Y}(x \mid y)\, dx \Big) p_Y(y)\, dy = \iint x\, p_{X,Y}(x,y)\, dx\, dy = \mathbb{E}[X].$$

Values of y with $p_Y(y) = 0$, where the conditional density is undefined, contribute to neither side because Y lands there with probability zero. The same computation with $g(x,y)$ in place of x gives $\mathbb{E}\big[\mathbb{E}[g(X,Y) \mid Y]\big] = \mathbb{E}[g(X,Y)]$. For the second identity, expand both terms on the right. Applying the tower property to X^2 gives

$$\begin{aligned} \mathbb{E}\big[\operatorname{Var}(X \mid Y)\big] &= \mathbb{E}\big[\mathbb{E}[X^2 \mid Y]\big] - \mathbb{E}\big[(\mathbb{E}[X \mid Y])^2\big] \\ &= \mathbb{E}[X^2] - \mathbb{E}\big[(\mathbb{E}[X \mid Y])^2\big]. \end{aligned}$$

Applying $\mathrm{Var}(Z) = \mathbb{E}[Z^2] - (\mathbb{E}[Z])^2$ (Section 36.1) to the random variable $\mathbb{E}[X \mid Y]$, whose mean is $\mathbb{E}[X]$ by the tower property, gives

$$\mathrm{Var}\big(\mathbb{E}[X \mid Y]\big) = \mathbb{E}\big[(\mathbb{E}[X \mid Y])^2\big] - \mathbb{E}[X]^2.$$

Adding the two lines, the middle term $\mathbb{E}[(\mathbb{E}[X \mid Y])^2]$ cancels and leaves $\mathbb{E}[X^2] - \mathbb{E}[X]^2 = \mathrm{Var}(X)$. ■

The second identity is often called *Eve's law*, after the pattern E, V, V, E on its right-hand side. It splits the total variance of X into two pieces: the average variance within each strip, which knowing Y can never remove, and the variance of the strip averages themselves, which is the portion that Y explains. Statisticians use this decomposition to attribute variation in data to groups under the name analysis of variance. As we now show, the same decomposition also settles our regression question.

Proposition (the conditional mean is the best squared-error predictor). *Let X have finite variance, and let h be any function with $\mathbb{E}[h(Y)^2] < \infty$. Then*

$$\mathbb{E}\big[(X - h(Y))^2\big] = \mathbb{E}\big[\mathrm{Var}(X \mid Y)\big] + \mathbb{E}\big[(\mathbb{E}[X \mid Y] - h(Y))^2\big]. \tag{36.23}$$

Proof. Write $m(Y) = \mathbb{E}[X \mid Y]$ and split the prediction error into two parts, $X - h(Y) = (X - m(Y)) + (m(Y) - h(Y))$. Squaring produces two squared terms and a cross term, and only the cross term requires care. The expectation of the first square is $\mathbb{E}[\mathrm{Var}(X \mid Y)]$ by the tower property. The cross term $2(X - m(Y))(m(Y) - h(Y))$ has conditional mean zero: the factor $m(Y) - h(Y)$ depends on Y alone and pulls out of the strip average, leaving $\mathbb{E}[X - m(Y) \mid Y] = m(Y) - m(Y) = 0$. By the tower property, its overall mean is zero as well. ■

Look at the two terms on the right. The first does not depend on h at all, while the second is non-negative and vanishes when $h(Y) = \mathbb{E}[X \mid Y]$. Thus the conditional mean is the optimal predictor under squared error, and no predictor can achieve a lower expected squared error than the average within-strip variance. A network trained with squared loss is therefore estimating $\mathbb{E}[\text{target} \mid \text{input}]$, which is how Section 2.1 interprets its model (written there as $\mathbb{E}[y \mid \mathbf{x}]$).

Two examples make this decomposition concrete. First, return to the dart and try to predict X from Y. Given $Y = y$, the horizontal coordinate X is uniform on a chord symmetric about 0, so $\mathbb{E}[X \mid Y = y] = 0$. A uniform variable on an interval of half-width w has variance $w^2/3$, which gives $\mathrm{Var}(X \mid Y = y) = (1 - y^2)/3$. By symmetry $\mathbb{E}[X^2] = \mathbb{E}[Y^2] = \frac{1}{2}\mathbb{E}[R^2] = \frac{1}{4}$, so Eve's law yields

$$\mathrm{Var}(X) = \mathbb{E}\Big[\frac{1 - Y^2}{3}\Big] + \mathrm{Var}(0) = \frac{1 - 1/4}{3} + 0 = \frac{1}{4}.$$

All of the variance stays in the unexplained term. Knowing Y narrows the range of X without ever shifting its center, so the best squared-error predictor of X from Y is the constant 0, even though X and Y are dependent.

For a second example, draw a binary label Z that equals 1 with probability 0.2 and 0 with probability 0.8. Given $Z = 1$, draw X from the normal distribution $\mathcal{N}(3, 1)$ with mean

3 and variance 1; given $Z = 0$, draw X from $\mathcal{N}(-1, 1)$. This mixture produces the two-bump density of Figure 36.1. Its overall mean is $0.2 \cdot 3 + 0.8 \cdot (-1) = -0.2$. Within each component the variance is 1, so $\mathbb{E}[\mathrm{Var}(X \mid Z)] = 1$. Between the components, $\mathbb{E}[X \mid Z]$ takes the values 3 and -1, whose variance is $0.2 \cdot 0.8 \cdot (3 - (-1))^2 = 2.56$. The total variance is therefore 3.56, and the label Z explains $2.56/3.56 = 72\%$ of it. In a sample of one million draws, the empirical within- and between-component terms will land near 1 and 2.56. Before running the code, ask yourself: will their sum match the sample variance only roughly, or exactly?

```
n = 10**6
z = rng.uniform(size=n) < 0.2                          # component label Z
x = np.where(z, rng.normal(3, 1, n), rng.normal(-1, 1, n))
w = np.array([np.mean(~z), np.mean(z)])                # P(Z=0), P(Z=1)
cond_mean = np.array([x[~z].mean(), x[z].mean()])      # E[X | Z]
within = w @ np.array([x[~z].var(), x[z].var()])       # E[Var(X | Z)]
between = w @ (cond_mean - x.mean())**2                # Var(E[X | Z])
print(f'Var(X) {x.var():.3f} = within {within:.3f} + between {between:.3f}')
```

```
Var(X) 3.562 = within 1.000 + between 2.562
```

Exactly. Each piece lands close to its theoretical value (1 and 2.56), with sampling noise in the third decimal place, yet their sum equals the sample variance in every run up to rounding in the last printed digit. Eve's law is an exact identity for the sample itself, with the empirical group fractions and group variances playing the roles of probabilities and conditional variances.

36.2.2 Covariance and Correlation

The conditional mean describes how the average of X moves with Y, which makes it an entire function. Often we prefer a single summary number: do X and Y tend to be large together, or does one tend to be large when the other is small? The covariance is that number. It has two limitations that the rest of this section addresses or exposes: it carries the units of both variables, which the correlation removes, and it detects only straight-line trends, which no rescaling can fix.

36.2.2.1 Covariance

Section 1.6 defined the *covariance* as the expected product of the two deviations from their means. For discrete X, Y taking (x_i, y_j) with probability p_{ij}, it is

$$\sigma_{XY} = \mathrm{Cov}(X, Y) = \sum_{i,j} (x_i - \mu_X)(y_j - \mu_Y)\, p_{ij} = \mathbb{E}[XY] - \mathbb{E}[X]\,\mathbb{E}[Y], \tag{36.24}$$

where the second form follows by expanding the product, just as in the computational form of the variance. In particular, $\mathrm{Cov}(X, X) = \mathrm{Var}(X)$. Throughout this section we assume both variances are finite; since $|xy| \le (x^2 + y^2)/2$, the expectation of XY is then guaranteed to exist.

As a concrete example, let $X \in \{1, 3\}$ and $Y \in \{-1, 3\}$ with joint probabilities

$$P(X{=}1, Y{=}{-1}) = \tfrac{\theta}{2}, \qquad P(X{=}1, Y{=}3) = \tfrac{1-\theta}{2},$$
$$P(X{=}3, Y{=}{-1}) = \tfrac{1-\theta}{2}, \qquad P(X{=}3, Y{=}3) = \tfrac{\theta}{2},$$

for a parameter $\theta \in [0, 1]$. We will refer to this as the *four-cell example*. Each variable takes its two values with probability $\frac{1}{2}$, so $\mu_X = 2$ and $\mu_Y = 1$. The expectation of the product is

$$\mathbb{E}[XY] = \tfrac{1}{2}(-\theta + 3(1-\theta) - 3(1-\theta) + 9\theta) = 4\theta,$$

which gives $\mathrm{Cov}(X, Y) = 4\theta - 2$: it is $+2$ at $\theta = 1$, -2 at $\theta = 0$, and 0 at $\theta = \frac{1}{2}$.

For continuous variables with a joint density, the sum becomes a double integral,

$$\sigma_{XY} = \iint (x - \mu_X)(y - \mu_Y)\, p_{X,Y}(x, y)\, dx\, dy, \tag{36.25}$$

averaging the product of deviations over the joint density. For our dart, both coordinates have mean 0, and $\mathrm{Cov}(X, Y) = \mathbb{E}[XY] = \frac{1}{\pi}\iint_{\text{disk}} xy\, dx\, dy = 0$, because the integrand is odd in x while the disk is symmetric. Yet X and Y are dependent, as the shrinking chords showed. Covariance looks only for a linear trend, and the dart's circular dependence has none, so zero covariance does not imply independence (Section 1.6 gives a discrete example, $Y = X^2$). Figure 36.9 shows both facts on the disk.

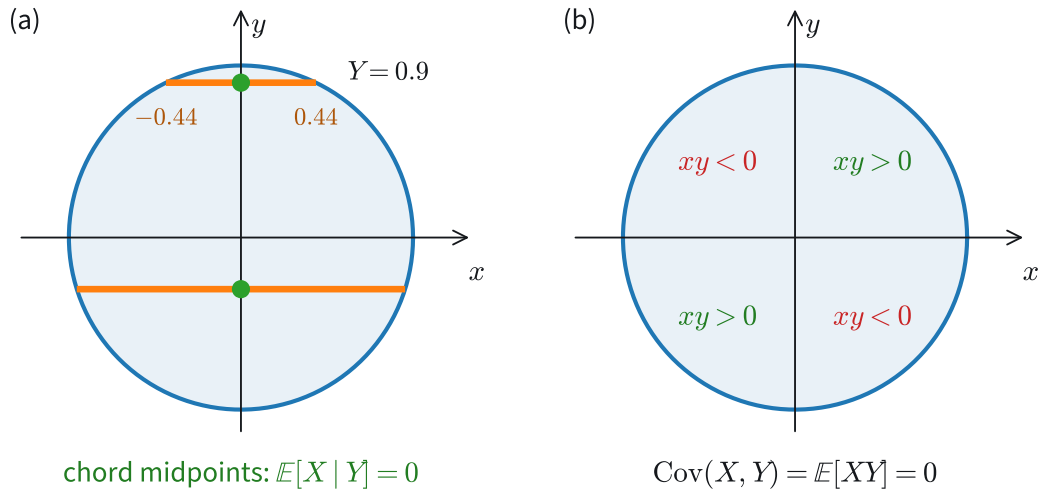


Figure 36.9: The dart's coordinates are dependent, yet unpredictable in mean and uncorrelated. (a) The chord at height $Y = 0.9$ confines X to $|X| \le 0.44$, while a chord near the middle is far longer, so X depends on Y. Every chord is centered on $x = 0$ (green dots), so $\mathbb{E}[X \mid Y] = 0$. (b) The product xy is positive on two quadrants and negative on the other two, all of equal area, so $\mathrm{Cov}(X, Y) = 0$.

Covariance also carries the units of both variables. In Figure 36.10, the bottom row plots the exact same draws as the top row, but with Y restated in cents instead of dollars. Every sample covariance jumps by a factor of 100, even though none of the point clouds changes

shape. Without the units, only the sign of a covariance is meaningful; the correlation coefficient below removes the units altogether.

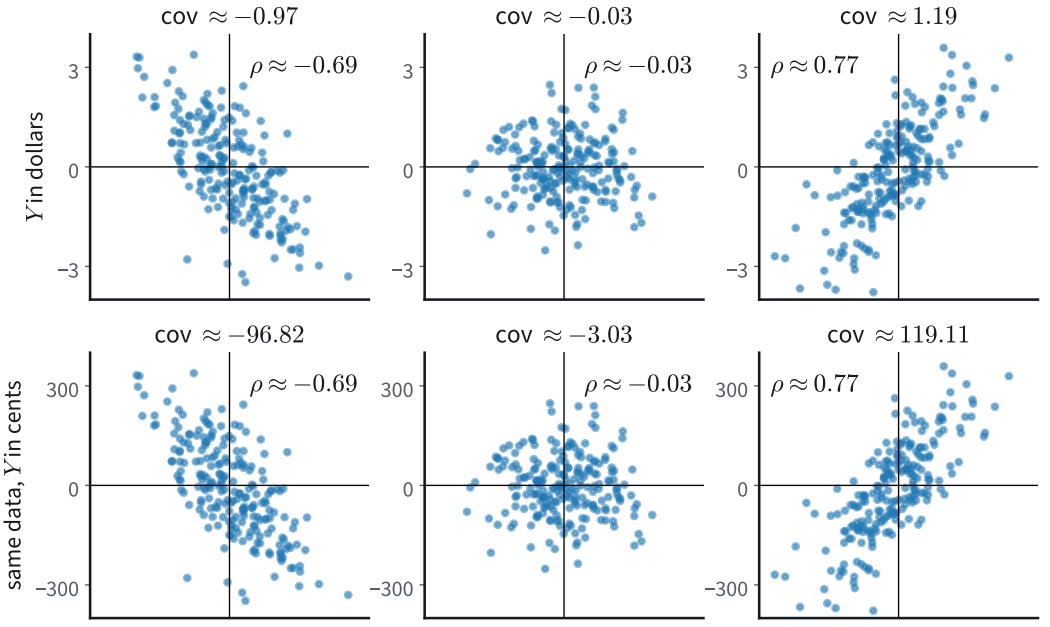


Figure 36.10: Top row: scatter clouds of (X, Y) pairs with covariance tuned negative, zero, and positive (left to right), Y in dollars: negative covariance tilts the cloud down-right, zero leaves it round, positive tilts it up-right. Bottom row: the identical draws with Y restated in cents. Every sample covariance is multiplied by 100 while each panel's correlation ρ (in its top corner) is unchanged: the magnitude of covariance carries units, only its sign is scale-free.

Covariance is linear in each argument. Adding a constant shifts both the variable and its mean by the same amount, so shifts drop out of the deviations, while multiplicative factors pull straight out: $\mathrm{Cov}(cX + d, Y) = c\,\mathrm{Cov}(X, Y)$. Likewise, the deviation of a sum is the sum of the deviations, so $\mathrm{Cov}(X + Z, Y) = \mathrm{Cov}(X, Y) + \mathrm{Cov}(Z, Y)$ by linearity of expectation, and the same rules hold in the second slot. Moreover, independent variables always have zero covariance: whenever the joint density factorizes, the double integral for $\mathbb{E}[XY]$ splits into $\mathbb{E}[X]\,\mathbb{E}[Y]$.

Covariance also completes the rule for the variance of a sum. Write $\tilde{X} = X - \mu_X$ and $\tilde{Y} = Y - \mu_Y$. By linearity, $X + Y$ has mean $\mu_X + \mu_Y$ and therefore deviation $\tilde{X} + \tilde{Y}$. Expanding $\mathbb{E}[(\tilde{X} + \tilde{Y})^2] = \mathbb{E}[\tilde{X}^2] + 2\,\mathbb{E}[\tilde{X}\tilde{Y}] + \mathbb{E}[\tilde{Y}^2]$ yields

$$\mathrm{Var}(X + Y) = \mathrm{Var}(X) + \mathrm{Var}(Y) + 2\,\mathrm{Cov}(X, Y). \tag{36.26}$$

When X and Y are independent, the covariance vanishes and the variances simply add.

The most important consequence of this rule is what happens when we average many draws. Expanding the square of a sum of n deviations in the same way gives $\mathrm{Var}(\sum_i X_i) =$

$\sum_i \sum_j \mathrm{Cov}(X_i, X_j)$. Suppose $X_1, \dots, X_n$ are independent with common variance σ^2. Every cross term vanishes, so the sum has variance $n\sigma^2$, and scaling by $c = 1/n$ divides that variance by n^2. The average $\bar{X}_n = \frac{1}{n}\sum_{i=1}^{n} X_i$ therefore has variance

$$\mathrm{Var}(\bar{X}_n) = \frac{1}{n^2} \cdot n\sigma^2 = \frac{\sigma^2}{n}. \tag{36.27}$$

Averaging n independent draws divides the variance by n and the standard deviation by $\sqrt{n}$: to halve the typical error, you need four times as many draws. This $\sigma/\sqrt{n}$ scaling underpins every *Monte Carlo* estimate, in which an average of simulated samples stands in for an expectation (Section 36.5.1.7). For our dart, the average distance of n independent throws has variance $1/(18n)$, so 556 darts suffice to bring its standard deviation down to 0.01. Notice that the $\sigma/\sqrt{n}$ rate requires a finite variance. The *law of large numbers* itself, which states that $\bar{X}_n$ converges to the mean, requires only a finite mean (Section 36.5). The Cauchy distribution has neither, and averages of its draws never settle down no matter how large n becomes.

36.2.2.2 Correlation

Whether the covariance $4\theta - 2$ in our four-cell example counts as large depends on the units of X and Y. To strip away the units, we divide by the two standard deviations, which carry the exact same units. The *correlation coefficient* is

$$\rho(X, Y) = \frac{\mathrm{Cov}(X, Y)}{\sigma_X \sigma_Y}. \tag{36.28}$$

Section 1.6 stated that ρ always lies between -1 and 1. Let us prove that bound and see exactly when equality holds.

Proposition (correlation bound). *For any X, Y with finite, nonzero variances, $-1 \le \rho(X, Y) \le 1$. Equality holds if and only if $Y = cX + d$ with probability one for constants $c \neq 0$ and d, with $\rho = 1$ when $c > 0$ and $\rho = -1$ when $c < 0$.*

Proof. A variance is never negative, so $\mathrm{Var}(tX + Y) \ge 0$ for every real t. Applying the variance-of-a-sum rule and the affine scaling rule turns this into a quadratic in t,

$$\mathrm{Var}(tX + Y) = \mathrm{Var}(X)\, t^2 + 2\,\mathrm{Cov}(X, Y)\, t + \mathrm{Var}(Y) \;\ge\; 0 \quad \text{for every } t.$$

A quadratic with a positive leading coefficient that never dips below zero can have at most one real root, so its discriminant must be non-positive:

$$\big(2\,\mathrm{Cov}(X, Y)\big)^2 - 4\,\mathrm{Var}(X)\,\mathrm{Var}(Y) \le 0, \qquad \text{that is} \qquad \mathrm{Cov}(X, Y)^2 \le \mathrm{Var}(X)\,\mathrm{Var}(Y).$$

Dividing both sides by $\sigma_X^2 \sigma_Y^2$ gives $\rho^2 \le 1$. If $\rho^2 = 1$, the discriminant vanishes and the quadratic has a double root $t^\star$ satisfying $\mathrm{Var}(t^\star X + Y) = 0$. A random variable with zero variance is constant with probability one, so $Y = -t^\star X + d$ with probability one, and $t^\star \neq 0$ because $\mathrm{Var}(Y) > 0$. Conversely, if $Y = cX + d$ with $c \neq 0$, then $\mathrm{Cov}(X, Y) = c\,\mathrm{Var}(X)$ and $\sigma_Y = |c|\sigma_X$, giving $\rho = c/|c| = \pm 1$. $\blacksquare$

The correlation measures how tightly X and Y hug a straight line, together with the sign of that line's slope, but it ignores the magnitude of the slope: $Y = 0.5X$ and $Y = 2X$ both have $\rho = 1$. In the four-cell example, $\sigma_X = 1$ and $\sigma_Y = 2$, so $\rho = (4\theta - 2)/2 = 2\theta - 1$, sweeping from -1 to 1 as θ goes from 0 to 1. Figure 36.11 highlights two blind spots of ρ: two point clouds with the same correlation but different slopes, and a cloud lying on a parabola whose correlation is zero.

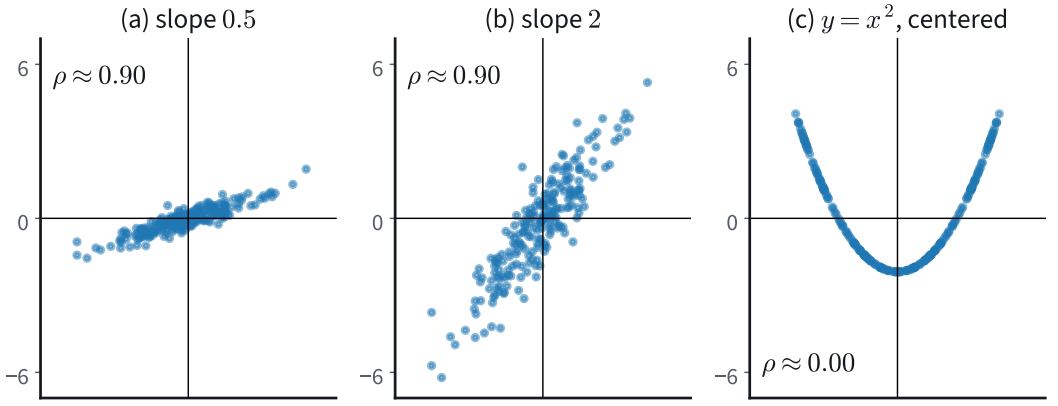


Figure 36.11: Correlation measures the tightness and sign of a linear relation, not its slope: (a) and (b) have the same $\rho \approx 0.9$ at slopes 0.5 and 2. It also misses a nonlinear relation: in (c) Y is a function of X and $\rho \approx 0$.

Notice that the proof above is the classic proof of the Cauchy–Schwarz inequality $|\mathbf{u}^\top \mathbf{v}| \le \|\mathbf{u}\|\,\|\mathbf{v}\|$, and this is no coincidence: correlation is literally a cosine. On the space of mean-zero random variables with finite variance, the expected product acts as an inner product (treating two variables as identical when they agree with probability one). For the centered variables $\tilde{X}$ and $\tilde{Y}$, this inner product is $\langle \tilde{X}, \tilde{Y} \rangle = \mathbb{E}[\tilde{X}\tilde{Y}] = \mathrm{Cov}(X, Y)$, the induced length is the standard deviation $\|\tilde{X}\| = \sqrt{\mathbb{E}[\tilde{X}^2]} = \sigma_X$, and the correlation is the cosine $\langle \tilde{X}, \tilde{Y} \rangle / (\|\tilde{X}\|\,\|\tilde{Y}\|)$. Geometrically, uncorrelated variables are orthogonal, and $\rho = \pm 1$ means they are parallel.

36.2.2.3 The Covariance Matrix

When we track n variables at once, there is a covariance for every pair, and the natural way to organize them is in a matrix. For a random vector $\mathbf{X} = (X_1, \dots, X_n)^\top$ with mean μ, the *covariance matrix* is the expected outer product of the deviation with itself,

$$\Sigma = \mathbb{E}\big[(\mathbf{X} - \mu)(\mathbf{X} - \mu)^\top\big], \qquad \Sigma_{ij} = \mathrm{Cov}(X_i, X_j). \tag{36.29}$$

Its diagonal holds the individual variances, while the off-diagonal entries hold the pairwise covariances. Because $\mathrm{Cov}(X_i, X_j) = \mathrm{Cov}(X_j, X_i)$, the matrix is symmetric. It is also positive semidefinite: for any fixed vector $\mathbf{a}$, the scalar projection $\mathbf{a}^\top \mathbf{X} = \sum_i a_i X_i$ has deviation $\mathbf{a}^\top (\mathbf{X} - \mu)$, so

$$\mathrm{Var}(\mathbf{a}^\top \mathbf{X}) = \mathbb{E}\big[\mathbf{a}^\top (\mathbf{X} - \mu)(\mathbf{X} - \mu)^\top \mathbf{a}\big] = \mathbf{a}^\top \Sigma\, \mathbf{a} \;\ge\; 0.$$

In words, the quadratic form $\mathbf{a}^\top \Sigma\, \mathbf{a}$ is simply the variance of the data projected onto the direction $\mathbf{a}$, and a variance can never be negative. For the dart, $\Sigma = \mathrm{diag}(\frac{1}{4}, \frac{1}{4})$: its coordinates are uncorrelated and equally spread. Later in the book, the eigenvectors of Σ appear as the directions of principal component analysis (Section 33.5) and as the axes of the multivariate Gaussian's elliptical contours (Section 36.3).

36.2.3 Transformations of Random Variables

Machine learning transforms random variables constantly: an activation function acts on a pre-activation, a sampler reshapes uniform noise, and a normalizing flow warps a simple Gaussian into a complex data distribution. If X has a known density and we set $Y = g(X)$, what is the density of Y?

36.2.3.1 The One-Dimensional Rule

The first guess you might try, $p_Y(y) = p_X(g^{-1}(y))$, is wrong, and our dart already shows why. Its distance can be generated as $R = \sqrt{U}$ with U uniform on $[0, 1]$, yet this map turns the flat density of U into the slanted density $2r$. The reason is that probability mass, not density height, is conserved under a transformation. A narrow interval of width Δx around x carries probability about $p_X(x)\,\Delta x$, and g sends it to an interval of width about $\Delta y = |g'(x)|\,\Delta x$ around $y = g(x)$, which must carry the same probability:

$$p_Y(y)\,|g'(x)|\,\Delta x \approx p_X(x)\,\Delta x.$$

Wherever g stretches the line, the same probability is spread over a wider interval, so the density must thin out by the stretch factor $|g'(x)|$; wherever g compresses the line, the density must rise. Figure 36.12 illustrates these two equal-area strips for $Y = e^X$.

Proposition (change of variables in one dimension). *Let X have a density p_X that is continuous on an open interval I with $P(X \in I) = 1$, and let $g : I \to \mathbb{R}$ be continuously differentiable with $g' \neq 0$ on I, so that g is strictly monotone and one-to-one. Then $Y = g(X)$ has the density*

$$p_Y(y) = p_X(g^{-1}(y)) \left| \frac{dg^{-1}}{dy}(y) \right| \tag{36.30}$$

on the interval $g(I)$, and 0 outside it.

Proof. Suppose first that g is increasing. Then $g(X) \le y$ holds precisely when $X \le g^{-1}(y)$, so the c.d.f.s satisfy $F_Y(y) = F_X(g^{-1}(y))$ for $y \in g(I)$. Differentiating via the chain rule and $F_X' = p_X$ (Section 36.1) gives $p_Y(y) = p_X(g^{-1}(y))\,\frac{dg^{-1}}{dy}(y)$. If g is decreasing instead, then $g(X) \le y$ holds when $X \ge g^{-1}(y)$, so $F_Y(y) = 1 - F_X(g^{-1}(y))$. Its derivative introduces a minus sign, which cancels the negative sign of dg^{-1}/dy; taking the absolute value covers both cases at once. In either case, F_Y is continuously differentiable on $g(I)$, which carries all of the probability, so by the fundamental theorem of calculus its derivative is a density of Y. ∎

Read in words, the new density at y is the old density evaluated at the point that maps to y, multiplied by the local compression factor of the inverse map. Since $dg^{-1}/dy = 1/g'(x)$ at $x = g^{-1}(y)$, this is the exact same $1/|g'(x)|$ factor we obtained from the stretching argument.

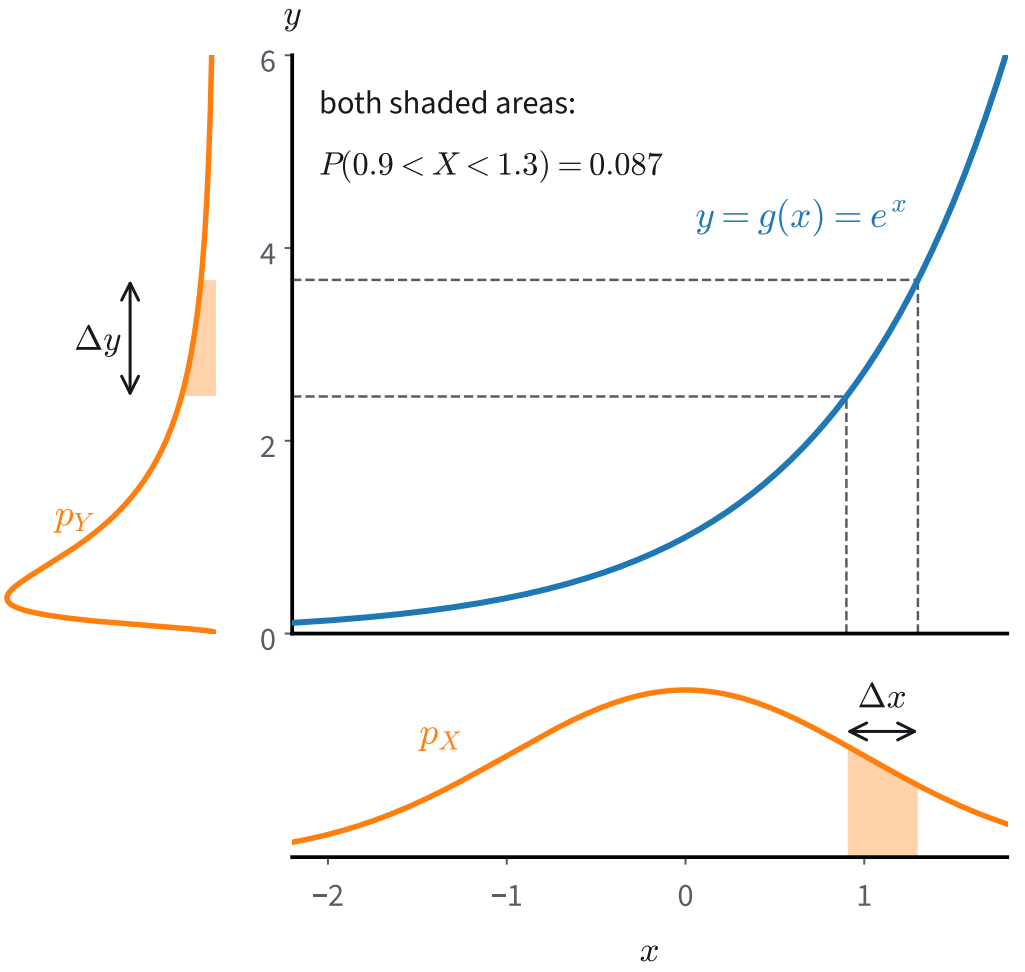


Figure 36.12: Change of variables. An interval of width Δx under p_X (bottom) is mapped by g to an interval of width $\Delta y = g'(x)\,\Delta x$ under p_Y (left). The two shaded areas are the same probability, so where g stretches ($\Delta y > \Delta x$) the density p_Y must be lower by the factor $|g'(x)|$. Here $X \sim \mathcal{N}(0, 1)$ and $Y = e^X$.

Inverse-transform sampling is a change of variables, and the rule immediately recovers the dart's distance density from its sampler. Take $X = U$ uniform on $I = (0, 1)$ and $g(u) = \sqrt{u}$, so $g^{-1}(r) = r^2$ and $|dg^{-1}/dr| = 2r$. Equation Equation 36.30 gives $p_R(r) = 1 \cdot 2r$, the density of the dart's distance.

When g is not one-to-one, several input points can map to the same output y, and their probability contributions simply add:

$$p_Y(y) = \sum_{x:\, g(x)=y} \frac{p_X(x)}{|g'(x)|},$$

for values y with finitely many preimages, each satisfying $g'(x) \neq 0$. For $Y = X^2$ and $y > 0$, the two preimages $\pm\sqrt{y}$ both have $|g'| = 2\sqrt{y}$, so $p_Y(y) = [p_X(\sqrt{y}) + p_X(-\sqrt{y})]/(2\sqrt{y})$. The hypothesis $g' \neq 0$ really matters. The ReLU activation has $g' = 0$ on the entire negative half-line $(-\infty, 0)$, collapsing all of that mass onto the single point 0 and producing a point atom rather than a pure density (Exercise 2).

As a worked example, let $X \sim \mathcal{N}(0, 1)$ have the standard normal density $p_X(x) = e^{-x^2/2}/\sqrt{2\pi}$, and set $Y = e^X$. Here $g^{-1}(y) = \log y$ and $|dg^{-1}/dy| = 1/y$ for $y > 0$, so the change-of-variables rule produces the *log-normal* density

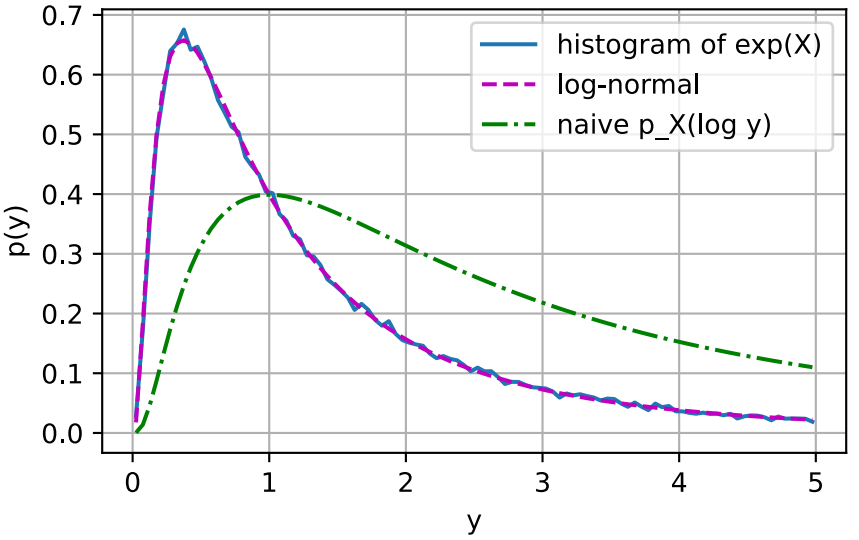
$$p_Y(y) = \frac{1}{y\sqrt{2\pi}} \exp\Big(-\tfrac{1}{2}(\log y)^2 \Big), \qquad y > 0.$$

The leading factor $1/y$ is the change-of-variables correction. Setting the derivative of $\log p_Y(y) = -\log y - \frac{1}{2}(\log y)^2 + \text{const}$ to zero gives $\log y = -1$, so the true density peaks at $y = e^{-1} \approx 0.368$. By contrast, the naive guess $p_X(\log y)$ peaks at $y = 1$, and it is not even a valid density: substituting $y = e^x$, its total area is $\int p_X(x)\, e^x\, dx = \mathbb{E}[e^X] = e^{1/2} \approx 1.65$ (completing the square gives $x - x^2/2 = \frac{1}{2} - \frac{1}{2}(x-1)^2$).

The same calculation gives all moments $\mathbb{E}[Y^k] = \mathbb{E}[e^{kX}] = e^{k^2/2}$, every one of them finite. Even so, the upper tail $P(Y > t) = P(X > \log t)$ decays more slowly than any exponential in t. The log-normal distribution is therefore heavy-tailed, yet unlike the Cauchy distribution it still has finite moments of every order.

Let us plot a histogram of 10^5 samples of e^X alongside both candidate curves. We already know the naive curve encloses 0.65 too much area; before looking at the figure, try guessing where that excess sits.

```
y = np.exp(rng.standard_normal(10**5))                # Y = e^X
edges = np.linspace(0, 5, 101)
mids = (edges[:-1] + edges[1:]) / 2
hist = np.histogram(y, edges)[0] / (len(y) * (edges[1] - edges[0]))
p_X = lambda x: np.exp(-x**2 / 2) / np.sqrt(2 * np.pi)
d2l.plot(mids, [hist, p_X(np.log(mids)) / mids, p_X(np.log(mids))],
         'y', 'p(y)', figsize=(5, 3),
         legend=['histogram of exp(X)', 'log-normal', 'naive p_X(log y)'])
```



The histogram tracks the log-normal density closely up to sampling noise, with its peak well to the left of $y = 1$, near e^{-1}. The naive curve puts the peak in the wrong place: because it equals the true density multiplied by y, it falls below the data for $y < 1$ and rises above it for $y > 1$, by a factor that keeps growing into the tail.

36.2.3.2 The Multivariate Rule

In several dimensions, a transformation stretches areas and volumes rather than lengths, so the stretch factor becomes a determinant. Start with the simplest case, a linear map $\mathbf{Y} = \mathbf{A}\mathbf{X}$ with $\mathbf{A} = \operatorname{diag}(2, 3)$. This map turns a unit square into a 2×3 rectangle of area $6 = \det \mathbf{A}$. Because the rectangle must carry the same probability as the original square, the density of $\mathbf{Y}$ must be one-sixth the density of $\mathbf{X}$ at the corresponding point: $p_{\mathbf{Y}}(\mathbf{y}) = p_{\mathbf{X}}(\mathbf{A}^{-1}\mathbf{y})/6$.

For a nonlinear map, the local stretch varies from point to point, and the change-of-variables theorem for multiple integrals accounts for it (Section 34.6). Let g be a continuously differentiable bijection between two regions of $\mathbb{R}^n$ with a continuously differentiable inverse, and write $J_{g^{-1}}$ for the Jacobian matrix of partial derivatives of the inverse. Locally, g^{-1} scales volume by $|\det J_{g^{-1}}|$, so substituting $\mathbf{x} = g^{-1}(\mathbf{y})$ in an integral over the preimage of a region $\mathcal{D}$ gives

$$\int_{g^{-1}(\mathcal{D})} f(\mathbf{x})\, d\mathbf{x} = \int_{\mathcal{D}} f(g^{-1}(\mathbf{y}))\, \left|\det J_{g^{-1}}(\mathbf{y})\right|\, d\mathbf{y}.$$

Now let $\mathbf{Y} = g(\mathbf{X})$. The event $\mathbf{Y} \in \mathcal{D}$ is the same as the event $\mathbf{X} \in g^{-1}(\mathcal{D})$, so applying the theorem with $f = p_{\mathbf{X}}$ yields

$$P(\mathbf{Y} \in \mathcal{D}) = \int_{g^{-1}(\mathcal{D})} p_{\mathbf{X}}(\mathbf{x})\, d\mathbf{x} = \int_{\mathcal{D}} p_{\mathbf{X}}(g^{-1}(\mathbf{y}))\, \left|\det J_{g^{-1}}(\mathbf{y})\right|\, d\mathbf{y}.$$

The integrand on the right is therefore the density of $\mathbf{Y}$:

$$\begin{aligned} p_{\mathbf{Y}}(\mathbf{y}) &= p_{\mathbf{X}}\big(g^{-1}(\mathbf{y})\big)\,\big|\det J_{g^{-1}}(\mathbf{y})\big|, \\ \log p_{\mathbf{Y}}(\mathbf{y}) &= \log p_{\mathbf{X}}(\mathbf{x}) - \log\big|\det J_{g}(\mathbf{x})\big| \\ &\qquad \text{(equivalently, at } \mathbf{x} = g^{-1}(\mathbf{y})\text{)}. \end{aligned} \tag{36.31}$$

The second line uses $J_{g^{-1}}(\mathbf{y}) = J_g(\mathbf{x})^{-1}$. Passing a sample through g simply subtracts the log of the local volume stretch from its log-density. Because $\log|\det(J_2 J_1)| = \log|\det J_1| + \log|\det J_2|$, a composition of maps subtracts the sum of their log-determinants. This is the exact mechanism by which *normalizing flows* compute exact likelihoods: they stack invertible layers whose log-determinants are cheap to evaluate (Rezende and Mohamed 2015), and Section 38.3.3 builds one whose layers are the steps of a differential equation. Exercise 5 implements a coupling layer from scratch.

Our dart provides one last illustration. Describe its position in polar coordinates, the distance R and the angle Θ, via $(x, y) = (r\cos\theta, r\sin\theta)$. The Jacobian determinant of this map from (r, θ) to (x, y) is $\cos\theta \cdot r\cos\theta - (-r\sin\theta)\sin\theta = r$, so the multivariate rule yields

$$p_{R,\Theta}(r, \theta) = \frac{1}{\pi} \cdot r = 2r \cdot \frac{1}{2\pi}, \qquad 0 \le r \le 1,\ 0 \le \theta < 2\pi.$$

The joint density factorizes cleanly. The angle Θ is uniform on $[0, 2\pi)$, the distance R recovers the density $2r$ once more (now straight from a Jacobian), and the two variables are independent. In Cartesian coordinates, the dart's X and Y were dependent; in the right coordinate system, the exact same dart is described by two independent variables.

This section leaves one central question about averages open. We now know that the average of n independent draws has variance σ^2/n Equation 36.27, but we do not yet know its distribution. Chebyshev's inequality (Section 36.1) turns that variance into a bound of $\sigma^2/(n\epsilon^2)$ on the probability that the average errs by ϵ or more, yet for bounded or Gaussian draws this bound is far too pessimistic: the true error probability drops exponentially fast in n. We will determine which distribution an average approaches in Section 36.3.2.3, and establish the exponential decay rate in Section 36.9. Before tackling either, however, we need a richer catalog of distributions than the dart alone.

36.2.4 Summary

Marginal densities come from integrating out unneeded variables, and conditional densities come from slicing the joint density along a thin strip and renormalizing. The conditional mean is the optimal squared-error predictor, and the law of total variance splits a variable's variance into the part that the conditioning variable explains and the part it cannot. Covariance and correlation measure linear association rather than general dependence, and averaging n independent draws with finite variance reduces the variance to σ^2/n. Under a smooth invertible map, a density is divided by the local volume stretch, given by the absolute Jacobian determinant. Our dart brings all of these ideas together: its Cartesian coordinates are dependent yet uncorrelated, while in polar coordinates they become two independent variables.

36.2.5 Exercises

1. **Uncorrelated but dependent.** Whitening, for example with the principal components of Section 33.5, makes coordinates uncorrelated. The following pair has Gaussian coordinates and zero covariance. Let $X \sim \mathcal{N}(0,1)$, let S be a random sign independent of X with $P(S=1) = P(S=-1) = \frac{1}{2}$, and let $Y = SX$.
   1. Show that $Y \sim \mathcal{N}(0,1)$ by computing its c.d.f., and that $\mathrm{Cov}(X,Y) = 0$.
   2. Compare $P(|X| \le 1, |Y| \le 1)$ with $P(|X| \le 1)\,P(|Y| \le 1)$. Are X and Y independent?
   3. Show that $X+Y$ has an atom at 0, so that $X+Y$ is not Gaussian. With probability one the point (X,Y) lies on one of the two lines $y = \pm x$, and a double integral over a line is zero, because a line has no area. Conclude that the pair (X,Y) has no joint density, although each coordinate has one.

2. **Averaging correlated estimates.** The text derives $\mathrm{Var}(\bar{X}_n) = \sigma^2/n$ for independent draws. Averages of dependent estimates behave differently.
   1. Let $Y_1, \dots, Y_n$ each have variance σ^2, with correlation ρ between every pair, as the predictions of n ensemble members trained on the same data might. Show that
   $$\mathrm{Var}\Big(\frac{1}{n}\sum_{i=1}^{n} Y_i\Big) = \sigma^2\Big(\rho + \frac{1-\rho}{n}\Big). \tag{36.32}$$
   What does averaging achieve as $n \to \infty$?
   2. Use $\mathbf{a}^\top \Sigma\, \mathbf{a} \ge 0$ for the covariance matrix Σ of $Y_1, \dots, Y_n$ to show that $\rho \ge -1/(n-1)$. Give variables that attain this bound.

3. code **Sigmoid of a Gaussian.** Let $X \sim \mathcal{N}(0, s^2)$ be the pre-activation of a sigmoid unit, and let $Y = \mathrm{sigmoid}(X)$, whose inverse is $\mathrm{logit}(y) = \log(y/(1-y))$.
   1. Use Equation 36.30 to find the density p_Y on $(0,1)$.
   2. For which s is $y = \frac{1}{2}$ a local minimum of p_Y? Writing $\log p_Y$ as a function of $z = \mathrm{logit}(y)$ simplifies the computation.
   3. Modify the `#random-variables-change-of-variables` cell to compare a histogram of Y with p_Y for $s \in \{1, \sqrt{2}, 3\}$. What does the histogram at $s = 3$ imply for the derivative $\mathrm{sigmoid}'(X) = Y(1-Y)$, through which backpropagation passes the gradient of these units?

4. **Explained variance.** ● A regression model h predicts a target X from an input Y. The fraction of the variance that it explains is $R^2 = 1 - \mathbb{E}\big[(X - h(Y))^2\big]/\mathrm{Var}(X)$.
   1. Use Equation 36.23 and the law of total variance Equation 36.22 to show that every predictor has $R^2 \le \mathrm{Var}(\mathbb{E}[X \mid Y])/\mathrm{Var}(X)$. Evaluate this ceiling for the two-component mixture of the text, predicting X from the label Z, and for the dart, predicting X from Y.
   2. Given the input, the target is -1 or $+1$ with probability $\frac{1}{2}$ each. Which prediction minimizes the expected squared error, and how often does the target take that value? Which predictions minimize the expected absolute error $\mathbb{E}\big[|X - h(Y)|\big]$?

5. code **Affine coupling layer and Box–Muller.** ● Normalizing flows stack invertible maps whose log-determinants in Equation 36.31 are cheap to compute. The affine coupling layer of RealNVP (Dinh, Sohl-Dickstein, et al. 2017) maps $\mathbf{x} \in \mathbb{R}^2$ to

$$y_1 = x_1, \qquad y_2 = x_2\, e^{s(x_1)} + t(x_1), \tag{36.33}$$

where s and t are arbitrary functions, for example small neural networks.

   1. Write down the inverse map and the Jacobian $J_g(\mathbf{x})$. Show that $\log|\det J_g(\mathbf{x})| = s(x_1)$, however complicated s and t are.
   2. Let X_1 and X_2 be independent standard normal variables, and take $s(x_1) = \tanh(x_1)$ and $t(x_1) = x_1^2/2$. Use Equation 36.31 to write $\log p_{\mathbf{Y}}(\mathbf{y})$ in closed form. With NumPy, evaluate $p_{\mathbf{Y}}$ on a grid over $[-6, 6] \times [-12, 24]$ with spacing 0.02 and check that its Riemann sum is close to 1. Check the log-determinant against the determinant of a finite-difference Jacobian at a few random points.
   3. The first coordinate passes through the layer unchanged. What is the distribution of Y_1, whatever s and t are? How must a second coupling layer be arranged so that the composition can change both marginals, and what is the log-determinant of the composition?
   4. Flows can also run from uniform noise to Gaussians. Let U_1, U_2 be independent and uniform on $(0, 1)$, and set $R = \sqrt{-2\log U_1}$, $\Theta = 2\pi U_2$, and $(Z_1, Z_2) = (R\cos\Theta, R\sin\Theta)$. Find the density of R with the one-dimensional rule, then the joint density of (R, Θ), and change to Cartesian coordinates with the polar Jacobian of the text to show that Z_1 and Z_2 are independent standard normal variables. This is the Box–Muller method. Check with 10^5 samples that the sample covariance matrix of (Z_1, Z_2) is close to the identity.

36.3 Distributions

In Section 36.1 and Section 36.2, we developed a general language for random quantities: probability mass functions, densities, expectations, and variances. Every predictive model in deep learning speaks this language, whether it says so explicitly or not. A binary classifier outputs a single number, the probability that the label is 1. A language model outputs a probability for every token in its vocabulary, constrained to sum to one. A regression network predicts a real number and is trained by minimizing squared error. In each case, the model is implicitly choosing a probability distribution for its labels, and that choice fixes the loss function it minimizes. Which distributions are these, where do they come from, and how do we compute their means and variances?

Surprisingly, a single coin flip will carry us through almost the entire answer. Suppose a coin lands heads 9 times in 13 flips. If the coin is fair, how unusual is that outcome? Answering this question requires the distribution of the number of heads in repeated flips, which happens to be the exact same distribution that governs how many test examples a classifier gets right. As we will see shortly, the probability of seeing 9 heads is about 9%.

The central idea of this section is that a handful of fundamental distributions all grow out of that one coin flip. Counting heads over a fixed number of flips gives the binomial distribution. Taking many flips that each rarely come up heads leads to the Poisson distribution, and the waiting time until the next rare event gives the exponential distribution. Adding up many ordinary flips and rescaling them produces the Gaussian. Because each distribution

is built from simpler pieces, its mean and variance follow directly from the construction, each time by a short and revealing argument.

Choosing the right distribution is not merely a matter of notation. A count model whose variance is too small, for instance, will make every error bar computed from it deceptively narrow. Three results in this section matter well beyond it: the Poisson limit and its waiting times, the central limit theorem, and the conditioning formula for Gaussian vectors. By contrast, the discrete uniform, Gamma, and Laplace distributions can be skimmed on a first reading.

Figure 36.13 maps out how the distributions we will study are constructed from one another.

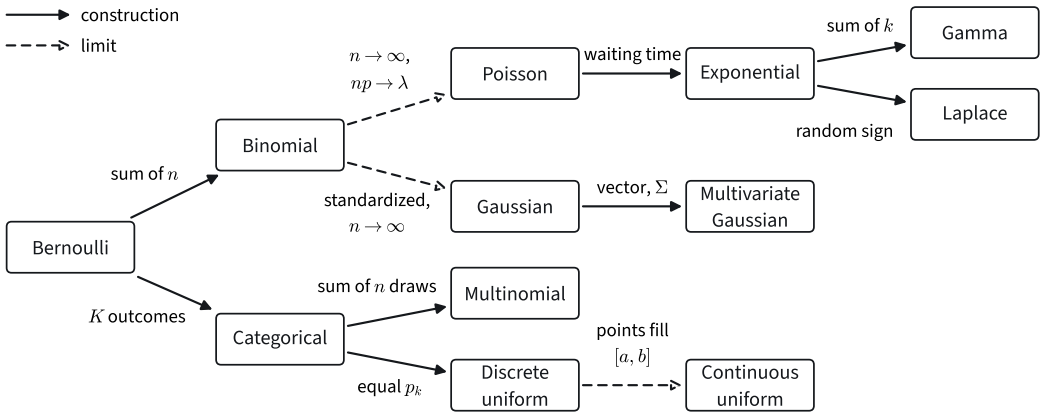


Figure 36.13: Every distribution in this section grows out of a single coin flip. Solid arrows construct a new distribution (through a sum, a waiting time, a random sign, a vector of coordinates, or equal probabilities), while dashed arrows take a limit as the number of trials or points grows. The Poisson and exponential distributions describe the same stream of events, counted in windows and timed between arrivals.

36.3.1 Discrete Distributions

A discrete random variable takes values in a countable set, such as $\{0, 1\}$ or $\{0, 1, 2, \dots\}$. Its distribution is specified by a probability mass function (pmf) $P(X = k) \geq 0$ satisfying $\sum_k P(X = k) = 1$. Its mean is $\mathbb{E}[X] = \sum_k k\, P(X = k)$, and its variance is $\mathrm{Var}(X) = \mathbb{E}[X^2] - \mathbb{E}[X]^2$. Figure 36.14 previews the discrete distributions we will meet in this part, evaluated on our running examples.

36.3.1.1 Bernoulli

The simplest random quantity has just two outcomes: a coin comes up heads or tails, a user clicks an ad or ignores it, a classifier gets an example right or wrong. A single number describes this experiment completely: the probability p of the outcome we label 1.

A *Bernoulli* random variable $X \sim \mathrm{Bernoulli}(p)$ takes the value 1 with probability p and 0 with probability $1 - p$, where $0 \leq p \leq 1$. Both cases combine neatly into a single formula:

$$P(X = x) = p^x (1 - p)^{1 - x}, \qquad x \in \{0, 1\}. \tag{36.34}$$

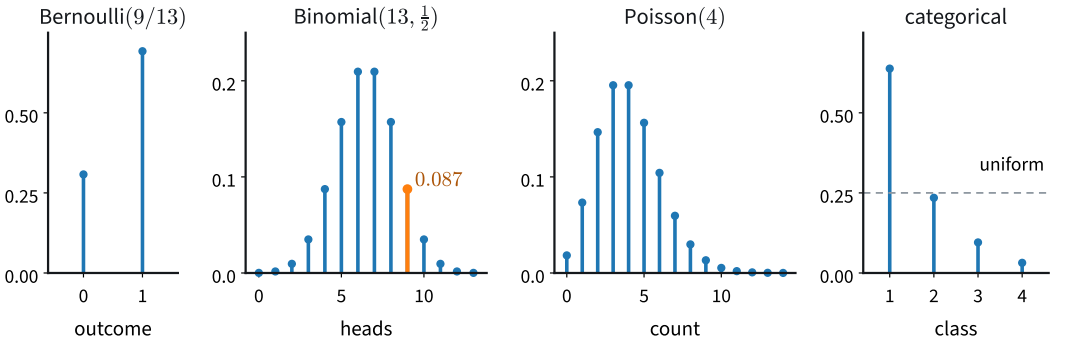


Figure 36.14: Four probability mass functions on the running examples. Bernoulli$(9/13)$: a single flip of a coin matching the observed frequency of heads. Binomial$(13, \frac{1}{2})$: the number of heads in 13 fair flips, with the orange stem highlighting $P(X = 9) = 715/8192 \approx 0.087$. Poisson$(4)$: server requests per minute at an average rate of four. A categorical distribution over four classes produced by the logits $(2, 1, 0.1, -1)$, shown against the uniform probability $1/4$. The binomial and Poisson distributions peak near their means (6.5 and 4) and share a vertical scale; the two panels with fewer outcomes share another.

Here the exponent acts as a switch: when $x = 1$ the formula reduces to p, and when $x = 0$ it reduces to $1 - p$.

The moments are easy to compute (as in Section 1.6). The mean is simply $\mathbb{E}[X] = 1 \cdot p + 0 \cdot (1 - p) = p$. Because X takes only the values 0 and 1, we have $X^2 = X$, so $\mathbb{E}[X^2] = p$ as well. The variance is therefore $\mathrm{Var}(X) = p - p^2 = p(1 - p)$, the product of the two outcome probabilities. It vanishes when the outcome is certain ($p = 0$ or $p = 1$) and reaches its maximum of $\frac{1}{4}$ for a fair coin ($p = \frac{1}{2}$).

In deep learning, a binary classifier outputs an estimate $\hat{p}$ of the probability that the label y equals 1. By Equation 36.34, the negative log-probability it assigns to the observed label is $-\log P(y) = -y \log \hat{p} - (1-y) \log(1-\hat{p})$, which is the *binary cross-entropy* loss. Training a classifier with binary cross-entropy is therefore equivalent to fitting a Bernoulli distribution to each label.

Two different coins will appear in what follows, and it pays to keep them distinct. Our opening question asks about a fair coin, $p = \frac{1}{2}$. By contrast, the coin whose probability matches our observed data has $p = 9/13 \approx 0.69$, and a single flip of it has variance $\frac{9}{13} \cdot \frac{4}{13} \approx 0.21$.

36.3.1.2 Binomial

Returning to our opening puzzle, we want to know the distribution of the number of heads in 13 fair flips, and in particular how likely it is to see 9.

Let us start with the smallest nontrivial case, two flips. The four possible outcomes are TT, HT, TH, and HH, giving

$$P(X = 0) = (1 - p)^2, \qquad P(X = 1) = 2p(1 - p), \qquad P(X = 2) = p^2.$$

The factor of 2 in $P(X = 1)$ counts the two ways to arrange one head and one tail, and by the binomial expansion the three probabilities sum to $(p + 1 - p)^2 = 1$.

In general, the number of heads in n flips is a sum of n indicator variables, one per flip. If the flips $X_1, \dots, X_n$ are independent and each follows Bernoulli(p) with the same p, their sum follows the *binomial* distribution:

$$X = \sum_{i=1}^{n} X_i \sim \text{Binomial}(n, p). \tag{36.35}$$

There are $\binom{n}{k}$ ways to choose which k of the n flips land heads, and by independence each specific ordering has probability $p^k(1-p)^{n-k}$. Multiplying the number of orderings by the probability of any one ordering gives the binomial pmf:

$$P(X = k) = \binom{n}{k} p^k (1-p)^{n-k}, \qquad k = 0, 1, \dots, n. \tag{36.36}$$

For $n = 2$, this reproduces the three probabilities above. For our fair coin, 9 heads in 13 flips can occur in $\binom{13}{9} = 715$ different orderings, each with probability $2^{-13} = 1/8192$, so

$$P(X = 9) = \frac{715}{8192} \approx 0.087.$$

The probability of seeing 9 heads *or more* is $(715 + 286 + 78 + 13 + 1)/8192$, which equals $1093/8192 \approx 0.133$. Whether a tail probability of 13.3% is enough reason to doubt the fairness of the coin is a statistical question that we set aside for now.

Mean and variance. Computing $\sum_k k\, P(X = k)$ directly from the pmf would be tedious. Instead, the sum representation Equation 36.35 gives both moments in one line. Because expectation is linear for any collection of random variables, dependent or not,

$$\mathbb{E}[X] = \sum_{i=1}^{n} \mathbb{E}[X_i] = np.$$

For the variance, recall from Section 36.2 that independent variables have zero covariance, so the variance of a sum of *independent* variables is the sum of their variances:

$$\text{Var}(X) = \sum_{i=1}^{n} \text{Var}(X_i) = np(1-p).$$

Thus n independent flips have n times the mean and n times the variance of a single flip. For 13 fair flips, the mean is 6.5, the variance is 3.25, and the standard deviation is $\sqrt{3.25} \approx 1.80$. Seeing 9 heads places us $(9 - 6.5)/1.80 \approx 1.4$ standard deviations above the mean.

Notice that the two moment formulas rely on different assumptions, a distinction that is easy to overlook. The mean np holds even if the flips are dependent, and if flip i has its own probability p_i the mean simply becomes $\sum_i p_i$. The variance formula, however, critically requires independence. If our 13 "flips" were actually a single fair flip copied 13 times, X would take only the values 0 or 13, and its variance would be $13^2 \cdot \frac{1}{4} = 42.25$, thirteen times larger than $np(1-p)$.

The successive ratio. Evaluating Equation 36.36 directly involves factorials, which overflow quickly when n is large. Fortunately, we can compute the entire pmf without a single factorial by looking at the ratio of consecutive probabilities, where all factorials cancel:

$$\frac{\binom{n}{k+1}}{\binom{n}{k}} = \frac{k!\,(n-k)!}{(k+1)!\,(n-k-1)!} = \frac{n-k}{k+1}, \qquad \text{so} \qquad \frac{P(X=k+1)}{P(X=k)} = \frac{n-k}{k+1}\cdot\frac{p}{1-p}.$$

This ratio has a direct combinatorial meaning. Starting from a set of k heads, we can pick any of the $n-k$ remaining tails and flip it to a head; each resulting set of $k+1$ heads is reached in $k+1$ ways, and the factor $p/(1-p)$ swaps the probability of a tail for that of a head. Starting from $P(X=0)=(1-p)^n$ and multiplying repeatedly by this ratio generates the entire pmf. Moreover, the ratio exceeds 1 while $k+1<(n+1)p$ and drops below 1 thereafter, proving that the binomial pmf rises smoothly to a single peak near its mean and then falls.

The recursion takes just three lines of Python. We package it as a reusable function, since we will compare both the Poisson limit and the Gaussian bell against it below.

```
def binomial_pmf(n, p, kmax):
    """P(X=k) for k = 0..kmax, built from P(X=0) by the successive ratio."""
    pmf = [(1 - p) ** n]
    for k in range(kmax):
        pmf.append(pmf[-1] * max(n - k, 0) / (k + 1) * p / (1 - p))
    return np.array(pmf)

n, p = 13, 0.5
pmf = binomial_pmf(n, p, n)
print(f'P(X=9): by the ratio {pmf[9]:.4f}, by counting {math.comb(13, 9) / 2**13:.4f}')
print(f'P(X>=9) = {pmf[9:].sum():.4f}')
print(f'mean np = {n * p}, variance np(1-p) = {n * p * (1 - p)}')
```

```
P(X=9): by the ratio 0.0873, by counting 0.0873
P(X>=9) = 0.1334
mean np = 6.5, variance np(1-p) = 3.25
```

As expected, the ratio recursion reproduces $715/8192 \approx 0.0873$ and the tail probability 0.1334.

36.3.1.3 Poisson

Many real-world counts have no natural upper limit n. Suppose a web server receives requests at an average rate of $\lambda = 4$ per minute, with no fixed cap on how many might arrive. How likely is a completely silent minute, or a sudden burst of 10 or more requests?

To connect this to our coin-flipping machinery, imagine dividing the minute into n tiny subintervals. Our model makes two simplifying assumptions: each subinterval is short enough that two requests never arrive in the same slice, and the slices act as independent trials with equal arrival probability $4/n$. Under these assumptions, the total count in a minute follows Binomial$(n, 4/n)$. Its mean is $n\cdot\frac{4}{n}=4$ for every n, and its variance $4(1-\frac{4}{n})$ creeps steadily toward 4. The probability of a silent minute is $(1-\frac{4}{n})^n$, which equals 0.0115 at

$n = 20$, 0.0169 at $n = 100$, and 0.0182 at $n = 1000$, converging to $e^{-4} \approx 0.0183$. The probabilities stabilize as the slices grow finer, pointing to a clean limiting distribution.

Derivation of the limit. Consider Binomial$(n, \lambda/n)$ for a fixed count k as $n \to \infty$. Rewriting the binomial pmf exposes four factors, three of which have elementary limits:

$$\binom{n}{k}\left(\frac{\lambda}{n}\right)^k\left(1-\frac{\lambda}{n}\right)^{n-k} = \underbrace{\frac{n(n-1)\cdots(n-k+1)}{n^k}}_{\to 1}\cdot\frac{\lambda^k}{k!} \cdot \underbrace{\left(1-\frac{\lambda}{n}\right)^n}_{\to e^{-\lambda}}\cdot\underbrace{\left(1-\frac{\lambda}{n}\right)^{-k}}_{\to 1}.$$

The first factor is a product of k fractions that each approach 1; the third is the classic limit $(1-\lambda/n)^n \to e^{-\lambda}$ that defines the exponential function; and the last approaches 1 because k is held fixed. The same argument works for any sequence of success probabilities p_n with $np_n \to \lambda$, since $(1-p_n)^n \to e^{-\lambda}$ still holds. This result is known as the *law of rare events*, and the surviving factors give the *Poisson* pmf with rate $\lambda > 0$:

$$P(X = k) = \frac{\lambda^k e^{-\lambda}}{k!}, \qquad k = 0, 1, 2, \ldots \tag{36.37}$$

We write $X \sim \text{Poisson}(\lambda)$. Whenever a count arises from many independent opportunities, each individually unlikely, with a fixed expected total λ, the Poisson distribution appears. Its probabilities sum to one immediately from the Taylor series $\sum_{k\geq 0} \lambda^k/k! = e^{\lambda}$.

Mean and variance. We can read both moments directly from the pmf. In the sum for $\mathbb{E}[X]$, the $k = 0$ term vanishes, the factor k cancels against $k!$, and shifting the index to $j = k-1$ leaves the total probability of a Poisson variable:

$$\mathbb{E}[X] = \sum_{k\geq 1} k\,\frac{e^{-\lambda}\lambda^k}{k!} = \lambda\sum_{k\geq 1}\frac{e^{-\lambda}\lambda^{k-1}}{(k-1)!} = \lambda\sum_{j\geq 0}\frac{e^{-\lambda}\lambda^j}{j!} = \lambda.$$

Applying the exact same trick to $k(k-1)$ yields $\mathbb{E}[X(X-1)] = \lambda^2$, from which the variance follows:

$$\text{Var}(X) = \mathbb{E}[X(X-1)] + \mathbb{E}[X] - \mathbb{E}[X]^2 = \lambda^2 + \lambda - \lambda^2 = \lambda.$$

A defining signature of the Poisson distribution is that its mean and variance are equal. For our web server, $P(X = 0) = e^{-4} \approx 0.018$, $P(X = 4) = \frac{4^4}{4!}e^{-4} \approx 0.195$, and $P(X \geq 10) \approx 0.008$: a silent minute happens roughly once an hour, and a burst of ten or more requests about once every two hours. This equality of mean and variance is also a sharp diagnostic. If observed request counts over many minutes show a variance several times larger than their mean, then at least one of our modeling assumptions has broken down (perhaps traffic fluctuates over the hour, or requests trigger cascading follow-up requests), and error bars from a Poisson model will be dangerously narrow. The standard remedy, letting the Poisson rate itself be random, is explored in exercise 1.

The Poisson process. Extending our slicing picture across the entire timeline with rate λ per unit time gives a model of events arriving in continuous time. Any window of length t contains nt slices, so its event count is Binomial$(nt, \lambda/n)$, which converges to Poisson(λt). Non-overlapping windows draw from disjoint slices, making their counts independent. This continuous-time stream of events is the *Poisson process* with rate λ. In the next part, we will see that the waiting times between its events follow the exponential distribution.

Like the binomial, the Poisson pmf obeys a simple successive ratio, $P(X = k+1)/P(X = k) = \lambda/(k+1)$. Let us use it to compute our server probabilities and check the moments against $100{,}000$ simulated minutes.

```
rng = np.random.default_rng()

def poisson_pmf(lam, kmax):
    """P(X=k) for k = 0..kmax, built from P(X=0) = e^-lam by the ratio lam/(k+1)."""
    pmf = [math.exp(-lam)]
    for k in range(kmax):
        pmf.append(pmf[-1] * lam / (k + 1))
    return np.array(pmf)

pmf = poisson_pmf(4.0, 60)
print(f'P(0) = {pmf[0]:.3f}, P(4) = {pmf[4]:.3f}, P(X>=10) = {pmf[10:].sum():.3f}')
counts = rng.poisson(4.0, 100000)
print(f'100,000 simulated minutes: mean {counts.mean():.2f}, '
      f'variance {counts.var(ddof=1):.2f}')
```

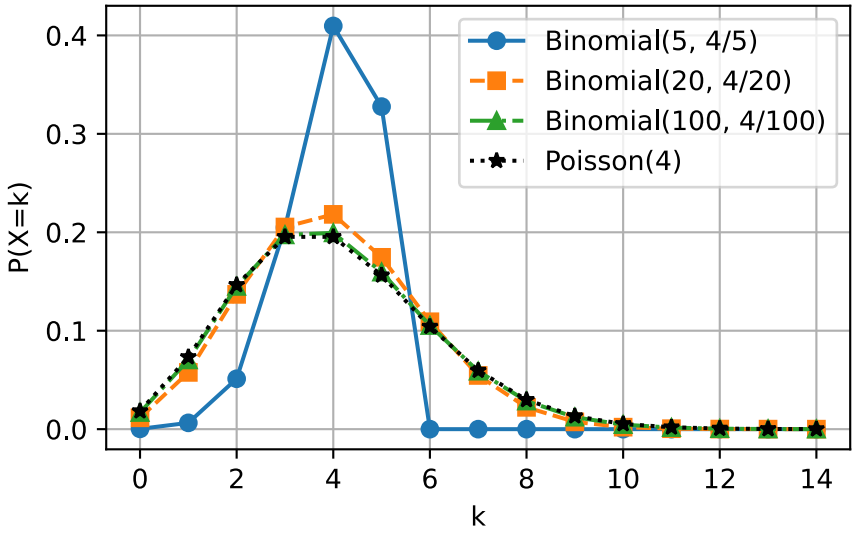
```
P(0) = 0.018, P(4) = 0.195, P(X>=10) = 0.008
100,000 simulated minutes: mean 4.00, variance 4.01
```

The simulated counts have both mean and variance right around 4.

The binomial-to-Poisson limit. How quickly does Binomial$(n, 4/n)$ converge to Poisson(4)? Let us overlay their pmfs and measure the discrepancy using the *total-variation distance*, $\frac{1}{2}\sum_k |P_{\text{Bin}}(k) - P_{\text{Poi}}(k)|$, which equals the largest difference between the probabilities that the two distributions assign to any event.

```
ns, ks = (5, 20, 100), np.arange(15)
d2l.plot(ks, [binomial_pmf(n, 4 / n, 14) for n in ns] + [poisson_pmf(4.0, 14)],
         xlabel='k', ylabel='P(X=k)', fmts=('o-', 's--', '^-.', 'k*:'),
         legend=[f'Binomial({n}, 4/{n})' for n in ns] + ['Poisson(4)'],
         figsize=(5, 3))
for n in (20, 100, 1000):
    tv = 0.5 * np.abs(binomial_pmf(n, 4 / n, 60) - poisson_pmf(4.0, 60)).sum()
    print(f'n = {n:4d}: total-variation distance {tv:.3f}')
```

```
n =   20: total-variation distance 0.056
n =  100: total-variation distance 0.010
n = 1000: total-variation distance 0.001
```



At $n = 5$, the binomial cannot exceed five requests and its peak sits noticeably too high. By $n = 100$, the two pmfs are visually indistinguishable. The total-variation distance drops from 0.056 at $n = 20$ to 0.010 at $n = 100$ and 0.001 at $n = 1000$, shrinking roughly in proportion to $1/n$.

36.3.1.4 Categorical and Multinomial

A classifier with K classes, or a language model predicting the next token from a vocabulary, chooses among more than two outcomes. The categorical distribution is simply a K-sided coin.

A *categorical* random variable takes values in $\{1, \dots, K\}$ with probabilities specified by a vector $\mathbf{p} = (p_1, \dots, p_K)$, where $p_k \geq 0$ and $\sum_k p_k = 1$:

$$P(X = k) = p_k. \tag{36.38}$$

When $K = 2$, this reduces to the Bernoulli distribution. In neural networks, we rarely predict probabilities directly; instead, the network outputs unconstrained real scores $z_1, \dots, z_K$ called *logits*, and converts them into a categorical distribution via the *softmax* function:

$$p_k = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}}. \tag{36.39}$$

Exponentiating guarantees that every entry is positive, and dividing by the sum ensures they add to one. For example, the logits $\mathbf{z} = (2, 1, 0.1, -1)$ produce $\mathbf{p} \approx (0.638, 0.235, 0.095, 0.032)$. Writing the true class as a one-hot vector $\mathbf{y}$, the negative log-probability assigned to the observed label is $-\log p_y = -\sum_k y_k \log p_k$, which is the *softmax cross-entropy* loss of Section 3.1.2. Just as binary cross-entropy fits a Bernoulli distribution, softmax cross-entropy fits a categorical distribution to each label.

The discrete uniform. The simplest categorical distribution gives every outcome the same probability. On $\{1, \dots, n\}$ with $p_k = 1/n$, this is the *discrete uniform* distribution $\mathrm{Unif}\{1, \dots, n\}$: a fair die, or an index drawn uniformly at random. Its mean and variance follow from the standard sums $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ and $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$:

$$\begin{aligned} \mathbb{E}[X] &= \frac{1}{n}\sum_{i=1}^{n} i = \frac{n+1}{2}, \\ \mathrm{Var}(X) &= \frac{(n+1)(2n+1)}{6} - \frac{(n+1)^2}{4} \\ &= \frac{(n+1)\big(2(2n+1) - 3(n+1)\big)}{12} = \frac{n^2-1}{12}. \end{aligned}$$

The mean sits right at the midpoint, while the variance grows like n^2 because the outcomes span a range of length $n-1$. For a fair six-sided die, the mean is 3.5 and the variance is $35/12 \approx 2.92$.

The multinomial. Just as repeating a Bernoulli trial n times yields the binomial, drawing n independent samples from a categorical distribution and tallying how often each class appears yields the *multinomial* distribution. The count vector $\mathbf{x} = (x_1, \dots, x_K)$, constrained by $\sum_k x_k = n$, has pmf

$$P(\mathbf{x}) = \frac{n!}{x_1! \cdots x_K!} \prod_{k=1}^{K} p_k^{x_k}. \tag{36.40}$$

As with the binomial, the leading multinomial coefficient counts the number of orderings of n draws that yield the counts $\mathbf{x}$, and the product gives the probability of any single ordering. For $K = 2$, the coefficient reduces to $\binom{n}{x_1}$ and we recover the binomial ($13!/(9!\,4!) = 715$). Looking at any single class k in isolation, each draw either lands in class k or does not, so $x_k \sim \mathrm{Binomial}(n, p_k)$ with $\mathbb{E}[x_k] = np_k$ and $\mathrm{Var}(x_k) = np_k(1-p_k)$. Across all classes jointly, the count vector has mean $n\mathbf{p}$ and covariance matrix $n(\mathrm{diag}(\mathbf{p}) - \mathbf{p}\mathbf{p}^\top)$, derived in exercise 7. Notice that every off-diagonal entry is negative: because the total number of draws n is fixed, extra counts in one class must come at the expense of the others. When $n = 1$, the count vector is simply the one-hot encoding of a single categorical draw.

Let us compute the softmax probabilities for our example logits and draw $10{,}000$ samples from the resulting categorical distribution to check the empirical frequencies.

```
z = np.array([2.0, 1.0, 0.1, -1.0])          # logits for K = 4 classes
p_cat = np.exp(z) / np.exp(z).sum()           # softmax
draws = rng.choice(4, size=10000, p=p_cat)
print('softmax p   =', p_cat.round(3))
print('frequencies =', (np.bincount(draws, minlength=4) / 10000).round(3))
```

```
softmax p   = [0.638 0.235 0.095 0.032]
frequencies = [0.639 0.234 0.095 0.033]
```

The empirical frequencies match $\mathbf{p}$ to within about a percentage point.

36.3.2 Continuous Distributions

A continuous random variable is described by a probability density function $p(x) \geq 0$ satisfying $\int p(x)\,dx = 1$. Probabilities are areas under the density curve, and expectations and variances are integrals, as in Section 36.1. We begin with the exponential distribution, because the Poisson process has already handed it to us.

36.3.2.1 Exponential

Let us return to our web server. The Poisson distribution tells us how many requests arrive in a minute. An engineer configuring a timeout asks a closely related question about the same stream of traffic: how long will we wait until the next request arrives?

Remarkably, the two questions share an answer. Waiting more than one minute for the next request is the exact same event as observing zero requests during that minute, so $P(\text{wait} > 1) = P(N = 0) = e^{-4} \approx 0.018$, the Poisson probability we already computed. Figure 36.15 illustrates how both distributions describe a single stream of events.

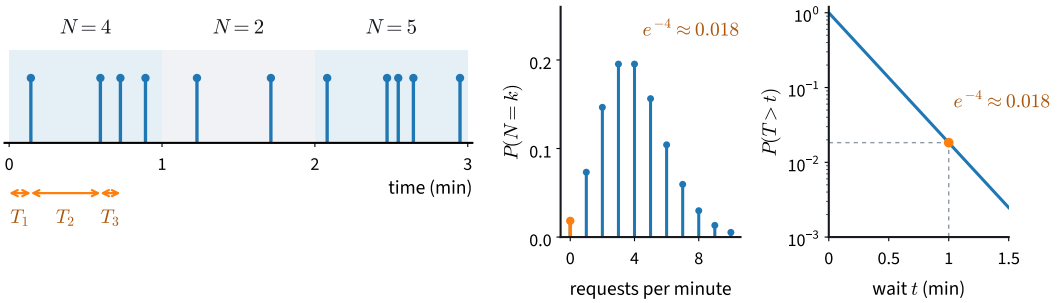


Figure 36.15: One stream of requests arriving at an average rate of four per minute, viewed two ways. (a) Event times over three minutes: the unit windows contain $N = 4$, 2, and 5 requests, while T_1, T_2, and T_3 are the first three waiting times. (b) The number of requests in a minute follows Poisson(4), so a silent minute has probability $e^{-4} \approx 0.018$. (c) The probability that a wait exceeds t minutes is e^{-4t}, a straight line on a logarithmic axis; waiting longer than a minute and seeing a silent minute are the same event.

The exact same reasoning applies to any duration $t \geq 0$. In a Poisson process with rate λ, the waiting time T from time 0 until the first event exceeds t if and only if the interval $[0, t]$ contains zero events. Since the count in $[0, t]$ follows Poisson(λt), we obtain

$$P(T > t) = P(N(t) = 0) = e^{-\lambda t}, \qquad t \geq 0.$$

The cumulative distribution function is therefore $F(t) = 1 - e^{-\lambda t}$, and differentiating it gives the density of the *exponential* distribution with rate $\lambda > 0$:

$$p(t) = \lambda e^{-\lambda t} \quad (t \geq 0), \qquad p(t) = 0 \quad (t < 0), \qquad F(t) = 1 - e^{-\lambda t}, \qquad T \sim \text{Exp}(\lambda). \tag{36.41}$$

In other words, the density measures the rate at which the probability of an empty window decays. Because future slices of a Poisson process after any arrival are fresh independent

trials, the waiting time after each event has the same distribution: the gaps between consecutive arrivals are independent $\mathrm{Exp}(\lambda)$ random variables.

Mean and variance. Two quick integrations by parts give the moments. Using $\int_0^\infty u\,e^{-u}\,du = [-ue^{-u}]_0^\infty + \int_0^\infty e^{-u}\,du = 1$ and $\int_0^\infty u^2 e^{-u}\,du = [-u^2e^{-u}]_0^\infty + 2\int_0^\infty u\,e^{-u}\,du = 2$, the substitution $u = \lambda t$ yields

$$\mathbb{E}[T] = \int_0^\infty t\,\lambda e^{-\lambda t}\,dt = \frac{1}{\lambda}, \qquad \mathbb{E}[T^2] = \frac{2}{\lambda^2}, \qquad \mathrm{Var}(T) = \frac{2}{\lambda^2} - \frac{1}{\lambda^2} = \frac{1}{\lambda^2}.$$

As intuition demands, a higher arrival rate means shorter waits. At four requests per minute, the average wait is $\frac{1}{4}$ minute, or 15 seconds. The median m solves $F(m) = \frac{1}{2}$, giving $m = \log 2/\lambda \approx 10.4$ seconds: half of all waits last less than about 10 seconds even though the mean is 15. Because the density peaks at $t = 0$ and trails off in a long right tail, the mean is pulled well above the median.

Memorylessness. Suppose you have already waited s minutes without seeing a request. What is the probability that you will have to wait at least t minutes more?

$$P(T > s + t \mid T > s) = \frac{P(T > s + t)}{P(T > s)} = \frac{e^{-\lambda(s+t)}}{e^{-\lambda s}} = e^{-\lambda t} = P(T > t).$$

Time already spent waiting tells you nothing at all about the remaining wait. Among all continuous distributions on $(0, \infty)$, the exponential is the only one with this *memoryless* property, as exercise 2 proves. Memorylessness also marks the boundary of where the model applies. A light bulb whose filament wears down with use is more likely to fail in the next hour the older it gets, so its lifetime is not exponential.

The Gamma distribution. What if we ask for the time T_k until the k-th request arrives? Since T_k is the sum of k independent exponential waits, linearity of expectation and independence immediately give its mean k/λ and variance k/λ^2. Its density also drops straight out of the Poisson process: the k-th arrival occurs after time t if and only if at most $k - 1$ events occur in $[0, t]$, so

$$P(T_k > t) = P(N(t) \le k - 1) = \sum_{j=0}^{k-1} e^{-\lambda t}\frac{(\lambda t)^j}{j!}.$$

When we differentiate the j-th term with respect to t, the product rule produces $-\lambda e^{-\lambda t}\frac{(\lambda t)^j}{j!} + \lambda e^{-\lambda t}\frac{(\lambda t)^{j-1}}{(j-1)!}$. Notice how the second piece cancels the first piece of the previous term. The entire sum telescopes, leaving only the first piece of the final ($j = k - 1$) term:

$$p(t) = -\frac{d}{dt}P(T_k > t) = \frac{\lambda^k\, t^{k-1} e^{-\lambda t}}{(k-1)!}, \qquad t > 0.$$

For $k = 1$, we recover the exponential density. To extend this family beyond integer k, we replace $(k-1)!$ with the *Gamma function* $\Gamma(z) = \int_0^\infty u^{z-1}e^{-u}\,du$ for $z > 0$. Integration

by parts shows that $\Gamma(z+1) = z\,\Gamma(z)$, and since $\Gamma(1) = 1$, we have $\Gamma(n+1) = n!$ for every nonnegative integer n. This defines the *Gamma distribution* with shape $\alpha > 0$ and rate $\beta > 0$:

$$p(t) = \frac{\beta^{\alpha}\, t^{\alpha-1} e^{-\beta t}}{\Gamma(\alpha)}, \qquad t > 0, \qquad T \sim \text{Gamma}(\alpha, \beta). \tag{36.42}$$

Substituting $u = \beta t$ reduces the integral of $p(t)$ to $\Gamma(\alpha)/\Gamma(\alpha) = 1$, and the same substitution yields $\mathbb{E}[T] = \Gamma(\alpha+1)/(\beta\,\Gamma(\alpha)) = \alpha/\beta$ and $\text{Var}(T) = \alpha/\beta^2$. When $\alpha = k$ and $\beta = \lambda$, these match the moments of the k-th waiting time. For our web server, the 4th request arrives after an average of 1 minute with standard deviation 0.5 minutes, and $P(T_4 > 1) = P(N(1) \le 3) \approx 0.43$: in 43% of minutes, fewer than four requests arrive, even though four is the average.

Sampling. Recall from Section 36.1 that the inverse-transform method converts a uniform draw U into a sample from any continuous distribution with an invertible c.d.f. F by computing $X = F^{-1}(U)$. For the exponential distribution, $F^{-1}(q) = -\log(1-q)/\lambda$, so $T = -\log(1-U)/\lambda$. Although $1-U$ and U have the same distribution, writing $1-U$ matters in practice: random number generators return values in $[0, 1)$, where $U = 0$ can occur and $-\log U$ would overflow to infinity.

If our connection between the Poisson process and the exponential distribution is airtight, we should be able to run the construction backwards: generate independent exponential gaps, accumulate them into arrival times, and recover Poisson counts per minute. Let us test this with $60{,}000$ gaps at rate 4 and inspect the first $10{,}000$ minutes.

```
lam = 4.0                                       # requests per minute
gaps = -np.log(1 - rng.random(60000)) / lam     # inverse transform: Exp(lam)
times = np.cumsum(gaps)                         # event times, about 15,000 min
counts = np.bincount(times[times < 10000].astype(int), minlength=10000)
print(f'requests per minute: mean {counts.mean():.2f}, variance {counts.var(ddof=1):.2f}')
print(f'silent minutes {np.mean(counts == 0):.4f},   waits over 1 min '
      f'{np.mean(gaps > 1):.4f},   e^-4 = {math.exp(-4):.4f}')
print(f'mean wait {gaps.mean():.3f} min,   minutes with at most 3 requests '
      f'{np.mean(counts <= 3):.3f}')
```

```
requests per minute: mean 3.99, variance 4.00
silent minutes 0.0204,   waits over 1 min 0.0191,   e^-4 = 0.0183
mean wait 0.252 min,   minutes with at most 3 requests 0.434
```

The simulated counts indeed have mean and variance close to 4. Both the fraction of silent minutes and the fraction of waits longer than a minute sit right around e^{-4}, and the fraction of minutes with at most three requests matches the Gamma prediction $P(T_4 > 1) \approx 0.43$. Generating exponential gaps reproduces Poisson counts.

36.3.2.2 Uniform

Every sampler in this section starts from the same raw material: a number drawn uniformly at random from $[0, 1)$. That distribution, generalized to any finite interval, is the continuous uniform distribution.

The *uniform* distribution $U(a,b)$ on an interval $[a,b]$ with $a<b$ has a constant density, fixed by the requirement that it integrate to one:

$$p(x)=\begin{cases}\frac{1}{b-a} & x\in[a,b],\\ 0 & \text{otherwise.}\end{cases} \tag{36.43}$$

Every subinterval of equal length receives the same probability, namely its length divided by $b-a$. Both moments follow by direct integration. For the variance, it is cleaner to center the interval at its midpoint so that it runs from $-c$ to c with $c=\frac{b-a}{2}$:

$$\mathbb{E}[X]=\int_a^b \frac{x}{b-a}\,dx=\frac{b^2-a^2}{2(b-a)}=\frac{a+b}{2},\qquad \mathrm{Var}(X)=\int_{-c}^{c}\frac{y^2}{2c}\,dy=\frac{c^2}{3}=\frac{(b-a)^2}{12}.$$

In particular, $U(0,1)$ has mean $\frac{1}{2}$ and variance $\frac{1}{12}$, while $U(-\sqrt{3},\sqrt{3})$ has variance $(2\sqrt{3})^2/12=1$. Notice how this formula is the continuous limit of the discrete uniform: n equally likely points spaced h apart have variance $h^2(n^2-1)/12$, which converges to $(b-a)^2/12$ as the points fill $[a,b]$ with $nh=b-a$. Through the inverse transform, $U(0,1)$ can be reshaped into any continuous distribution, as we just did for the exponential. Visually, the uniform has flat density, hard edges, and no tails at all, so it looks nothing like a Gaussian. That makes it the ideal test case when we watch the central limit theorem turn sums of uniforms into a bell curve below.

36.3.2.3 Gaussian

Our successive ratio computes any binomial probability to high precision, yet it does not tell us what the distribution looks like as a whole. Is there a simple closed-form expression for its shape near the peak? Remarkably, there is, and only the mean and variance of the coin appear in it. This brings us to the most important distribution in probability and machine learning.

The log-ratio near the mean. Let $q=1-p$, and measure the count by its distance x from the mean, so $k=np+x$. Substituting this into the binomial's successive ratio gives

$$\frac{P(X=k+1)}{P(X=k)}=\frac{(n-k)\,p}{(k+1)\,q}=\frac{(nq-x)\,p}{(np+x+1)\,q}=\frac{1-x/(nq)}{1+(x+1)/(np)}.$$

When x is on the order of the standard deviation $\sqrt{npq}$ and n is large, both $x/(nq)$ and $(x+1)/(np)$ are small. Taking the logarithm and applying the linear approximation $\log(1+\epsilon)\approx\epsilon$ yields

$$\log\frac{P(X=k+1)}{P(X=k)}\approx-\frac{x}{nq}-\frac{x+1}{np}=-\frac{x}{npq}-\frac{1}{np},$$

where we combined the fractions using $\frac{1}{nq}+\frac{1}{np}=\frac{p+q}{npq}=\frac{1}{npq}$. To leading order, each step from $\log P(X=k)$ to $\log P(X=k+1)$ changes linearly with the distance x from the mean, at slope $-1/(npq)$. What kind of function has a linearly changing step? A parabola. Indeed, the parabola $-x^2/(2npq)$ changes by

$$-\frac{(x+1)^2 - x^2}{2npq} = -\frac{x}{npq} - \frac{1}{2npq}$$

as x increases by 1. This matches the step of $\log P$ up to a tiny constant $(p-q)/(2npq)$, which vanishes altogether for a fair coin and in general accumulates to only $O(1/\sqrt{n})$ over the $\sqrt{npq}$ steps around the peak. We conclude that

$$\log P(X = np + x) \approx C - \frac{x^2}{2npq}$$

for some constant C, at every count $k = np + x$ within a few standard deviations of the mean, regardless of whether np is an integer (for our 13 flips, $np = 6.5$). Exponentiating both sides, we see that near its peak the binomial takes the shape of a bell curve, $e^C e^{-x^2/(2npq)}$, centered at the mean with width set by its standard deviation $\sqrt{npq}$. Because we dropped higher-order remainders informally, this is a derivation sketch rather than a rigorous proof; the rigorous version is the *de Moivre–Laplace theorem*, the earliest form of the central limit theorem.

The normalizer. To determine the constant C, we use the fact that the probabilities must sum to one. Since integer counts are spaced 1 apart, the sum of the bell over all k is closely approximated by the integral $e^C \int_{-\infty}^{\infty} e^{-x^2/(2npq)}\,dx$. Everything therefore reduces to evaluating $I = \int_{-\infty}^{\infty} e^{-x^2/2}\,dx$. At first glance this integral looks hopeless, because $e^{-x^2/2}$ has no elementary antiderivative. The classic trick is to compute I^2 instead: as a double integral over the plane, switching to polar coordinates $x = r\cos\theta$, $y = r\sin\theta$ with area element $dx\,dy = r\,dr\,d\theta$ makes the integrand elementary:

$$\begin{aligned} I^2 &= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} e^{-(x^2+y^2)/2}\,dx\,dy = \int_0^{2\pi}\int_0^{\infty} e^{-r^2/2}\,r\,dr\,d\theta \\ &= 2\pi\left[-e^{-r^2/2}\right]_0^{\infty} = 2\pi. \end{aligned}$$

Hence $I = \sqrt{2\pi}$. The extra factor of r from the polar Jacobian gives $r\,e^{-r^2/2}$ the elementary antiderivative $-e^{-r^2/2}$, which $e^{-x^2/2}$ alone lacks. Changing variables via $x = \sqrt{npq}\,u$ then gives $\int_{-\infty}^{\infty} e^{-x^2/(2npq)}\,dx = \sqrt{2\pi npq}$, fixing $e^C = 1/\sqrt{2\pi npq}$ and leaving

$$P(X = np + x) \approx \frac{1}{\sqrt{2\pi npq}}\,e^{-x^2/(2npq)}.$$

Let us test this on our 13 fair flips, where $npq = 3.25$ and $k = 9$ corresponds to $x = 2.5$: the bell gives $e^{-2.5^2/6.5}/\sqrt{2\pi \cdot 3.25} \approx 0.0846$, surprisingly close to the exact value 0.0873 even for so small an n. Below, we measure how quickly the bell approximation sharpens as n grows.

```
for n in (13, 100, 1000):
    k = np.arange(n + 1)
    bell = np.exp(-(k - n / 2)**2 / (n / 2)) / np.sqrt(np.pi * n / 2)
```

```
    exact = binomial_pmf(n, 0.5, n)
    peak = n // 2
    print(f'n = {n:4d}: relative error at the peak {abs(bell[peak] / exact[peak] - 1):.2%}')
print(f'n = 13, k = 9: bell {np.exp(-2.5**2 / 6.5) / np.sqrt(np.pi * 6.5):.4f},'
      f' exact {binomial_pmf(13, 0.5, 13)[9]:.4f}')
```

```
n =   13: relative error at the peak 1.66%
n =  100: relative error at the peak 0.25%
n = 1000: relative error at the peak 0.03%
n = 13, k = 9: bell 0.0846, exact 0.0873
```

At $n = 13$ and $k = 9$, the bell is only about 3% low, and its relative error at the peak falls from 1.7% at $n = 13$ to 0.25% at $n = 100$ and 0.03% at $n = 1000$, shrinking roughly like $1/n$.

The standard Gaussian. The parabolic exponent suggests a natural rescaling. The standardized count $Y^{(n)} = (X - np)/\sqrt{npq}$ has mean 0 and variance 1 for every n, and its possible values are spaced $1/\sqrt{npq}$ apart. In these units, our bell approximation reads $P(X = k) \approx \phi(y)/\sqrt{npq}$ at $y = (k - np)/\sqrt{npq}$: a smooth density $\phi(y)$ multiplied by the grid spacing, where

$$\phi(y) = \frac{1}{\sqrt{2\pi}}\, e^{-y^2/2}.$$

This is the density of the *standard Gaussian* $\mathcal{N}(0, 1)$, the limiting distribution of the standardized binomial. It integrates to one because $I = \sqrt{2\pi}$, and its mean is 0 by symmetry around the origin. For the variance, integrating by parts with $u = x$ and $dv = x\, e^{-x^2/2}\, dx$ gives

$$\int_{-\infty}^{\infty} x^2\, \frac{e^{-x^2/2}}{\sqrt{2\pi}}\, dx = \frac{1}{\sqrt{2\pi}} \Big(\big[-x\, e^{-x^2/2}\big]_{-\infty}^{\infty} + \int_{-\infty}^{\infty} e^{-x^2/2}\, dx \Big) = 0 + 1 = 1.$$

Thus the standard Gaussian has variance 1. Shifting Y by μ and scaling by $\sigma > 0$ produces $X = \mu + \sigma Y$. By the change-of-variables formula of Section 36.2, its density is $\phi((x - \mu)/\sigma)/\sigma$, which defines the general *Gaussian* (or normal) distribution:

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}}\, \exp\!\Big(-\frac{(x - \mu)^2}{2\sigma^2} \Big), \qquad X \sim \mathcal{N}(\mu, \sigma^2). \tag{36.44}$$

Because $X = \mu + \sigma Y$, its mean is μ and its variance is σ^2: the two parameters are directly the first two moments, with σ controlling the width of the bell.

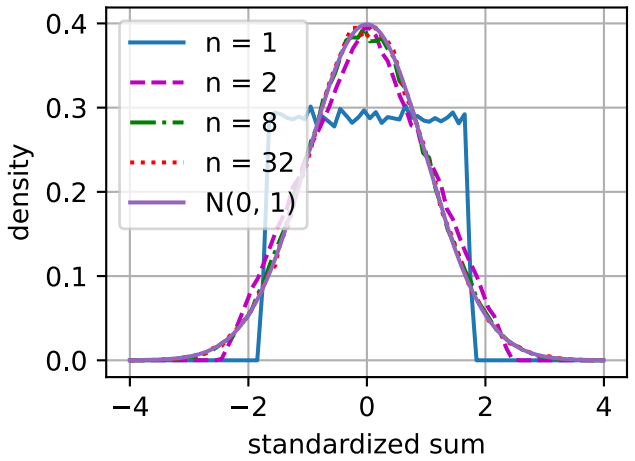
The central limit theorem. Looking back at our derivation, surprisingly little about coin flips entered the final shape beyond their mean and variance. In fact, the general theorem needs nothing else. Let $X_1, X_2, \ldots$ be independent and identically distributed random variables with mean μ and finite variance $0 < \sigma^2 < \infty$, and let $S_n = \sum_{i=1}^{n} X_i$. Then for every real number y,

$$\lim_{n\to\infty} P\Big(\frac{S_n - n\mu}{\sigma\sqrt{n}} \le y\Big) = \Phi(y) = \int_{-\infty}^{y} \phi(u)\,du. \tag{36.45}$$

In words, the standardized sum of many independent copies of *any* distribution with finite variance converges to the standard Gaussian. For Bernoulli summands, this is the de Moivre–Laplace theorem; for the general proof, see (Wasserman 2013). Only the mean and variance of the individual summands survive in the limit, which is why quantities built from many small independent effects, such as sensor or measurement noise, so reliably look Gaussian.

Sums of uniforms. The central limit theorem is even more striking when watched in action. Let us start from $U(0,1)$, which is completely flat with sharp edges and zero tails, sum n independent draws, standardize by the mean $n/2$ and standard deviation $\sqrt{n/12}$, and compare the resulting histograms with ϕ.

```
ns, x = (1, 2, 8, 32), np.arange(-4, 4, 0.01)
curves = []
for n in ns:                                    # standardized sums of n uniforms
    s = (rng.random((100000, n)).sum(1) - n / 2) / np.sqrt(n / 12)
    hist, edges = np.histogram(s, bins=80, range=(-4, 4), density=True)
    curves.append(np.interp(x, (edges[:-1] + edges[1:]) / 2, hist))
curves.append(np.exp(-x**2 / 2) / np.sqrt(2 * np.pi))     # N(0, 1) density
d2l.plot(x, curves, xlabel='standardized sum', ylabel='density',
         legend=[f'n = {n}' for n in ns] + ['N(0, 1)'])
```



At $n = 1$ we see the flat plateau of the uniform, and at $n = 2$ a sharp triangle. Already by $n = 8$, the histogram closely tracks the Gaussian bell, and by $n = 32$ the two curves are indistinguishable at the resolution of the plot.

The finite-variance assumption, however, cannot be dropped. As we saw in Section 36.1, averages of Cauchy draws never settle down no matter how many samples we average, whereas the average of n standard Gaussians follows $\mathcal{N}(0, 1/n)$ and concentrates tightly around zero.

Gaussian noise and squared error. When a regression network predicts $\hat{y}$ and models the observed label as $y \sim \mathcal{N}(\hat{y}, \sigma^2)$, the negative log-probability it assigns to y is $-\log p(y) = (y-\hat{y})^2/(2\sigma^2) + \frac{1}{2}\log(2\pi\sigma^2)$. Up to a positive scale factor and an additive constant, this is the squared error loss (Section 2.1.3).

36.3.2.4 Laplace

Because squared error grows quadratically, a residual of 10 is penalized a hundred times more heavily than a residual of 1, allowing a handful of outliers to pull a fit far off course. A loss that grows only linearly, the absolute error, corresponds to a heavier-tailed noise distribution (Section 2.1.3.3).

To construct it, take a fair random sign S with $P(S=1) = P(S=-1) = \frac{1}{2}$, together with an independent exponential variable E of mean $b > 0$ (rate $1/b$), and set $X = \mu + SE$. Whenever $S = 1$, X lies above μ with $X - \mu = E$, so for $x > \mu$ the density is $\frac{1}{2} \cdot \frac{1}{b} e^{-(x-\mu)/b}$. By symmetry, the same expression holds below μ with $x-\mu$ replaced by $|x-\mu|$. This gives the *Laplace* distribution:

$$p(x) = \frac{1}{2b}\, e^{-|x-\mu|/b}, \qquad X \sim \text{Laplace}(\mu, b). \tag{36.46}$$

Each half of the distribution carries half the mass of an exponential. The density is symmetric around μ, comes to a sharp peak there, and decays exponentially on both sides.

Mean and variance. Both moments follow immediately from $X = \mu + SE$. By independence, $\mathbb{E}[SE] = \mathbb{E}[S]\,\mathbb{E}[E] = 0$, so $\mathbb{E}[X] = \mu$. And since $S^2 = 1$ always, the variance is simply the exponential's second moment, $2/\lambda^2$ with $\lambda = 1/b$:

$$\text{Var}(X) = \mathbb{E}\big[S^2 E^2\big] = \mathbb{E}[E^2] = 2b^2.$$

Laplace noise and absolute error. If a regression network models the label as $y \sim \text{Laplace}(\hat{y}, b)$, its negative log-probability is $-\log p(y) = |y-\hat{y}|/b + \log 2b$, which is the absolute error up to scaling and a constant. Used as a prior over model weights, the Laplace density yields the ℓ_1 penalty $\|\mathbf{w}\|_1$ of the LASSO (Tibshirani 1996). Moreover, when fitting a location parameter μ to data $x_1, \ldots, x_m$, maximizing the Laplace probability minimizes $\sum_i |x_i - \mu|$, which is achieved at the sample median. Whereas a single corrupted point can drag the sample mean arbitrarily far, it shifts the median by at most one rank.

Tails. Figure 36.16 compares the uniform, Gaussian, and Laplace densities scaled to have the same variance. At matched variance $\sigma^2 = 2b^2$, the probability that a Laplace variable lands more than three standard deviations from its center is $P(|X-\mu| > 3\sigma) = e^{-3\sigma/b} = e^{-3\sqrt{2}} \approx 0.014$, compared with 0.0027 for the Gaussian: roughly five times as much tail mass.

Let us verify this tail comparison directly by sampling the Laplace from its construction (a random sign times an exponential) and counting how often draws exceed three standard deviations.

```
b, m = 1.0, 200000
lap = rng.choice([-1.0, 1.0], m) * rng.exponential(b, m)   # S * E
```

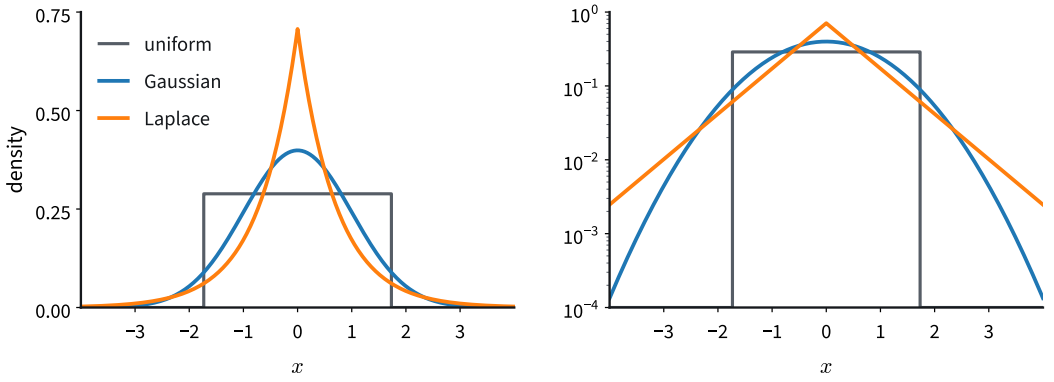


Figure 36.16: Three densities with unit variance: the uniform on $[-\sqrt{3}, \sqrt{3}]$, the standard Gaussian, and the Laplace with $b = 1/\sqrt{2}$. (a) Densities on a linear scale. (b) The same densities on a logarithmic scale, where the Gaussian is a parabola, the Laplace forms two straight lines, and the uniform is a flat plateau ending at $\pm\sqrt{3}$. Beyond three standard deviations, the Laplace carries about five times the tail probability of the Gaussian.

```
sigma = math.sqrt(2) * b                                    # matched sd
gau = rng.normal(0.0, sigma, m)
print(f'sample variances: Laplace {lap.var(ddof=1):.2f}, '
      f'Gaussian {gau.var(ddof=1):.2f}   (2b^2 = 2)')
print(f'P(|X| > 3 sd): Laplace {np.mean(np.abs(lap) > 3 * sigma):.4f}, '
      f'Gaussian {np.mean(np.abs(gau) > 3 * sigma):.4f}')
```

```
sample variances: Laplace 1.98, Gaussian 2.00   (2b^2 = 2)
P(|X| > 3 sd): Laplace 0.0140, Gaussian 0.0027
```

Both sample variances land right around 2, while the tail fractions beyond three standard deviations come out near 0.014 and 0.003.

36.3.3 The Multivariate Gaussian

In deep learning, random quantities are rarely scalars: image noise, latent representations in a generative model, and predictions across a batch of inputs are all vectors whose coordinates interact. The multivariate Gaussian is the one distribution over correlated vectors that remains analytically tractable under nearly every operation we care about. As our running example, consider a pair of coordinates with variance 2 each and covariance 1:

$$\Sigma = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}.$$

36.3.3.1 Contours and Eigenvectors

The *multivariate Gaussian* $\mathcal{N}(\mu, \Sigma)$ on $\mathbb{R}^d$ is parameterized by a mean vector μ and a symmetric, positive definite covariance matrix Σ. Its density is

$$p(\mathbf{x}) = \frac{1}{(2\pi)^{d/2}|\Sigma|^{1/2}} \exp\Big(-\tfrac{1}{2}(\mathbf{x}-\mu)^\top \Sigma^{-1}(\mathbf{x}-\mu)\Big), \tag{36.47}$$

where $|\Sigma|$ denotes the determinant of Σ. When $d = 1$, this reduces to Equation 36.44. Because $\mathbf{x}$ enters the density solely through the quadratic form in the exponent, the density is constant on the level sets where that quadratic form is constant. What geometry do these contours have?

The running example by hand. Take $\mu = \mathbf{0}$ and rotate the coordinate axes by 45° into the sum and difference coordinates $u = (x_1+x_2)/\sqrt{2}$ and $w = (x_1-x_2)/\sqrt{2}$. Their variances and covariance follow directly from the entries of Σ:

$$\mathrm{Var}(u) = \frac{2+2+2\cdot 1}{2} = 3, \qquad \mathrm{Var}(w) = \frac{2+2-2\cdot 1}{2} = 1, \qquad \mathrm{Cov}(u,w) = \frac{2-2}{2} = 0.$$

In these rotated coordinates, the covariance matrix is diagonal. The new axes are simply the eigenvectors $\mathbf{v}_1 = (1,1)/\sqrt{2}$ and $\mathbf{v}_2 = (1,-1)/\sqrt{2}$ of Σ, with eigenvalues $\gamma_1 = 3$ and $\gamma_2 = 1$ (indeed, $\Sigma\mathbf{v}_1 = (3,3)/\sqrt{2} = 3\mathbf{v}_1$).

The general case. By the spectral theorem, every symmetric matrix admits an orthonormal basis of eigenvectors, $\Sigma = \mathbf{V}\,\mathrm{diag}(\gamma_1,\dots,\gamma_d)\mathbf{V}^\top$ with $\mathbf{V}^\top\mathbf{V} = \mathbf{I}$, and positive definiteness ensures that every eigenvalue satisfies $\gamma_i > 0$. Passing to the eigen-coordinates $\mathbf{u} = \mathbf{V}^\top(\mathbf{x}-\mu)$ turns the quadratic form into a simple weighted sum of squares:

$$(\mathbf{x}-\mu)^\top\Sigma^{-1}(\mathbf{x}-\mu) = \mathbf{u}^\top\,\mathrm{diag}(\gamma_1^{-1},\dots,\gamma_d^{-1})\,\mathbf{u} = \sum_{i=1}^{d}\frac{u_i^2}{\gamma_i}.$$

The level set where this sum equals r^2 is an ellipsoid aligned with the eigenvectors $\mathbf{v}_i$, with half-axis lengths $r\sqrt{\gamma_i}$. Thus the contours of any multivariate Gaussian are ellipsoids whose principal axes are the eigenvectors of Σ, stretched in proportion to the standard deviations $\sqrt{\gamma_i}$. For our running example, they are ellipses tilted at 45° with axis ratio $\sqrt{3} : 1$ (Figure 36.17, left). Because $|\Sigma| = \prod_i \gamma_i$, the density also splits in $\mathbf{u}$ into a product of d one-dimensional Gaussians: along the eigenvector directions, the coordinates are independent with variances γ_i. In the special case $\Sigma = \sigma^2\mathbf{I}$, the contours are spheres and the original coordinates are already independent.

One caveat is worth flagging early. In high dimensions, our low-dimensional intuition of a bell concentrated near the mean can mislead: a standard Gaussian vector in $\mathbb{R}^d$ almost always has norm close to $\sqrt{d}$, and two independent draws are nearly orthogonal. We prove both facts in Section 36.9.

36.3.3.2 Linear Maps and Sampling

In practice, we routinely build new Gaussian vectors from existing ones by mixing coordinates or projecting onto a subspace. Both are linear maps, and a key reason Gaussians are so convenient is that linear maps preserve Gaussianity. We prove this first for invertible square matrices, which is all we need for sampling; rectangular projections will follow in the next subsection once we show that any subvector of a Gaussian is itself Gaussian.

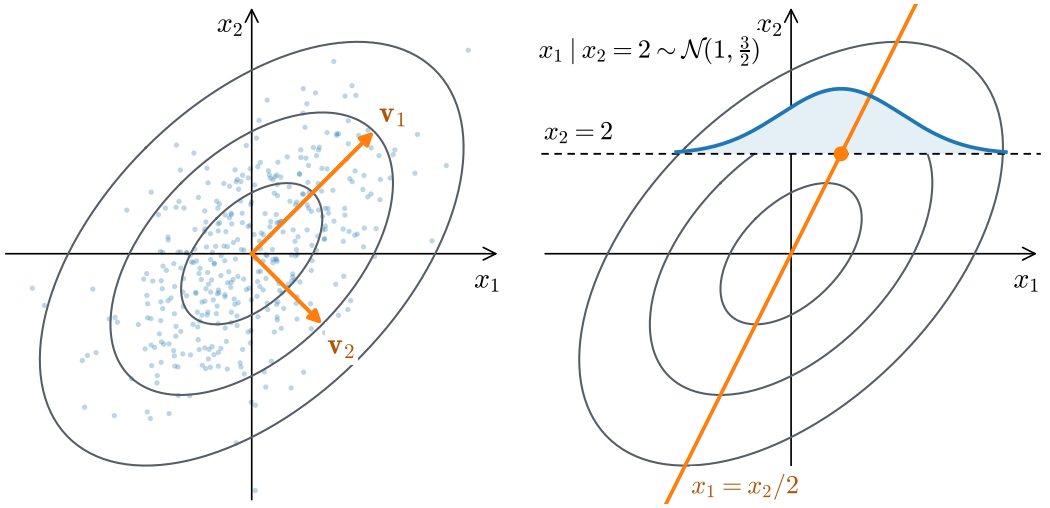


Figure 36.17: The running bivariate Gaussian with covariance Σ having rows $(2, 1)$ and $(1, 2)$. (a) 400 samples and three density contours on which the quadratic form in the exponent equals 1, 4, and 9. The ellipse axes lie along the eigenvectors of Σ with half-lengths proportional to $\sqrt{\gamma_1} = \sqrt{3}$ and $\sqrt{\gamma_2} = 1$; the arrows $\mathbf{v}_1$ and $\mathbf{v}_2$ reach the middle contour. (b) Observing $x_2 = 2$ slices the density along the dashed line, yielding the conditional density $\mathcal{N}(1, \frac{3}{2})$ for x_1. Across all slices, the conditional means lie along $x_1 = x_2/2$, and the conditional variance stays fixed at $3/2$.

Proposition (invertible linear maps). Let $\mathbf{x} \sim \mathcal{N}(\mu, \Sigma)$ in $\mathbb{R}^d$, let $\mathbf{A}$ be an invertible $d \times d$ matrix, and let $\mathbf{b} \in \mathbb{R}^d$. Then

$$\mathbf{A}\mathbf{x} + \mathbf{b} \sim \mathcal{N}\big(\mathbf{A}\mu + \mathbf{b},\ \mathbf{A}\Sigma\mathbf{A}^\top\big).$$

Proof. By linearity of expectation, the stated mean and covariance hold for any distribution with finite second moments: $\mathbb{E}[(\mathbf{A}\mathbf{x} - \mathbf{A}\mu)(\mathbf{A}\mathbf{x} - \mathbf{A}\mu)^\top]$ equals $\mathbf{A}\,\mathbb{E}[(\mathbf{x} - \mu)(\mathbf{x} - \mu)^\top]\,\mathbf{A}^\top = \mathbf{A}\Sigma\mathbf{A}^\top$. It remains only to check that the transformed density is Gaussian. By the change-of-variables formula of Section 36.2, the density of $\mathbf{y} = \mathbf{A}\mathbf{x} + \mathbf{b}$ is $p_{\mathbf{y}}(\mathbf{y}) = p_{\mathbf{x}}(\mathbf{A}^{-1}(\mathbf{y} - \mathbf{b}))/|\det \mathbf{A}|$. Setting $\mathbf{c} = \mathbf{y} - \mathbf{A}\mu - \mathbf{b}$, the exponent rewrites as $-\frac{1}{2}\mathbf{c}^\top \mathbf{A}^{-\top}\Sigma^{-1}\mathbf{A}^{-1}\mathbf{c} = -\frac{1}{2}\mathbf{c}^\top(\mathbf{A}\Sigma\mathbf{A}^\top)^{-1}\mathbf{c}$, while the normalizing factor satisfies $|\det \mathbf{A}|\,|\Sigma|^{1/2} = |\mathbf{A}\Sigma\mathbf{A}^\top|^{1/2}$. Thus $p_{\mathbf{y}}$ is the Gaussian density with the claimed mean and covariance. $\blacksquare$

Invertibility is necessary here. If $\mathbf{A} = \left(\begin{smallmatrix} 1 & 1 \\ 1 & 1 \end{smallmatrix}\right)$, the two output coordinates are always identical and $\mathbf{A}\Sigma\mathbf{A}^\top = \left(\begin{smallmatrix} 6 & 6 \\ 6 & 6 \end{smallmatrix}\right)$ is singular. The resulting random vector lives entirely along the line $y_1 = y_2$ and has no two-dimensional density in the plane.

Sampling by a matrix factor. How do we draw from $\mathcal{N}(\mu, \Sigma)$ when our random number generator only produces independent standard normals? Stack d independent draws into $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}_d)$, whose joint density is the product of d copies of ϕ. For any square matrix $\mathbf{L}$ satisfying $\mathbf{L}\mathbf{L}^\top = \Sigma$, $\mathbf{L}$ is necessarily invertible (since Σ is positive definite), and our proposition guarantees that $\mathbf{x} = \mu + \mathbf{L}\mathbf{z}$ is Gaussian with mean μ and covariance $\mathbf{L}\,\mathbf{I}\,\mathbf{L}^\top = \Sigma$.

In short, multiplying by a matrix factor turns independent noise into correlated noise. Be-

cause $\mathbf{x}$ is a smooth, differentiable function of μ and $\mathbf{L}$ for a fixed draw $\mathbf{z}$, this same construction is the *reparameterization trick* that lets variational autoencoders backpropagate through stochastic layers (Section 34.7.3). The fastest factor to compute is the *Cholesky factor*: every symmetric positive definite matrix admits a unique decomposition $\Sigma = \mathbf{L}\mathbf{L}^\top$ where $\mathbf{L}$ is lower triangular with positive diagonal entries. For our running example, matching entries of $\mathbf{L}\mathbf{L}^\top$ to Σ one by one gives $L_{11} = \sqrt{2}$, then $L_{21} = \Sigma_{21}/L_{11} = 1/\sqrt{2}$, and finally $L_{22} = \sqrt{\Sigma_{22} - L_{21}^2} = \sqrt{3/2}$; indeed, $L_{21}^2 + L_{22}^2 = \frac{1}{2} + \frac{3}{2} = 2$.

Any other square-root factor works equally well, such as the eigen-factor $\mathbf{V}\,\mathrm{diag}(\sqrt{\gamma_1}, \dots, \sqrt{\gamma_d})$. As exercise 5 shows, any two factors of the same Σ differ only by an orthogonal matrix. In practice, library samplers differ: many use the Cholesky factor, whereas NumPy's `multivariate_normal` defaults to a factor obtained from the singular value decomposition.

Rather than calling a library sampler, let us implement the Cholesky recipe directly so we can see the mechanism at work, and verify both the sample covariance and its eigenvalues.

```
Sigma = np.array([[2.0, 1.0], [1.0, 2.0]])
L = np.linalg.cholesky(Sigma)                  # Sigma = L L^T
x = rng.standard_normal((400000, 2)) @ L.T     # each row is x = L z
C = np.cov(x.T)
print('L =', L.round(3).tolist())
print('sample covariance =', C.round(2).tolist())
print('its eigenvalues   =', np.linalg.eigvalsh(C).round(2))
```

```
L = [[1.414, 0.0], [0.707, 1.225]]
sample covariance = [[1.99, 0.99], [0.99, 1.99]]
its eigenvalues   = [1.   2.99]
```

The Cholesky factor matches our hand calculation, the empirical covariance agrees with Σ to within about 0.01, and its eigenvalues sit close to 1 and 3.

36.3.3.3 Conditioning

Frequently we observe some coordinates of a Gaussian vector and want to infer the rest. In our running example, if we observe $x_2 = 2$, where should we expect x_1 to lie? Partition the vector into two blocks $\mathbf{x} = (\mathbf{x}_1, \mathbf{x}_2)$ and partition its mean and covariance conformably:

$$\mu = \begin{pmatrix} \mu_1 \\ \mu_2 \end{pmatrix}, \qquad \Sigma = \begin{pmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{pmatrix}.$$

The trick behind Gaussian conditioning is geometric and clean: subtract from $\mathbf{x}_1$ the linear prediction of $\mathbf{x}_1$ based on $\mathbf{x}_2$. What remains is a residual that carries zero information about $\mathbf{x}_2$.

Proposition (residual and marginal). Let $\mathbf{B} = \Sigma_{12}\Sigma_{22}^{-1}$. Then the residual $\mathbf{r} = \mathbf{x}_1 - \mathbf{B}\mathbf{x}_2$ is independent of $\mathbf{x}_2$, and the marginal distribution of the second block is $\mathbf{x}_2 \sim \mathcal{N}(\mu_2, \Sigma_{22})$.

Proof. Because Σ_{22} is a principal diagonal block of a positive definite matrix, it is positive definite and hence invertible. By construction, the residual is uncorrelated with $\mathbf{x}_2$: $\mathrm{Cov}(\mathbf{r}, \mathbf{x}_2) = \Sigma_{12} - \mathbf{B}\Sigma_{22} = \mathbf{0}$. The linear map $\mathbf{x} \mapsto (\mathbf{r}, \mathbf{x}_2)$ corresponds to the square block matrix $\left(\begin{smallmatrix} \mathbf{I} & -\mathbf{B} \\ \mathbf{0} & \mathbf{I} \end{smallmatrix}\right)$, whose determinant is 1, so $(\mathbf{r}, \mathbf{x}_2)$ is jointly Gaussian by our proposition on

invertible linear maps. Since its off-diagonal covariance blocks vanish ($\text{Cov}(\mathbf{r}, \mathbf{x}_2) = \mathbf{0}$), its covariance matrix is block diagonal. Consequently, the inverse covariance is block diagonal as well, the quadratic form in the exponent splits cleanly into a function of $\mathbf{r}$ plus a function of $\mathbf{x}_2$, and the determinant factors into the product of the two block determinants. The joint density therefore factors into a Gaussian density in $\mathbf{r}$ times a Gaussian density in $\mathbf{x}_2$: the two blocks are independent, and $\mathbf{x}_2$ has mean μ_2 and covariance Σ_{22}. ■

The entire proof turns on choosing $\mathbf{B}$ to make the residual uncorrelated with $\mathbf{x}_2$. As a bonus, the second claim proves that any marginal subvector of a Gaussian vector is itself Gaussian (for the first block, simply swap the block order, which is an invertible linear map). That immediately settles the rectangular linear maps we postponed earlier.

Corollary (wide matrices). If $\mathbf{A}$ is an $m \times d$ matrix of full row rank $m \le d$ and $\mathbf{b} \in \mathbb{R}^m$, then $\mathbf{Ax} + \mathbf{b} \sim \mathcal{N}(\mathbf{A}\mu + \mathbf{b}, \mathbf{A}\Sigma\mathbf{A}^\top)$. Indeed, we can augment $\mathbf{A}$ with $d - m$ additional rows to form an invertible $d \times d$ matrix; its image of $\mathbf{x}$ is Gaussian, and $\mathbf{Ax} + \mathbf{b}$ is simply the first block of that image.

Conditioning on $\mathbf{x}_2$ is now effortless. Because $\mathbf{x}_1 = \mathbf{r} + \mathbf{B}\mathbf{x}_2$ and $\mathbf{r}$ is independent of $\mathbf{x}_2$, fixing $\mathbf{x}_2$ merely shifts the Gaussian variable $\mathbf{r}$ by the known vector $\mathbf{B}\mathbf{x}_2$. The residual has mean $\mu_1 - \mathbf{B}\mu_2$ and covariance $\Sigma_{11} - \mathbf{B}\Sigma_{21} - \Sigma_{12}\mathbf{B}^\top + \mathbf{B}\Sigma_{22}\mathbf{B}^\top$, which equals $\Sigma_{11} - \Sigma_{12}\Sigma_{22}^{-1}\Sigma_{21}$, so $\mathbf{x}_1 \mid \mathbf{x}_2$ is Gaussian with parameters

$$\begin{aligned} \mu_{1|2} &= \mu_1 + \Sigma_{12}\Sigma_{22}^{-1}(\mathbf{x}_2 - \mu_2), \\ \Sigma_{1|2} &= \Sigma_{11} - \Sigma_{12}\Sigma_{22}^{-1}\Sigma_{21}. \end{aligned} \tag{36.48}$$

Two features of Equation 36.48 stand out: the conditional mean is an affine function of the observed block $\mathbf{x}_2$, and the conditional covariance does not depend on the observed value at all. In our running example, $\mathbf{B} = \frac{1}{2}$, so observing $x_2 = 2$ gives conditional mean $\frac{1}{2} \cdot 2 = 1$ and conditional variance $2 - 1 \cdot \frac{1}{2} \cdot 1 = \frac{3}{2}$. Thus $x_1 \mid x_2 = 2 \sim \mathcal{N}(1, \frac{3}{2})$, matching the vertical slice shown in Figure 36.17 (right).

Notice how $\mathbf{B}$ is the linear regression matrix predicting $\mathbf{x}_1$ from $\mathbf{x}_2$: the multiple of $\mathbf{x}_2$ whose subtraction leaves zero correlation. In a multivariate Gaussian, conditional expectation *is* linear regression read straight off the covariance matrix. Meanwhile, the conditional covariance, known in linear algebra as the *Schur complement* of Σ_{22} (Section 33.2.3.5), never exceeds Σ_{11}, because the subtracted term $\Sigma_{12}\Sigma_{22}^{-1}\Sigma_{21}$ is positive semidefinite. Observing x_2 shrinks our uncertainty about x_1 from variance 2 down to $\frac{3}{2}$. When $\mathbf{x}_2$ represents observed values of an unknown function and $\mathbf{x}_1$ its values at new test inputs, this formula becomes Gaussian process regression (Chapter 40).

A careful reader may object that zero covariance does not in general imply independence. That is true, and our proof relied on something stronger: that $(\mathbf{r}, \mathbf{x}_2)$ is *jointly* Gaussian. Without joint Gaussianity, the conclusion breaks down. For example, let $Z \sim \mathcal{N}(0, 1)$ and let S be an independent fair random sign. Then SZ is also marginally $\mathcal{N}(0, 1)$ and satisfies $\text{Cov}(SZ, Z) = \mathbb{E}[S]\,\mathbb{E}[Z^2] = 0$, yet $|SZ| = |Z|$, so the two variables are strongly dependent.

We can also check Gaussian conditioning directly on our Cholesky samples, without using any matrix formula: simply keep the samples whose x_2 falls within 0.05 of 2, inspect the distribution of their x_1 values, and check the residual $r = x_1 - \frac{1}{2}x_2$.

```
r = x[:, 0] - 0.5 * x[:, 1]                    # residual x1 - B x2, B = 1/2
print(f'Cov(r, x2) = {np.cov(r, x[:, 1])[0, 1]:.3f},  Var(r) = {r.var(ddof=1):.3f}')
near = x[np.abs(x[:, 1] - 2) < 0.05, 0]        # x1 of samples with x2 near 2
print(f'{near.size} samples with x2 near 2: mean of x1 {near.mean():.2f}, '
      f'variance {near.var(ddof=1):.2f}')
```

```
Cov(r, x2) = -0.003,  Var(r) = 1.499
4091 samples with x2 near 2: mean of x1 0.97, variance 1.52
```

As predicted, the residual has covariance near 0 with x_2 and variance near $\frac{3}{2}$. Roughly four thousand samples fall inside the slice, and their x_1 values have mean close to 1 and variance close to $\frac{3}{2}$, matching $\mathcal{N}(1, \frac{3}{2})$.

Two natural questions remain open. First, nearly all of the distributions we have met share a single algebraic template, which Section 36.4 exploits to unify their losses, gradients, and conjugate updates. Second, our opening calculation of $715/8192$ assumed that the coin was fair; inferring an unknown bias from 13 flips and quantifying our uncertainty about it is the subject of Section 36.5.

36.3.4 Summary

The table below collects the distributions introduced in this section, in the order they appeared. In the categorical and multinomial rows, each draw is represented as a one-hot vector, matching how cross-entropy loss encodes class labels.

Distribution	pmf or pdf	Mean	Variance	Model assumption
Bernoulli(p)	$p^x(1-p)^{1-x}$	p	$p(1-p)$	one yes/no outcome
Binomial(n, p)	$\binom{n}{k}p^k(1-p)^{n-k}$	np	$np(1-p)$	n fixed, independent trials, one p
Poisson(λ)	$\lambda^k e^{-\lambda}/k!$	λ	λ	many rare independent chances; mean equals variance
Categorical$(\mathbf{p})$	p_k	$\mathbf{p}$	$\mathrm{diag}(\mathbf{p}) - \mathbf{p}\mathbf{p}^\top$	one of K outcomes
Multinomial$(n, \mathbf{p})$	Equation 36.40	$n\mathbf{p}$	$n(\mathrm{diag}(\mathbf{p}) - \mathbf{p}\mathbf{p}^\top)$	n fixed, independent draws
Unif$\{1, \dots, n\}$	$1/n$	$(n+1)/2$	$(n^2-1)/12$	equally likely values
Exp(λ)	$\lambda e^{-\lambda t}$, $t \geq 0$	$1/\lambda$	$1/\lambda^2$	constant event rate (no ageing)
Gamma(α, β)	$\beta^\alpha t^{\alpha-1}e^{-\beta t}/\Gamma(\alpha)$	α/β	α/β^2	time of the α-th event (integer α)
$U(a, b)$	$1/(b-a)$ on $[a, b]$	$(a+b)/2$	$(b-a)^2/12$	equal-length intervals equally likely
$\mathcal{N}(\mu, \sigma^2)$	$\frac{1}{\sqrt{2\pi\sigma^2}}e^{-(x-\mu)^2/(2\sigma^2)}$	μ	σ^2	sum of many small independent effects, finite variance
Laplace(μ, b)	$\frac{1}{2b}e^{-\lvert x-\mu\rvert/b}$	μ	$2b^2$	symmetric, exponential tails
$\mathcal{N}(\mu, \Sigma)$	Equation 36.47	μ	Σ	jointly Gaussian coordinates

Several other important distributions appear elsewhere in this chapter: the Cauchy in Sec-

tion 36.1, the log-normal in Section 36.2, the Beta and Dirichlet in Section 36.4, and the negative binomial, geometric, and Student's t in exercises 1, 2, and 4.

A compact family of distributions models the labels and noise of most machine learning systems, and each distribution's negative log-probability yields a standard training loss. All of them grow out of a single coin flip via sums, limits, waiting times, random signs, or correlated vectors. Those constructions give us each distribution's mean, variance, and sampling algorithm in just a few lines. The central limit theorem explains why the Gaussian emerges whenever many small, independent effects with finite variance add together. Finally, multivariate Gaussians remain closed under linear transformations and conditioning, where conditioning one block on another reduces directly to linear regression.

36.3.5 Exercises

1. **Gamma–Poisson mixture.** Let the Poisson rate itself be random, $\lambda \sim \text{Gamma}(\alpha, \beta)$ with the density Equation 36.42, and let $X \mid \lambda \sim \text{Poisson}(\lambda)$.
   1. Use the law of total expectation $\mathbb{E}[X] = \mathbb{E}\big[\mathbb{E}[X \mid \lambda]\big]$ and the law of total variance
   $$\text{Var}(X) = \mathbb{E}\big[\text{Var}(X \mid \lambda)\big] + \text{Var}\big(\mathbb{E}[X \mid \lambda]\big)$$
   to compute $\mathbb{E}[X]$ and $\text{Var}(X)$. Write the variance in terms of the mean $m = \alpha/\beta$ and the shape α. In which limit of α and β does the Poisson reappear?
   2. Show that X has the *negative binomial* pmf
   $$P(X = k) = \frac{\Gamma(k+\alpha)}{k!\,\Gamma(\alpha)} \Big(\frac{\beta}{1+\beta}\Big)^{\alpha} \Big(\frac{1}{1+\beta}\Big)^{k}, \qquad k = 0, 1, 2, \dots, \tag{36.49}$$
   by integrating out λ with the help of the Gamma normalizer.
   3. A count dataset has sample mean 2 and sample variance 6. Choose α and β so that the model matches both. Which named distribution results?
2. **Converse of memorylessness.** ● The exponential distribution is memoryless. This problem proves the converse.
   1. Let $X > 0$ be a continuous random variable whose survival function $G(t) = P(X > t)$ satisfies $G(s+t) = G(s)\,G(t)$ for all $s, t \geq 0$. Show that $G(t) = e^{-\lambda t}$ for some $\lambda > 0$. Start from $G(1)$, extend the formula to rational t, and then use that G is nonincreasing. Why can $G(1)$ be neither 0 nor 1?
   2. Let X take values in $\{1, 2, \dots\}$ and satisfy $P(X > s + t \mid X > s) = P(X > t)$ for all integers $s, t \geq 0$. Show that X is geometric, $P(X > t) = (1-p)^t$ for some $p \in (0, 1]$.
3. [extended] **Burstiness of word counts.** ●● Church and Gale (1995) found that word counts in text vary across documents more than a Poisson model allows, and that Poisson mixtures such as the negative binomial fit them better. Reproduce the finding at small scale on *The Time Machine* (Section 8.2). Download the text with `d2l.download(d2l.DATA_URL + 'timemachine.txt')`, keep only letters as `d2l.TimeMachine._preprocess` does, split it into words, and cut the word sequence into consecutive chunks of $L = 500$ words, which play the role of documents.
   1. For each of the words "the", "and", "of", "time", "traveller", "weena" and "morlocks", count its occurrences in every chunk, and compute the ratio of the sample variance to the sample mean of the counts. Which ratio does a Poisson model predict? Which words deviate most from it?

2. Fit a Poisson and a negative binomial model (problem 1b) to the counts of each word by maximum likelihood, and compare the maximized log-likelihoods. Parameterize the negative binomial by its mean m and its shape α. The maximum-likelihood m is the sample mean, so only α requires a one-dimensional search, for example on a logarithmic grid.
3. For each word, compare the empirical fraction of chunks that do not contain the word, and the empirical *adaptation* $P(X \geq 2 \mid X \geq 1)$, with the predictions of both fitted models. What does the adaptation measure, and why does a Poisson model underestimate it for a name such as "traveller"?
4. Repeat (a) for chunk lengths $L \in \{100, 250, 500, 2000\}$. Assume that the rate of a word in a chunk is L times a per-word rate drawn from a Gamma distribution with a fixed shape. Use problem 1a to predict how the variance-to-mean ratio grows with L, and compare the prediction with your measurements.

4. **The Student-t as a Gamma scale mixture.** ● A Gaussian whose precision (inverse variance) is itself random has heavier tails than any Gaussian. The result is Student's t distribution, which Section 36.5 uses for the mean of a small sample. Let $\tau \sim \text{Gamma}(\nu/2, \nu/2)$ with the density Equation 36.42 (shape and rate both $\nu/2$, for some $\nu > 0$), and let $X \mid \tau \sim \mathcal{N}(\mu, 1/\tau)$.
   1. Integrate τ out to show that X has the density
   $$p(x) = \frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\sqrt{\pi\nu}\,\Gamma\left(\frac{\nu}{2}\right)} \left(1 + \frac{(x-\mu)^2}{\nu}\right)^{-(\nu+1)/2}, \tag{36.50}$$
   Student's t distribution with ν degrees of freedom. *Hint:* $\int_0^\infty \tau^{a-1} e^{-c\tau}\, d\tau = \Gamma(a)/c^a$ for $a, c > 0$.
   2. Show that $p(x)$ decays like $|x|^{-(\nu+1)}$, so that $\mathbb{E}|X-\mu|^k$ is finite only for $k < \nu$. Which distribution of Section 36.1 is the case $\nu = 1$?
   3. Show that $\text{Var}(X) = \nu/(\nu-2)$ for $\nu > 2$ by the law of total variance,
   $$\text{Var}(X) = \mathbb{E}\big[\text{Var}(X \mid \tau)\big] + \text{Var}\big(\mathbb{E}[X \mid \tau]\big).$$
   4. code Draw 10^5 samples for $\nu = 3$ and $\mu = 0$ by the mixture recipe (first τ, then X), and compare the fraction with $|X| > 3$ with the Gaussian's 0.0027. Then draw 10^4 sample means of 20 draws each and compare their standard deviation with $\sqrt{3/20}$.

5. **Two factors of one covariance.** ● Let $\Sigma = \left(\begin{smallmatrix} 2 & 1 \\ 1 & 2 \end{smallmatrix}\right)$.
   1. Compute the Cholesky factor $\mathbf{L}$ by hand, and the eigen-factor $\mathbf{M} = \mathbf{V}\,\text{diag}(\sqrt{\gamma_1}, \sqrt{\gamma_2})$ from the eigenvectors and eigenvalues found in the text. Verify $\mathbf{L}\mathbf{L}^\top = \mathbf{M}\mathbf{M}^\top = \Sigma$.
   2. Show that any two invertible factors with $\mathbf{M}_1\mathbf{M}_1^\top = \mathbf{M}_2\mathbf{M}_2^\top$ satisfy $\mathbf{M}_2 = \mathbf{M}_1\mathbf{Q}$ with $\mathbf{Q}$ orthogonal (*hint:* set $\mathbf{Q} = \mathbf{M}_1^{-1}\mathbf{M}_2$). What is the distribution of $\mathbf{Q}\mathbf{z}$ for $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$? Conclude that $\mu + \mathbf{M}_1\mathbf{z}$ and $\mu + \mathbf{M}_2\mathbf{z}$ have the same distribution.
   3. *Whitening.* Show that $\mathbf{z} = \mathbf{L}^{-1}(\mathbf{x} - \mu)$ has covariance $\mathbf{I}$ when $\mathbf{x} \sim \mathcal{N}(\mu, \Sigma)$. Explain why computing $\mathbf{z}$ for one $\mathbf{x}$ takes about d^2 operations once $\mathbf{L}$ is known, against about $d^3/3$ for computing $\mathbf{L}$.
   4. code Sample $10{,}000$ points by each factor, print both empirical covariances, whiten the Cholesky samples with a triangular solve, and print the covariance of the result.

6. **Superposition, thinning, and the order of arrivals.** ● Requests reach a server from two independent sources, which are Poisson processes with rates λ_1 and λ_2.

1. Show that the merged stream is a Poisson process with rate $\lambda_1 + \lambda_2$. Use the slicing picture, or show by the binomial theorem that the sum of independent Poisson(λ_1) and Poisson(λ_2) counts is Poisson$(\lambda_1 + \lambda_2)$.
2. Each request of a rate-λ process is logged independently with probability q. Show that the logged requests form a Poisson process with rate $q\lambda$: condition on the number of requests and sum a binomial over a Poisson. With $\lambda = 4$ per minute and $q = \frac{1}{4}$, what is the probability that a minute logs nothing?
3. Given that exactly n events fell in $[0, t]$, show that their times have the distribution of n independent $U(0, t)$ draws, sorted. Use the slicing picture: every slice is equally likely to hold an event.
4. code Simulate a rate-4 process for $1{,}000$ minutes by exponential gaps, thin it with $q = \frac{1}{4}$, and check that the thinned counts per minute have mean and variance close to 1. Collect the positions within the minute of the events in minutes with exactly three events, and check that their histogram is flat.

7. **Covariance of the multinomial.** ● Write the multinomial count vector as $\mathbf{x} = \sum_{i=1}^{n} \mathbf{e}_{Y_i}$, a sum of n independent one-hot vectors with $P(Y_i = k) = p_k$.
   1. Show that $\mathbb{E}[\mathbf{e}_Y] = \mathbf{p}$ and $\mathrm{Cov}(\mathbf{e}_Y) = \mathrm{diag}(\mathbf{p}) - \mathbf{p}\mathbf{p}^\top$, and hence that $\mathbb{E}[\mathbf{x}] = n\mathbf{p}$ and $\mathrm{Cov}(\mathbf{x}) = n(\mathrm{diag}(\mathbf{p}) - \mathbf{p}\mathbf{p}^\top)$. Read off $\mathrm{Var}(x_k) = np_k(1 - p_k)$ and $\mathrm{Cov}(x_j, x_k) = -np_jp_k$ for $j \neq k$. Why is every covariance negative?
   2. Show that $\mathrm{Cov}(\mathbf{x})\mathbf{1} = \mathbf{0}$, and explain this by the constraint $\sum_k x_k = n$.
   3. For 12 rolls of a fair die, compute $\mathrm{Var}(x_k)$, $\mathrm{Cov}(x_j, x_k)$, and the correlation between two counts.
   4. Show that x_k alone is Binomial(n, p_k) by grouping the other $K - 1$ classes into one.

36.4 The Exponential Family

Linear regression, logistic regression, and softmax regression each minimize a different loss function. Yet when we differentiate those losses with respect to the model weights, all three yield the exact same training rule: the input multiplied by the difference between the prediction and the observation. When three seemingly separate derivations converge on one formula, there is usually a single structure underneath. The catalogue of Section 36.3, where each distribution's mean and variance seemed to require its own clever algebraic trick, points to the same suspicion.

Consider the coin that runs through this chapter, which landed heads 9 times in 13 flips. The natural estimate of its bias is $9/13 \approx 0.69$. If we fit a logistic-regression model with only a bias term to those same 13 flips, it arrives at $9/13$ by an indirect route: its final layer computes the log-odds $\log(9/4) \approx 0.81$, and a sigmoid maps that number back to $9/13$. Why take this detour through the log-odds instead of working directly with the probability?

One idea explains both puzzles. Nearly every distribution in our catalogue can be written in a single canonical form, where the parameters enter only through the exponential of a linear combination of a few data statistics. Distributions with this structure form the *exponential family*, and for a coin, the parameter sitting in the exponent is precisely the log-odds. Remarkably, four central properties follow from this form alone: the mean and variance are simply the first two derivatives of a single function; maximum likelihood becomes a convex optimization problem whose gradient is always prediction minus observation; the softmax emerges automatically when we model class labels; and a prior of the same form updates

by adding counts.

For deep learning, the second property is the central result, and the first is what proves it (Section 36.4.3.2 and Section 36.4.3.3). Together, they tell us which loss function to pair with a new kind of label, such as a count, and when a maximum-likelihood fit has no finite solution at all, as happens when a coin lands heads on every single flip. Two parts of this section can be skipped on a first reading: the maximum-entropy derivation (Section 36.4.2), which explains where the exponential form comes from, and the alternative derivations of the softmax from noisy choices and statistical physics.

36.4.1 One Form for Many Distributions

Let us start with the simplest distribution in our catalogue, a single coin flip, and rewrite it until the log-odds appears on its own. That small bit of algebra will show us what the general form has to look like.

36.4.1.1 The Bernoulli in Exponential Form

A coin with bias p lands heads ($x = 1$) with probability p and tails ($x = 0$) with probability $1 - p$. In the previous section we wrote both cases in one expression, $p^x(1-p)^{1-x}$. To see how the outcome x and the parameter p interact, take the logarithm and group the terms that contain x:

$$\log P(X = x) = x \log p + (1 - x) \log(1 - p) = x \log \frac{p}{1-p} + \log(1 - p).$$

Notice how cleanly the two roles separate: the outcome enters only through x multiplied by the log-odds $\log \frac{p}{1-p}$, while the second term depends on p alone. Let us give the log-odds a name, $\eta = \log \frac{p}{1-p}$, and rewrite the second term in terms of η. Solving for p yields $p = e^\eta/(1 + e^\eta)$, so $1 - p = 1/(1 + e^\eta)$ and $\log(1 - p) = -\log(1 + e^\eta)$. Exponentiating both sides gives

$$P(X = x) = \exp\big(x\eta - \log(1 + e^\eta)\big), \qquad x \in \{0, 1\}.$$

The coin's probability is now the exponential of a linear function of x, minus a term that does not depend on x at all. That second term is simply the normalizer: the two unnormalized weights are $e^0 = 1$ and e^η, and they sum to $1 + e^\eta$.

Three ingredients from this expression will reappear throughout the section. The *statistic* $T(x) = x$ is the part of the outcome that matters. The *natural parameter* η multiplies it; for the Bernoulli distribution, this is the log-odds, also known as the *logit*. Finally, the *log-partition function* $A(\eta) = \log(1+e^\eta)$ is the logarithm of the normalizer, a function you may recognize as the *softplus*. Reading our earlier substitution in reverse,

$$p = \frac{e^\eta}{1 + e^\eta} = \text{sigmoid}(\eta), \qquad \text{sigmoid}(\eta) = \frac{1}{1 + e^{-\eta}}.$$

The sigmoid of logistic regression is therefore nothing more than the map from the coin's natural parameter back to its mean. Nobody had to invent it.

For our running coin example, the empirical bias $9/13$ corresponds to the natural parameter

$$\hat{\eta} = \log \frac{9/13}{4/13} = \log \frac{9}{4} \approx 0.811,$$

and the sigmoid takes it right back: $\text{sigmoid}(\log \frac{9}{4}) = \frac{9/4}{1+9/4} = \frac{9}{13}$. Shortly, we will see why $\hat{\eta}$ is the maximum-likelihood value of the log-odds, and what the curvature of the loss around $\hat{\eta}$ tells us about how uncertain that estimate is.

36.4.1.2 Definition and Members

Our rewriting of the Bernoulli had three moving parts: a statistic of the data, a parameter that multiplies it, and a normalizer. Generalizing to several statistics at once, while allowing a fixed, parameter-free weight on each outcome, gives a single template that captures most of our catalogue.

A family of distributions is an *exponential family* if its probability mass or density function can be written as

$$p(\mathbf{x} \mid \eta) = h(\mathbf{x}) \, \exp\big(\eta^\top T(\mathbf{x}) - A(\eta)\big). \tag{36.51}$$

In this form, the data and the parameters interact only through the inner product $\eta^\top T(\mathbf{x})$. The *base measure* $h(\mathbf{x}) \geq 0$ assigns a fixed weight to each outcome independently of the parameters, and the region where $h > 0$ defines the support of the distribution. The *natural parameters* $\eta \in \mathbb{R}^d$ are the adjustable parameters, and the *sufficient statistics* $T(\mathbf{x}) \in \mathbb{R}^d$ are the features of the data they multiply. The *log-partition function* is the logarithm of the normalizing constant,

$$A(\eta) = \log \int h(\mathbf{x}) \, \exp\big(\eta^\top T(\mathbf{x})\big) \, d\mathbf{x}, \tag{36.52}$$

with an integral for continuous densities and a sum for discrete $\mathbf{x}$. In plain terms, A is whatever constant makes Equation 36.51 integrate or sum to one, and the valid natural parameters are those for which $A(\eta)$ is finite. The Bernoulli distribution is the special case $h(x) = 1$, $T(x) = x$, and $A(\eta) = \log(1 + e^\eta)$.

Why are the features $T(\mathbf{x})$ called *sufficient*? Because they hold everything the data can tell us about η. For n independent observations, the joint likelihood factors as

$$\prod_{i=1}^{n} p(\mathbf{x}_i \mid \eta) = \Big(\prod_{i=1}^{n} h(\mathbf{x}_i)\Big) \exp\Big(\eta^\top \sum_{i=1}^{n} T(\mathbf{x}_i) - nA(\eta)\Big).$$

The parameters touch the data only through the sum $\sum_i T(\mathbf{x}_i)$. Any two datasets that share the same sum produce likelihoods differing only by a constant factor independent of η, so they yield the same parameter estimates. For a coin, this sum is simply the total number of heads. Any permutation of the 13 flips has the same probability under every bias, so once we know that 9 flips came up heads, the order in which they appeared adds nothing further about the coin.

Matching pieces against Equation 36.51 quickly identifies the other members of the family. For a Poisson distribution with rate λ, the probability of k events is $\frac{\lambda^k e^{-\lambda}}{k!} = \frac{1}{k!}\exp(k\log\lambda - \lambda)$, so we read off $h(k) = 1/k!$, $T(k) = k$, $\eta = \log\lambda$, and $A(\eta) = e^{\eta} = \lambda$. A Gaussian with unknown mean and variance needs two statistics. Expanding the square in its exponent and collecting the terms in x and x^2 gives

$$\frac{1}{\sqrt{2\pi}}\exp\Big(\underbrace{\tfrac{\mu}{\sigma^2}}_{\eta_1}\,x + \underbrace{\left(-\tfrac{1}{2\sigma^2}\right)}_{\eta_2}\,x^2 - \underbrace{\left(\tfrac{\mu^2}{2\sigma^2} + \log\sigma\right)}_{A(\eta)}\Big),$$

so $T(x) = (x, x^2)^{\top}$ and $h(x) = 1/\sqrt{2\pi}$. Notice that the second natural parameter must be strictly negative: when $\eta_2 < 0$, the factor $e^{\eta_2 x^2}$ supplies the Gaussian's rapid tail decay, whereas for $\eta_2 \geq 0$ the normalizing integral for A diverges. Table 36.1 collects these and several other common members.

Table 36.1: Members of the exponential family. In each row the probability or density is $h(\mathbf{x})\exp(\eta^{\top}T(\mathbf{x}) - A(\eta))$; the binomial requires a fixed number of trials n and the Laplace a fixed location μ.

distribution	$h(\mathbf{x})$	$T(\mathbf{x})$	η	$A(\eta)$
Bernoulli(p)	1	x	$\log\frac{p}{1-p}$	$\log(1+e^{\eta})$
Binomial(n, p)	$\binom{n}{x}$	x	$\log\frac{p}{1-p}$	$n\log(1+e^{\eta})$
Poisson(λ)	$1/k!$	k	$\log\lambda$	e^{η}
Categorical, one-hot $\mathbf{y}$	1	$\mathbf{y}$	logits $\mathbf{z}$	$\log\sum_k e^{z_k}$
Exponential(λ)	1 for $x \geq 0$	x	$-\lambda$	$-\log(-\eta)$
$\mathcal{N}(\mu, \sigma^2)$	$1/\sqrt{2\pi}$	(x, x^2)	$(\frac{\mu}{\sigma^2}, -\frac{1}{2\sigma^2})$	$-\frac{\eta_1^2}{4\eta_2} - \frac{1}{2}\log(-2\eta_2)$
Laplace(μ, b)	1	$\lvert x-\mu\rvert$	$-1/b$	$\log(-2/\eta)$

The categorical row expresses class probabilities through the logits, giving rise to the softmax, whose origins we explore in Section 36.4.4. Two further members live in higher dimensions: the multivariate Gaussian, with sufficient statistics $\mathbf{x}$ and $\mathbf{x}\mathbf{x}^{\top}$ and natural parameters $\Sigma^{-1}\mu$ and $-\frac{1}{2}\Sigma^{-1}$, and the multinomial with a fixed number of draws. Keep in mind that the split among h, η, and T is not unique. The one redundancy that will matter later sits in the categorical row: because the K entries of a one-hot vector always sum to one, one of the K logits is redundant (Section 36.4.3.2).

36.4.1.3 Distributions Outside the Family

Not every distribution fits inside Equation 36.51, and the two classic exceptions fail for illuminatingly different reasons.

First, consider the uniform distribution $U(0, \theta)$, whose density is $1/\theta$ on $[0, \theta]$ and zero outside. Here the support itself moves with the parameter θ. By contrast, every exponential family has a fixed support, the set where $h > 0$, because the exponential factor never vanishes. A moving support changes estimation fundamentally. Suppose we observe three draws: 0.1, 0.2, and 0.9. The likelihood is θ^{-3} whenever $\theta \geq 0.9$ and 0 whenever $\theta < 0.9$

(a smaller θ could never have generated 0.9), so the maximum-likelihood estimate is the sample maximum, $\hat{\theta} = \max_i x_i = 0.9$.

Inside an exponential family, as we will prove shortly, maximum likelihood always equates the model's mean of the sufficient statistic to its sample average. Applying that recipe here sets the mean $\theta/2$ equal to $\bar{x} = 0.4$, yielding $\theta = 0.8$, a value under which the observed draw 0.9 is impossible. Because the uniform likelihood depends on the data through their maximum rather than a sum, the machinery of this section does not apply. Exercise 7 in Section 36.6 compares how these two estimators behave as the sample size grows.

Heavy-tailed distributions fall outside the family for a second reason. The *Student-t* distribution arises by taking a Gaussian whose precision $1/\sigma^2$ is itself random; averaging over that precision replaces the rapidly decaying e^{-x^2} tails with polynomial ones (Exercise 4 derives its density). With unit scale and ν degrees of freedom, the likelihood of n observations is a product of factors $\left(1 + (x_i - \mu)^2/\nu\right)^{-(\nu+1)/2}$, which cannot be compressed into any fixed number of sums of the data. In fact, a classical theorem of Pitman, Koopman, and Darmois states that among regular families with fixed support, only exponential families possess a sufficient statistic of fixed dimension (Darmois 1935; Koopman 1936; Pitman 1936).

As a practical consequence, the convexity of the negative log-likelihood, which we will prove holds for every exponential family, can break down outside it. For a Student-t distribution, when the data fall into two well-separated clusters, the negative log-likelihood for the location μ develops a separate local minimum near each cluster.

36.4.2 Maximum Entropy

So far, the exponential form has appeared simply by rewriting familiar distributions. A deeper question asks where this form comes from in the first place. Suppose all we know about a random quantity is the average value of a few of its statistics. Which distribution should we choose? Remarkably, the distribution that assumes nothing beyond those averages is always a member of the exponential family.

36.4.2.1 Entropy and a Loaded Die

To make "assumes nothing beyond the averages" precise, we need a way to measure how spread out, or uncommitted, a distribution is. For a distribution $p = (p_1, \dots, p_K)$ on a finite set, the *entropy* is

$$H[p] = -\sum_{k=1}^{K} p_k \log p_k,$$

measured in nats, with the convention $0 \log 0 = 0$. Each term weights the surprise $-\log p_k$ of outcome k by its probability, so entropy is simply the average surprise. A fair six-sided die assigns every face a surprise of $\log 6$, giving an entropy of $\log 6 \approx 1.79$ nats. A die that always lands on six holds no surprises at all and has entropy 0. Among all distributions on K outcomes, the uniform distribution achieves the largest possible entropy, $\log K$, a fact we will prove in a moment.

E. T. Jaynes framed the problem with a concrete puzzle (Jaynes 1957). Suppose someone tells you that a six-sided die averages 4.5 rather than 3.5, and nothing else. Many distributions on $\{1, \dots, 6\}$ have mean 4.5. Putting probability $1/2$ on 3 and $1/2$ on 6 is one of them,

yet it boldly asserts that 1, 2, 4, and 5 can never occur, a claim with no support in what we were told. Its entropy is only $\log 2 \approx 0.69$.

Jaynes's *principle of maximum entropy* selects, among all distributions matching the known averages, the one with the largest entropy, the least committal choice consistent with our knowledge. Let us state the answer before proving it. The unique maximizer is an *exponential tilt* of the uniform distribution,

$$p_k = \frac{e^{\eta k}}{\sum_{j=1}^{6} e^{\eta j}}, \qquad k = 1, \dots, 6,$$

with η tuned so that the mean equals 4.5. The distribution concentrated on threes and sixes surrendered $\log 6 - \log 2 \approx 1.10$ of the fair die's 1.79 nats to hit this mean. Before running the code below, pause and guess how much entropy the exponential tilt has to give up. Because the mean of the tilt increases monotonically with η, simple bisection finds the exact value of η.

```
k = np.arange(1, 7)
def tilt(eta):                                  # p_k proportional to exp(eta k)
    w = np.exp(eta * k)
    return w / w.sum()

lo, hi = -5.0, 5.0
for _ in range(50):                             # bisection: the mean increases with eta
    eta = (lo + hi) / 2
    lo, hi = (eta, hi) if (tilt(eta) * k).sum() < 4.5 else (lo, eta)
p = tilt(eta)
print('eta =', round(eta, 3), ' p =', p.round(3))
print('H =', round(-(p * np.log(p)).sum(), 3), ' log 6 =', round(np.log(6), 3))
```

```
eta = 0.371  p = [0.054 0.079 0.114 0.165 0.24  0.347]
H = 1.614  log 6 = 1.792
```

The resulting distribution is geometric across the faces: each face is $e^{0.371} \approx 1.45$ times as likely as the face below it. Its entropy is 1.61 nats, giving up only 0.18 nats compared with the fair die, and setting $\eta = 0$ recovers the fair die. Figure 36.18 compares the exponential tilts for three different required means.

Notice that this tilt already has the exponential-family form Equation 36.51, with $h \equiv 1$ on the six faces, statistic $T(k) = k$, and natural parameter η. Fixing the mean of a statistic and maximizing entropy led us straight to the exponential of a multiple of that statistic. That is no coincidence of the die.

36.4.2.2 The Maximum-Entropy Proposition

To state the general theorem, we need one small extension of entropy, because some outcome spaces carry natural baseline weights. For a distribution q and a base measure h, the *entropy relative to* h is

$$H_h[q] = -\sum_{\mathbf{x}} q(\mathbf{x}) \log \frac{q(\mathbf{x})}{h(\mathbf{x})}$$

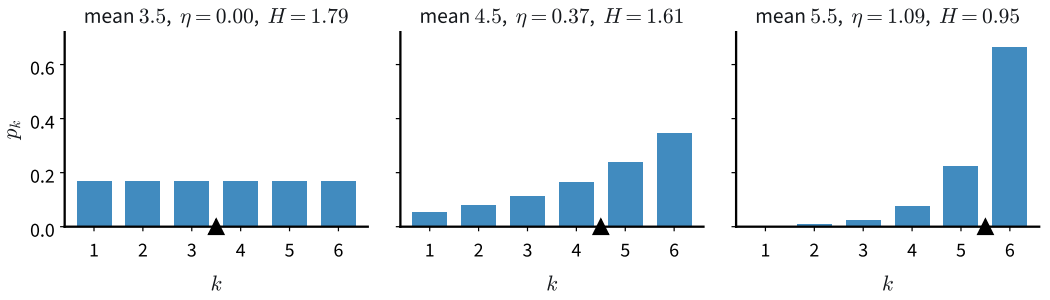


Figure 36.18: Maximum-entropy distributions of a die whose average roll is fixed at 3.5, 4.5, and 5.5 (black triangles). Each is an exponential tilt $p_k \propto e^{\eta k}$ of the uniform distribution; as the required mean rises, the tilt η grows and the entropy H falls.

for discrete outcomes, with an integral in place of the sum for continuous densities. Setting $h \equiv 1$ recovers the ordinary entropy defined above.

The proof compares any candidate distribution q directly against a member p of the exponential family. Given two distributions p and q on the same space, with $p(\mathbf{x}) > 0$ wherever $q(\mathbf{x}) > 0$, the *Kullback–Leibler divergence* of q from p is $D_{\mathrm{KL}}(q\|p) = \sum q \log(q/p)$ summed over outcomes with $q > 0$, or the corresponding integral for densities (Cover and Thomas 1999). Section 37.1 studies entropy and this divergence in depth. All we need here is one elementary fact, proved by a single tangent line: because the logarithm is concave, $\log u \le u - 1$ for every $u > 0$, with equality only at $u = 1$ (Figure 36.19).

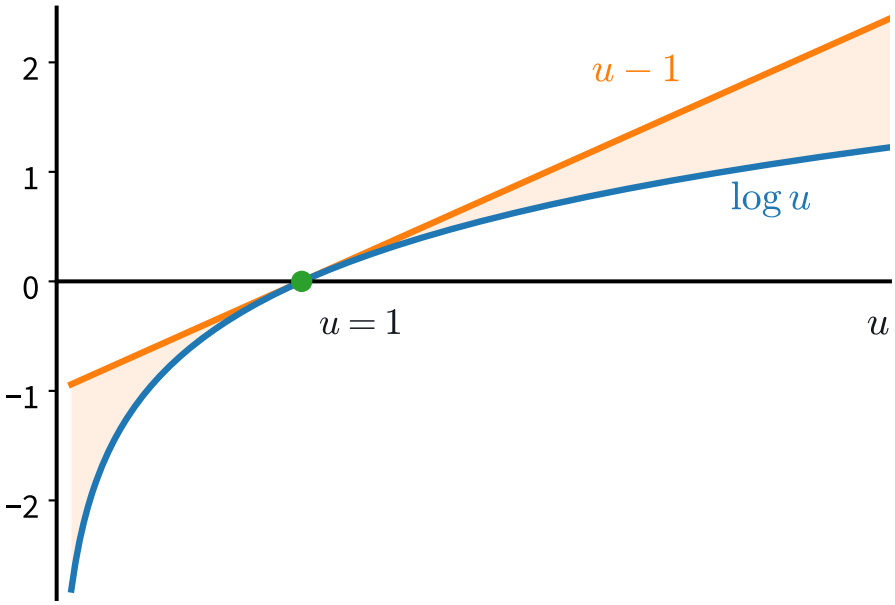


Figure 36.19: The tangent-line bound $\log u \le u - 1$. The concave logarithm (blue) lies below its tangent at $u = 1$ (orange) for every $u > 0$ and touches it only there.

Lemma (Gibbs' inequality). *The Kullback–Leibler divergence is never negative,*

$$D_{\mathrm{KL}}(q\|p) = \sum_{\mathbf{x}:\, q(\mathbf{x})>0} q(\mathbf{x}) \log \frac{q(\mathbf{x})}{p(\mathbf{x})} \geq 0, \tag{36.53}$$

with equality if and only if $q = p$.

Proof. Apply the tangent-line bound $\log u \leq u - 1$ with $u = p(\mathbf{x})/q(\mathbf{x})$ and sum over all outcomes with $q(\mathbf{x}) > 0$:

$$-D_{\mathrm{KL}}(q\|p) = \sum_{q>0} q \log \frac{p}{q} \leq \sum_{q>0} q\Big(\frac{p}{q} - 1\Big) = \sum_{q>0} p - 1 \leq 0.$$

Equality in the first inequality requires $p = q$ wherever $q > 0$. That forces $\sum_{q>0} p = 1$, leaving p with no remaining probability mass outside the support of q, so $p = q$ everywhere. ■

Proposition (maximum entropy). *Let $p(\mathbf{x}) = h(\mathbf{x}) \exp(\eta^\top T(\mathbf{x}) - A(\eta))$ be a member of an exponential family, and let q be any distribution (with a density, in the continuous case) that vanishes wherever h vanishes, has a finite mean statistic $\mathbb{E}_q[T(\mathbf{x})]$, and matches p on that mean: $\mathbb{E}_q[T(\mathbf{x})] = \mathbb{E}_p[T(\mathbf{x})]$. Then $H_h[q] \leq H_h[p]$, with equality if and only if $q = p$.*

Proof. Split the logarithm in $H_h[q]$ by inserting p: $\log \frac{q}{h} = \log \frac{q}{p} + \log \frac{p}{h}$. The first term yields $-D_{\mathrm{KL}}(q\|p) \leq 0$ by Gibbs' inequality, which applies because $p > 0$ wherever $h > 0$. The second term is linear in the sufficient statistic, $\log \frac{p}{h} = \eta^\top T - A(\eta)$, so q enters only through its mean of T:

$$-\sum_{\mathbf{x}} q \log \frac{p}{h} = -\eta^\top \mathbb{E}_q[T] + A(\eta) = -\eta^\top \mathbb{E}_p[T] + A(\eta) = H_h[p].$$

The middle equality uses the mean constraint, and the last repeats the same calculation with p in place of q. Combining the two terms gives $H_h[q] = H_h[p] - D_{\mathrm{KL}}(q\|p) \leq H_h[p]$, with equality only when $q = p$. ■

The key step is the second one: because $\log(p/h)$ is linear in T, the cross term depends on q solely through $\mathbb{E}_q[T]$, which the constraint holds fixed. In plain words, *whenever a member of the exponential family satisfies the mean constraints, it is the unique distribution of largest entropy among all distributions that satisfy them*. Remarkably, no Lagrange multipliers are needed, and the proof certifies a global maximum rather than merely a stationary point.

For the loaded die, the tilt with $\eta \approx 0.371$ belongs to the family with $h \equiv 1$ and $T(k) = k$ and has mean 4.5, so it is the unique maximizer. Three familiar distributions follow immediately as corollaries:

- *No constraint* ($T \equiv 0$): every q is admissible, and the maximizer is $p \propto h$, provided h has a finite sum. On a finite set with $h \equiv 1$, this is the uniform distribution with entropy $\log K$; the fair die is the case $K = 6$.
- *Mean and variance on $\mathbb{R}$*: with $T(x) = (x, x^2)$ and constant h, the maximizer is the Gaussian. Because a constant h shifts H_h by a fixed constant, the Gaussian has the largest entropy among all densities with a prescribed mean and variance.

- *Mean on the counts* $\{0, 1, 2, \dots\}$ with $T(k) = k$: taking $h(k) = 1/k!$ makes the maximizer of H_h with mean m the Poisson distribution with rate $\lambda = m$, whereas taking $h \equiv 1$ yields the geometric distribution $p_k = (1 - r)r^k$ with $r = m/(1 + m)$. The base measure decides which distribution emerges, as Exercise 2 explores.

Be careful not to drop the hypothesis that some member of the family meets the constraints. Suppose we fix only the mean m on all of $\mathbb{R}$, with $h \equiv 1$. The candidate density $e^{\eta x}$ is not integrable on $\mathbb{R}$ for any η, so no member exists. Nor does any maximizer exist at all: the Gaussian $\mathcal{N}(m, s^2)$ has mean m and entropy $\frac{1}{2}\log(2\pi e s^2)$, which grows without bound as $s \to \infty$.

36.4.3 The Log-Partition Function

In the catalogue of Section 36.3, every mean and variance seemed to demand its own trick: $X^2 = X$ for the Bernoulli, a sum of independent flips for the binomial, an index shift in the series for the Poisson, and integration by parts for the Gaussian. Inside the exponential family, all of those calculations collapse into one operation: differentiating the log-partition function A. The first derivative gives the mean, the second gives the variance, and the very same function governs maximum likelihood.

36.4.3.1 The Moment Property

Recall that $A(\eta)$ is the logarithm of an integral or sum of exponentials $\exp(\eta^\top T(\mathbf{x}))$. Differentiating such a logarithm pulls down a factor of $T(\mathbf{x})$, weighted by the probability of $\mathbf{x}$, which is simply the expectation of T.

Proposition (mean of the sufficient statistics). *For an exponential-family distribution Equation 36.51,*

$$\nabla_\eta A(\eta) = \mathbb{E}[T(\mathbf{x})]. \tag{36.54}$$

Proof. Differentiate Equation 36.52 under the integral sign. This interchange is justified in the interior of the set of natural parameters where $A(\eta) < \infty$ (Folland 1999); that set is convex (by Hölder's inequality), and we take η in its interior from here on. Writing $Z(\eta) = e^{A(\eta)} = \int h(\mathbf{x}) e^{\eta^\top T(\mathbf{x})} d\mathbf{x}$, the chain rule gives $\nabla A = (\nabla Z)/Z$, where

$$\nabla_\eta Z = \int h(\mathbf{x})\, T(\mathbf{x})\, e^{\eta^\top T(\mathbf{x})}\, d\mathbf{x}.$$

Dividing by $Z = e^{A}$ brings the factor e^{-A} inside the integral, completing the density Equation 36.51:

$$\begin{aligned} \nabla_\eta A &= \frac{1}{Z} \int h(\mathbf{x})\, T(\mathbf{x})\, e^{\eta^\top T(\mathbf{x})}\, d\mathbf{x} \\ &= \int T(\mathbf{x}) \underbrace{h(\mathbf{x})\, e^{\eta^\top T(\mathbf{x}) - A(\eta)}}_{p(\mathbf{x}\mid\eta)}\, d\mathbf{x} = \mathbb{E}[T(\mathbf{x})]. \quad \blacksquare \end{aligned}$$

Differentiating $\nabla A = (\nabla Z)/Z$ a second time, under the same interchange, produces the covariance matrix of the sufficient statistics:

$$\nabla^2 A = \frac{\nabla^2 Z}{Z} - \frac{(\nabla Z)(\nabla Z)^\top}{Z^2} = \mathbb{E}[T\,T^\top] - \mathbb{E}[T]\,\mathbb{E}[T]^\top = \mathrm{Cov}(T).$$

In short, the slope of A is the mean of T, and its curvature is the covariance of T. Because every covariance matrix is positive semidefinite ($\mathbf{v}^\top \mathrm{Cov}(T)\mathbf{v} = \mathrm{Var}(\mathbf{v}^\top T) \geq 0$ for all $\mathbf{v}$; Section 33.3.2.2), the log-partition function A is always convex.

Let us check this on two familiar examples. For the coin, $A(\eta) = \log(1+e^\eta)$ gives $A'(\eta) = \mathrm{sigmoid}(\eta) = p$, the Bernoulli mean, and $A''(\eta) = \mathrm{sigmoid}(\eta)(1-\mathrm{sigmoid}(\eta)) = p(1-p)$, its variance. At $\hat{\eta} = \log\frac{9}{4}$, these evaluate to $9/13 \approx 0.692$ and $\frac{9}{13}\cdot\frac{4}{13} = \frac{36}{169} \approx 0.213$. For the Poisson distribution, $A(\eta) = e^\eta$ is its own derivative, so at $\lambda = 4$ ($\eta = \log 4$) both A' and A'' equal 4. Since moments are just derivatives, automatic differentiation computes them directly for any A we can write in code. Let us verify both cases.

```
def derivatives(A, eta):
    """First and second derivative of the scalar function A at eta."""
    eta = torch.tensor(eta, dtype=torch.float64, requires_grad=True)
    (dA,) = torch.autograd.grad(A(eta), eta, create_graph=True)
    (d2A,) = torch.autograd.grad(dA, eta)
    return dA.item(), d2A.item()

cases = [('Bernoulli', torch.nn.functional.softplus, math.log(9 / 4), 9 / 13, 36 / 169),
         ('Poisson', torch.exp, math.log(4), 4, 4)]
for name, A, eta, mean, var in cases:
    dA, d2A = derivatives(A, eta)
    print(f"{name:9s} A' = {dA:.4f} (mean {mean:.4f})  A'' = {d2A:.4f} (variance {var:.4f})")
```

```
Bernoulli A' = 0.6923 (mean 0.6923)  A'' = 0.2130 (variance 0.2130)
Poisson   A' = 4.0000 (mean 4.0000)  A'' = 4.0000 (variance 4.0000)
```

Both derivatives match the analytical mean and variance to all four printed digits. Notice how the Poisson's famous equality of mean and variance, which required a power-series calculation in the catalogue, now reduces to a single line: $A = A' = A''$ for $A(\eta) = e^\eta$.

36.4.3.2 Maximum Likelihood as Moment Matching

The moment property turns maximum likelihood into a remarkably clean condition on expectations: the fitted model must reproduce the sample average of the sufficient statistics. Given independent observations $\mathbf{x}_1, \dots, \mathbf{x}_n$, taking the negative logarithm of the product likelihood from Equation 36.51 gives

$$-\sum_{i=1}^{n} \log p(\mathbf{x}_i \mid \eta) = n\,A(\eta) - \eta^\top \sum_{i=1}^{n} T(\mathbf{x}_i) - \sum_{i=1}^{n} \log h(\mathbf{x}_i). \tag{36.55}$$

Here the data enter solely through the sum $\sum_{i=1}^{n} T(\mathbf{x}_i)$. Because A is convex and the second term is linear in η, the negative log-likelihood is convex in η. Setting its gradient to zero and applying the moment property Equation 36.54 yields

$$\nabla A(\hat{\eta}) = \bar{T}, \qquad \bar{T} = \frac{1}{n}\sum_{i=1}^{n} T(\mathbf{x}_i).$$

Maximum likelihood therefore sets the model's expected sufficient statistic equal to its observed sample average, a principle called *moment matching*. And because the objective is convex, any solution to this equation is automatically a global minimizer.

Let us see what this gives for our coin with 9 heads in 13 flips. Its negative log-likelihood is $13A(\eta) - 9\eta = 13\log(1+e^{\eta}) - 9\eta$. The derivative $13\,\mathrm{sigmoid}(\eta) - 9$ vanishes precisely where $\mathrm{sigmoid}(\hat{\eta}) = 9/13$, which gives $\hat{\eta} = \log\frac{9}{4} \approx 0.811$. At this minimum, the loss equals

$$13\log\frac{13}{4} - 9\log\frac{9}{4} \approx 8.02 \text{ nats},$$

which is 13 times the entropy of a coin with bias $9/13$, $-\frac{9}{13}\log\frac{9}{13} - \frac{4}{13}\log\frac{4}{13} \approx 0.617$. At the optimum, the average loss per flip equals the entropy of the fitted coin.

This identity holds for every exponential family with $h \equiv 1$: the average negative log-likelihood at $\hat{\eta}$ is $A(\hat{\eta}) - \hat{\eta}^{\top}\bar{T}$, while the entropy of the fitted distribution is $A(\hat{\eta}) - \hat{\eta}^{\top}\nabla A(\hat{\eta})$. Because $\nabla A(\hat{\eta}) = \bar{T}$, the two numbers coincide.

The second derivative of the loss at the minimum is $13A''(\hat{\eta}) = 13 \cdot \frac{36}{169} = \frac{36}{13} \approx 2.77$. Large curvature means the loss climbs steeply away from $\hat{\eta}$, pinning down the log-odds tightly. Thirteen flips leave plenty of wiggle room: moving by $1/\sqrt{2.77} \approx 0.60$ away from $\hat{\eta} \approx 0.81$ raises the loss by only about half a nat. Section 36.6 identifies this curvature as the Fisher information and turns it into an error bar. Figure 36.20 illustrates both the tangent of A at $\hat{\eta}$ and the shape of the negative log-likelihood.

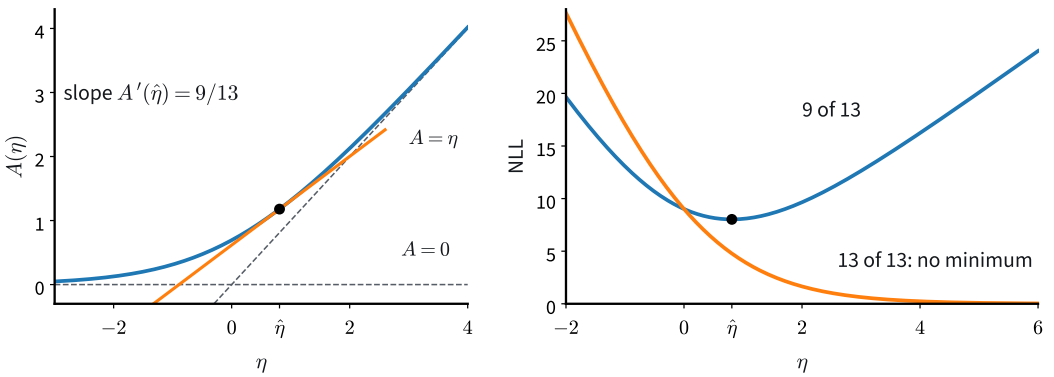


Figure 36.20: Left: the Bernoulli log-partition function $A(\eta) = \log(1+e^{\eta})$ with its asymptotes and its tangent at $\hat{\eta} = \log(9/4)$, whose slope is the mean $9/13$. Right: the negative log-likelihood $13A(\eta) - x\eta$ for $x = 9$ heads in 13 flips, with its minimum at $\hat{\eta}$, and for $x = 13$ heads, which decreases for every η and has no finite minimizer.

There is an important caveat: the moment-matching equation $\nabla A(\hat{\eta}) = \bar{T}$ has a solution only if $\bar{T}$ is a mean that some member of the family can actually attain. For a Bernoulli

coin, the attainable means are sigmoid(η) for real η, namely the open interval $(0, 1)$. If all 13 flips come up heads, then $\bar{T} = 1$ lies on the boundary and is not attainable. The negative log-likelihood becomes $13A(\eta) - 13\eta = 13\log(1 + e^{-\eta})$, which drops monotonically for every η: it equals 9.01 at $\eta = 0$, 0.087 at $\eta = 5$, and 0.0006 at $\eta = 10$. It approaches 0 without ever reaching it, so no finite minimizer exists; the fitted probability approaches 1 as $\widehat{\eta} \to \infty$.

Logistic regression hits the exact same wall whenever a hyperplane separates the two classes in the training set: the loss keeps decreasing as the weights scale up, driving them toward infinity. Adding a prior on the parameters (which we introduce at the end of this section) or using weight decay restores a finite optimum; Exercise 4 in Section 36.6 works out the details.

When a minimizer does exist, it is unique whenever the negative log-likelihood is strictly convex. That holds if $\mathrm{Cov}(T)$ is positive definite, meaning no nontrivial linear combination of the sufficient statistics is constant across the support. Such a parameterization is called *minimal*. The categorical distribution parameterized by all K logits is not minimal: the K one-hot coordinates always sum to one, so adding the same constant to every logit leaves all probabilities unchanged. As a result, its negative log-likelihood is flat along the direction $\mathbf{1}$, and its minimizers form a line rather than a point. Deep learning libraries keep all K logits anyway and exploit this shift invariance to subtract the largest logit before exponentiating for numerical stability (Section 3.3.5).

36.4.3.3 Generalized Linear Models

Supervised learning models predict a conditional distribution over a label y given an input $\mathbf{x}$. A *generalized linear model* chooses that distribution from an exponential family Equation 36.51 and lets the input enter through a linear map $\mathbf{W}^\top \mathbf{x}$ (folding any bias term into a constant feature). The cleanest choice sets this linear map directly equal to the natural parameter, $\eta = \mathbf{W}^\top \mathbf{x}$, a pairing known as the *canonical link*. Writing $\mu(\mathbf{x}) = \nabla A(\eta) = \mathbb{E}[T(y) \mid \mathbf{x}]$ for the model's predicted mean of the sufficient statistic, the negative log-likelihood of a single example is $A(\eta) - \eta^\top T(y) - \log h(y)$. By the moment property Equation 36.54, its gradient with respect to η is simply $\mu(\mathbf{x}) - T(y)$. Applying the chain rule through $\eta = \mathbf{W}^\top \mathbf{x}$ immediately gives

$$\nabla_{\mathbf{W}}\big(-\log p(y \mid \mathbf{x})\big) = \mathbf{x}\,\big(\mu(\mathbf{x}) - T(y)\big)^\top. \tag{36.56}$$

Whatever the exponential family, the gradient is always the input multiplied by the prediction error: predicted mean minus observed statistic. With the moment property in hand, the mystery from the opening of this section resolves in two lines. Moreover, the loss is convex in $\mathbf{W}$, since it composes the convex function A with a linear function of $\mathbf{W}$ and adds a linear term. Note that this tidy gradient belongs specifically to the canonical link: the probit model in Section 36.4.4 feeds the same linear score into a Gaussian CDF instead, and its gradient picks up an extra factor that varies with the prediction. Four familiar models arise simply by swapping the exponential family:

label y	family	$T(y)$	mean $\mu(\mathbf{x})$	model
real number	Gaussian, variance 1	y	η	linear regression
yes or no	Bernoulli	$y \in \{0, 1\}$	sigmoid(η)	logistic regression

label y	family	$T(y)$	mean $\mu(\mathbf{x})$	model
one of K classes	categorical	one-hot $\mathbf{y}$	softmax(η)	softmax regression
count	Poisson	y	e^{η}	Poisson regression

To see the rule on a less familiar label, consider a Poisson model for count data. If the model predicts a mean of $e^{\eta} = 3$ for an input and the observed count is $y = 5$, then the gradient of the loss with respect to η is $3 - 5 = -2$. A gradient descent step therefore increases η and pushes the predicted mean upward toward 5.

In the first row, fixing the Gaussian variance at 1 puts the quadratic term $e^{-y^2/2}$ into the base measure $h(y)$, so the negative log-likelihood reduces to the squared loss of Section 2.1 up to an additive constant. A deep neural network keeps this final layer intact (a linear map into natural parameters followed by the family's negative log-likelihood), while replacing the raw input $\mathbf{x}$ with learned representations. Although the overall loss is no longer convex in the network weights, its gradient with respect to the natural parameters at the output layer remains prediction minus observation (Bishop 2006).

36.4.4 Origins of the Softmax

Throughout the main text, classifiers convert a vector of K logits $\mathbf{z}$ into class probabilities using the *softmax*,

$$\text{softmax}(\mathbf{z})_k = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}}, \qquad k = 1, \dots, K. \tag{36.57}$$

We just saw this map emerge from the categorical row of our exponential-family tables. It has obvious desirable properties: each probability increases with its own logit, and all probabilities are positive and sum to one. Yet many other maps share those properties; normalizing $\log(1 + e^{z_k})$ instead of e^{z_k}, for instance, also produces positive probabilities that preserve the ranking of the logits. What sets the softmax apart is that logit differences are log-odds: $z_i - z_j = \log(p_i/p_j)$. Remarkably, three independent arguments from three different fields all converge on this exact formula: a model of noisy human choices, the Boltzmann distribution of statistical physics, and the exponential family. The third is the shortest and is the only one the rest of this section relies on.

36.4.4.1 Noisy Choices and Gumbel Noise

The first argument models how discrete labels are generated in the first place. Imagine a decision maker, such as a shopper choosing a product or an annotator labeling an image, who assigns each option k an underlying score z_k but makes occasional errors. A natural way to model noisy selection is to add independent random noise ϵ_k to every score and pick the option with the highest perturbed score:

$$y = \operatorname*{argmax}_k \, (z_k + \epsilon_k).$$

Under this model, the probability of class k is the chance that option k wins the noisy contest, so its formula is determined by the distribution of the noise. Gaussian noise is the first thing most people try. With two options and $\epsilon_k \sim \mathcal{N}(0, s^2)$, option 1 wins whenever $\epsilon_2 - \epsilon_1 < z_1 - z_2$. Because the difference $\epsilon_2 - \epsilon_1$ is itself Gaussian with variance $2s^2$, the winning probability is

$$P(y = 1) = \Phi\Big(\frac{z_1 - z_2}{\sqrt{2}\, s}\Big),$$

where Φ denotes the standard Gaussian CDF (and ϕ its density). This is the *probit* model: the choice probability depends only on the score gap, measured in units of the noise scale.

Unfortunately, the probit model loses its closed form as soon as there are more than two options. The probability that option k beats all $K-1$ rivals becomes an integral of a product of $K-1$ Gaussian CDFs, which has no elementary formula and would have to be integrated numerically, along with its gradient, for every training example at every step.

Switching from Gaussian to Gumbel noise eliminates that integral completely. The standard *Gumbel* distribution has the CDF $G(x) = \exp(-e^{-x})$. It arises naturally in extreme-value theory as the limiting distribution of the suitably shifted and scaled maximum of many independent draws from an exponentially decaying tail, such as a Gaussian or exponential distribution (Gumbel 1954), and McFadden (1974) introduced it to model economic choice.

Proposition (Gumbel noise gives the softmax). *If $\epsilon_1, \dots, \epsilon_K$ are independent standard Gumbel random variables, then the noisy argmax follows the softmax distribution,*

$$P\Big(\operatorname*{argmax}_j\,(z_j + \epsilon_j) = k\Big) = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}}.$$

Proof. Let $S = \sum_j e^{z_j}$. Each perturbed score $z_j + \epsilon_j$ has CDF $G(t - z_j) = \exp(-e^{z_j} e^{-t})$. Conditioned on option k achieving a score of $t = z_k + \epsilon_k$, independence implies that all other $K - 1$ options fall below t with probability $\prod_{j \neq k} \exp(-e^{z_j} e^{-t}) = \exp(-(S - e^{z_k}) e^{-t})$. Multiplying by the density $e^{z_k} e^{-t} \exp(-e^{z_k} e^{-t})$ of $t = z_k + \epsilon_k$, integrating over t, and substituting $v = e^{-t}$ yields

$$P(y = k) = \int_{-\infty}^{\infty} e^{z_k} e^{-t} \exp(-S e^{-t})\, dt = e^{z_k} \int_0^{\infty} e^{-Sv}\, dv = \frac{e^{z_k}}{S}. \quad \blacksquare$$

The calculation succeeds because a product of Gumbel CDFs is still a Gumbel CDF. Across all K options, the maximum score satisfies $P(\max_j (z_j + \epsilon_j) \le t) = \exp(-S e^{-t}) = G(t - \log S)$, meaning the winning score is simply a standard Gumbel variable shifted by the log-sum-exp $\log \sum_j e^{z_j}$. Viewed this way, log-sum-exp acts as a smooth maximum of the logits, and its gradient, the softmax, gives the probability that each option achieves that maximum. Exercise 1 offers a second proof of the same identity using competing exponential waiting times.

Read in reverse, the proposition doubles as a sampling algorithm: to draw a category from $\mathrm{softmax}(\mathbf{z})$, add independent standard Gumbel noise to each logit and take the argmax. This is the widely used *Gumbel-max trick*.

For the remainder of Section 36.4.4, we follow the main text and use T to denote a temperature rather than a sufficient statistic. Because $\operatorname{argmax}_k(z_k + T\epsilon_k) = \operatorname{argmax}_k(z_k/T + \epsilon_k)$ for any $T > 0$, scaling the Gumbel noise by T samples from $\operatorname{softmax}(\mathbf{z}/T)$: noisier decisions correspond to higher temperatures. Let us test the Gumbel-max trick directly by drawing 10^5 choices over four logits at $T = 1$ and $T = 2$ and comparing the empirical win rates against the exact softmax probabilities.

```
z = np.array([2.0, 1.0, 0.1, -1.0])                    # four logits
softmax = lambda v: np.exp(v) / np.exp(v).sum()
rng = np.random.default_rng()
n = 100_000
for T in (1.0, 2.0):                                    # noise scale = temperature
    winners = np.argmax(z + T * rng.gumbel(size=(n, 4)), axis=1)
    freq = np.bincount(winners, minlength=4) / n
    print(f'T = {T:.0f}: argmax frequencies {freq.round(3)}')
    print(f'      softmax(z / T)        {softmax(z / T).round(3)}'
          f'  max diff {np.abs(freq - softmax(z / T)).max():.3f}')
```

```
T = 1: argmax frequencies [0.638 0.234 0.095 0.033]
      softmax(z / T)        [0.638 0.235 0.095 0.032]  max diff 0.001
T = 2: argmax frequencies [0.451 0.274 0.173 0.102]
      softmax(z / T)        [0.451 0.274 0.174 0.101]  max diff 0.002
```

The empirical frequencies agree with $\operatorname{softmax}(\mathbf{z}/T)$ to within a few thousandths, right at the Monte Carlo noise floor for 10^5 draws. Doubling the noise scale flattens the win rates in the exact same way as halving the logits.

With two options, the difference of two independent standard Gumbel random variables follows the logistic distribution, whose CDF is $\operatorname{sigmoid}(x) = 1/(1 + e^{-x})$, so this *logit* model yields $P(y = 1) = \operatorname{sigmoid}(z_1 - z_2)$. Because the sigmoid closely tracks a rescaled Gaussian CDF ($|\operatorname{sigmoid}(1.7x) - \Phi(x)| < 0.01$ for all x), fitted probit and logit models produce nearly identical probabilities on ordinary examples. Where they diverge is in the tails: on a confidently mislabeled outlier, the logit loss grows only linearly in the score difference, whereas the probit loss grows quadratically, allowing outliers to pull a probit fit much harder (Exercise 4).

36.4.4.2 Temperature and the Boltzmann Distribution

The second argument comes from statistical physics, where the exact same exponential law governs energy states. A physical system in thermal equilibrium at temperature T, such as a molecule in a gas, occupies a state k of energy E_k with probability

$$p_k = \frac{\exp(-E_k/(k_{\mathrm{B}}T))}{\sum_j \exp(-E_j/(k_{\mathrm{B}}T))},$$

where k_{B} is Boltzmann's constant. Lower-energy states are more probable, and the temperature T controls how sharply probability concentrates on them. Boltzmann discovered this distribution for gas molecules, and Gibbs (1902) built statistical mechanics upon it. If we measure temperature in energy units (setting $k_{\mathrm{B}} = 1$) and identify energies with negative logits ($E_k = -z_k$), the Boltzmann law becomes $\operatorname{softmax}(\mathbf{z}/T)$. In this language, a logit is a negative energy, and the most likely class is simply the lowest-energy state.

Our maximum-entropy proposition derives the same formula without invoking physics at all. A physicist fixes the mean energy and assumes nothing else; in classification, the analogous constraint fixes the average logit $\sum_k p_k z_k$. Take the K classes as outcomes with $h \equiv 1$ and treat the logit z_k as the statistic. Among all distributions on the K classes achieving a prescribed average logit (strictly between the smallest and largest entries of $\mathbf{z}$), the maximum-entropy distribution is $p_k \propto e^{\eta z_k}$. Whenever the target average exceeds the unweighted mean of the logits, $\eta > 0$, and setting $T = 1/\eta$ recovers $\mathrm{softmax}(\mathbf{z}/T)$. As T cools from ∞ down to 0, the average logit rises smoothly from the mean of the logits to their maximum. For the logits $\mathbf{z} = (2, 1, 0.1, -1)$ in our simulation, the standard softmax ($T = 1$) achieves an average logit of 1.49, sitting between the unweighted mean 0.525 and the maximum 2.

36.4.4.3 The Categorical as an Exponential Family

The third argument is the one we already uncovered earlier in this section. Encoding a class label as a one-hot vector $\mathbf{y}$, the categorical distribution is $P(\mathbf{y}) = \exp(\mathbf{z}^\top \mathbf{y} - A(\mathbf{z}))$ with log-partition function $A(\mathbf{z}) = \log \sum_k e^{z_k}$, an exponential family whose natural parameters are the logits. By the moment property, its mean is $\nabla A(\mathbf{z}) = \mathrm{softmax}(\mathbf{z}) = \mathbb{E}[\mathbf{y}]$. Its negative log-likelihood is automatically convex in $\mathbf{z}$, and by Equation 36.56, its gradient is simply $\mathrm{softmax}(\mathbf{z}) - \mathbf{y}$.

All three arguments arrive at the same function, yet each illuminates a different facet of it: noisy choices give us an exact sampler, the Boltzmann distribution explains temperature scaling, and the exponential family hands us a convex loss and its effortless gradient.

36.4.5 Conjugate Priors

Maximum likelihood compresses 9 heads in 13 flips into a single number, $9/13$. A point estimate alone cannot tell us how confident to be in it: 9 heads in 13 flips and 900 heads in 1300 flips produce the exact same ratio. Representing our knowledge as a probability distribution over the coin's bias captures that uncertainty directly, and ideally it should absorb each new flip through a simple update rather than starting from scratch.

From here on, we treat the bias itself as an unknown random quantity and denote it by θ, matching the notation of the upcoming sections. A distribution over θ before seeing the data is a *prior*. Bayes' rule multiplies the prior by the likelihood and normalizes the result, $p(\theta \mid x) \propto p(\theta)\, P(x \mid \theta)$, producing the *posterior*. A prior is *conjugate* to a likelihood when the posterior belongs to the very same family of distributions as the prior, so that observing data merely updates its parameters.

36.4.5.1 The Beta Prior on a Coin

A prior on a coin's bias must be a density supported on $[0, 1]$. Viewed as a function of θ, the likelihood of x heads in n flips is $\theta^x (1-\theta)^{n-x}$, so a prior with the same algebraic form combines with it simply by adding exponents. Its normalizing constant uses the Gamma function $\Gamma(z) = \int_0^\infty t^{z-1} e^{-t}\, dt$ from Section 36.3, which generalizes the factorial via $\Gamma(z+1) = z\,\Gamma(z)$ and $\Gamma(n+1) = n!$. The *Beta* distribution with parameters $a, b > 0$ has the density

$$\text{Beta}(\theta \mid a, b) = \frac{\theta^{a-1}(1-\theta)^{b-1}}{B(a,b)}, \qquad 0 \le \theta \le 1, \tag{36.58}$$

where the normalizer $B(a,b) = \int_0^1 \theta^{a-1}(1-\theta)^{b-1} d\theta$ equals $\Gamma(a)\Gamma(b)/\Gamma(a+b)$. The mean follows effortlessly from this normalizer, since multiplying the density by θ simply increments a by one:

$$\mathbb{E}[\theta] = \frac{1}{B(a,b)} \int_0^1 \theta^{a}(1-\theta)^{b-1}\, d\theta = \frac{B(a+1,b)}{B(a,b)} = \frac{a}{a+b},$$

where we used $\Gamma(a+1) = a\,\Gamma(a)$ and $\Gamma(a+b+1) = (a+b)\Gamma(a+b)$ to cancel the Gamma factors. Applying the same trick twice yields the variance; writing $m = a/(a+b)$ for the mean,

$$\text{Var}(\theta) = \frac{ab}{(a+b)^2(a+b+1)} = \frac{m(1-m)}{a+b+1}.$$

Larger values of $a+b$ concentrate the density more tightly around its mean, while $a = b = 1$ gives the flat uniform density on $[0,1]$.

Now suppose we observe x heads in n flips. Multiplying the prior by the likelihood adds the exponents directly,

$$\underbrace{\theta^{a-1}(1-\theta)^{b-1}}_{\text{prior}} \cdot \underbrace{\theta^{x}(1-\theta)^{n-x}}_{\text{likelihood}} = \theta^{(a+x)-1}(1-\theta)^{(b+n-x)-1},$$

which, once normalized, is again a Beta density with parameters shifted by the observed counts:

$$\theta \mid x \;\sim\; \text{Beta}\big(a + x,\; b + n - x\big). \tag{36.59}$$

This update has a clean *pseudo-count* interpretation: a and b act as prior counts of heads and tails, and seeing x heads and $n-x$ tails simply adds them to the tally. The posterior mean,

$$\mathbb{E}[\theta \mid x] = \frac{a+x}{a+b+n},$$

is the fraction of heads after folding in those pseudo-counts. It sits between the prior mean $a/(a+b)$ and the maximum-likelihood estimate x/n, converging to x/n as n grows large.

Be careful about one subtle distinction: how many flips the prior is worth depends on which summary you compute. When both posterior parameters exceed 1, the posterior mode is $(a+x-1)/(a+b+n-2)$, so the prior contributes $a-1$ heads and $b-1$ tails to the mode. To keep the convention clear once and for all: *a and b are the pseudo-counts of the posterior mean, whereas $a-1$ and $b-1$ are those of the posterior mode.* Under the uniform prior $\text{Beta}(1,1)$, the mean includes one imagined head and one imagined tail, while the mode includes none.

Returning to our chapter's coin with the uniform prior, 9 heads in 13 flips update $\text{Beta}(1, 1)$ to $\text{Beta}(10, 5)$. Its posterior mean is $10/15 = 2/3$, a classic formula known as *Laplace's rule of succession*, $(x + 1)/(n + 2)$ (Laplace 1814). Its mode $9/13$ coincides with the maximum-likelihood estimate, and its standard deviation is $\sqrt{(2/3)(1/3)/16} \approx 0.118$.

If we instead see 90 heads in 130 flips, the posterior sharpens to $\text{Beta}(91, 41)$, with mean 0.689 and standard deviation 0.040. Collecting ten times as many flips shrinks the standard deviation by a factor of $2.9 \approx \sqrt{133/16}$: posterior uncertainty narrows at the $1/\sqrt{n}$ rate, the same scaling we will see for standard errors in Section 36.5.

What probability should we assign to the next flip landing heads? Given θ, that probability is θ; averaging over our posterior beliefs yields $P(\text{heads} \mid x) = \int_0^1 \theta\, p(\theta \mid x)\, d\theta = \mathbb{E}[\theta \mid x]$. For our coin, that predictive probability is $2/3$: Laplace's rule directly gives the probability of the next head.

Conjugacy saved us from computing the normalizing integral in Bayes' rule. What if our prior is not conjugate? In one dimension, we can simply evaluate the product of prior and likelihood on a fine grid of θ values and normalize the sum. Consider a Gaussian prior on the natural parameter, $\eta \sim \mathcal{N}(0, 1)$, the same kind of prior that Section 36.8 places on logistic-regression weights. It expresses a belief that the coin is probably near fair, and it is not conjugate to the Bernoulli likelihood: by a change of variables, its density in θ is $\phi(\eta)/(\theta(1-\theta))$, where ϕ is the standard Gaussian density, which is not a Beta density. Let us compute the posterior under both priors on the same grid and compare their summaries.

```
x, n = 9, 13                                    # 9 heads in 13 flips
theta = np.linspace(0, 1, 100_001)[1:-1]        # interior grid points
eta = np.log(theta) - np.log1p(-theta)          # natural parameter
priors = {'Beta(1, 1)': np.ones_like(theta),    # densities in theta
          'eta ~ N(0, 1)': np.exp(-eta**2 / 2) / (theta * (1 - theta))}
for name, prior in priors.items():
    post = prior * theta**x * (1 - theta)**(n - x)
    post /= post.sum()                          # grid weights sum to one
    mean = post @ theta
    sd = math.sqrt(post @ (theta - mean)**2)
    print(f'{name:13s}: mean {mean:.3f}, sd {sd:.3f}, '
          f'mode {theta[post.argmax()]:.3f}')
```

```
Beta(1, 1)   : mean 0.667, sd 0.118, mode 0.692
eta ~ N(0, 1): mean 0.644, sd 0.111, mode 0.665
```

Under the uniform prior, the grid recovers the exact moments of $\text{Beta}(10, 5)$: mean 0.667, standard deviation 0.118, and mode $9/13 \approx 0.692$. Placing a Gaussian prior on the log-odds pulls the posterior closer to a fair coin, lowering the mean to 0.644, the mode to 0.665, and the standard deviation slightly to 0.111. A grid handles the non-conjugate prior just as easily as the conjugate one, but only because θ is scalar; Section 36.8 develops methods that scale to high-dimensional parameter vectors. Figure 36.21 visualizes how the Beta posterior sharpens as flips accumulate.

36.4.5.2 Conjugacy for Every Exponential Family

At first glance, the Beta update looks like a lucky algebraic coincidence. In fact, it is a universal property of exponential families: every exponential family admits a conjugate prior,

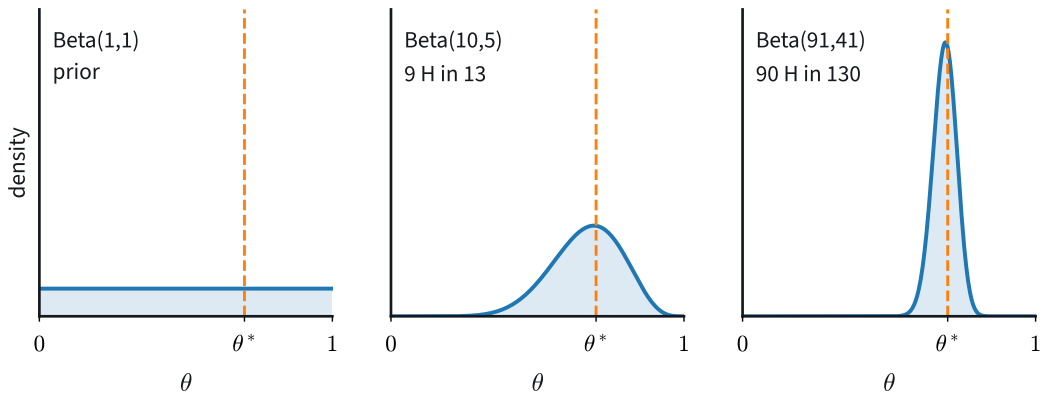


Figure 36.21: Bayesian updating of a coin's bias θ. Left: the flat Beta$(1,1)$ prior. Middle: the Beta$(10,5)$ posterior after 9 heads in 13 flips of a coin with true bias $\theta^* = 0.7$ (dashed), with mean $2/3$ and standard deviation 0.12. Right: the Beta$(91,41)$ posterior after 130 flips, with standard deviation 0.04. Each update adds observed heads and tails to the pseudo-counts, and the posterior width shrinks like $1/\sqrt{n}$.

and updating it always amounts to adding the sufficient statistics.

Proposition (conjugate prior). *Consider a likelihood Equation 36.51 and the prior on its natural parameters*

$$p(\eta) \propto \exp(\nu^\top \eta - \kappa\, A(\eta)), \tag{36.60}$$

where $\nu \in \mathbb{R}^d$ and $\kappa \in \mathbb{R}$ are chosen so that the density is normalizable. After n independent observations $\mathbf{x}_1, \dots, \mathbf{x}_n$, the posterior retains the exact same form, with ν updated to $\nu + \sum_i T(\mathbf{x}_i)$ and κ updated to $\kappa + n$.

Proof. Dropping factors that do not depend on η, the likelihood contributes $\exp(\eta^\top \sum_i T(\mathbf{x}_i) - nA(\eta))$. Multiplying it by the prior combines the exponents in a single line:

$$e^{\nu^\top \eta - \kappa A(\eta)} \cdot e^{\eta^\top \sum_i T(\mathbf{x}_i) - nA(\eta)} = e^{(\nu + \sum_i T(\mathbf{x}_i))^\top \eta - (\kappa + n) A(\eta)}.$$

Normalizing this product gives the posterior, which therefore belongs to the same family with the updated parameters $\nu + \sum_i T(\mathbf{x}_i)$ and $\kappa + n$. ■

The prior's parameters ν and κ, called *hyperparameters*, behave like *pseudo-data*: ν acts as a prior sum of sufficient statistics and κ as a prior sample size (Bishop 2006). For the Bernoulli coin, this prior is written in terms of the logit $\eta = \log \frac{\theta}{1-\theta}$; changing variables from η to θ transforms it into Beta$(\nu, \kappa - \nu)$, and the general update $\nu \to \nu + x$, $\kappa \to \kappa + n$ reproduces the Beta update $a \to a + x$, $b \to b + n - x$. Exercise 7 works through this change of variables, including its Jacobian.

36.4.5.3 Gamma–Poisson and Dirichlet–Multinomial

Two more widely used conjugate pairs follow the same pattern, one for event rates and one for categorical probabilities over words or classes. Viewed as a function of the rate λ, the likelihood of n Poisson counts $x_1, \dots, x_n$ is proportional to $\lambda^{\sum_i x_i} e^{-n\lambda}$. The matching conjugate prior is the *Gamma* distribution with shape $a > 0$ and rate $b > 0$, whose density is $\mathrm{Gamma}(\lambda \mid a, b) = \frac{b^a}{\Gamma(a)} \lambda^{a-1} e^{-b\lambda}$ for $\lambda > 0$ (with mean a/b). Multiplying prior and likelihood yields the posterior $\mathrm{Gamma}\big(a + \sum_i x_i,\ b + n\big)$, so the prior encodes a imagined events spread across b imagined time periods. Exercise 7 derives this pair directly from the general proposition.

To generalize the Beta prior from two outcomes to K categories, we use the *Dirichlet* distribution, a density over probability vectors $\mathbf{p}$ with $p_k \geq 0$ and $\sum_k p_k = 1$. With concentration parameters $a_k > 0$, its density is

$$\mathrm{Dir}(\mathbf{p} \mid \mathbf{a}) = \frac{\Gamma(\sum_k a_k)}{\prod_k \Gamma(a_k)} \prod_{k=1}^{K} p_k^{a_k - 1},$$

which has mean $\mathbb{E}[p_k] = a_k / \sum_j a_j$ and reduces to the Beta distribution when $K = 2$. Observing category counts $x_1, \dots, x_K$ gives a likelihood proportional to $\prod_k p_k^{x_k}$, so the posterior is $\mathrm{Dir}(\mathbf{a} + \mathbf{x})$: each category's observed count is added to its pseudo-count.

To see why this matters in practice, suppose a three-word vocabulary produces the counts $(3, 0, 1)$. The maximum-likelihood estimate $(0.75, 0, 0.25)$ assigns probability zero to the unseen second word, which would make any future document containing that word seem impossible. Placing a uniform prior $\mathrm{Dir}(1, 1, 1)$ on $\mathbf{p}$ yields the posterior $\mathrm{Dir}(4, 1, 2)$, whose mean is $(4/7, 1/7, 2/7) \approx (0.571, 0.143, 0.286)$. Adding one to every count before normalizing, known as *add-one smoothing*, is simply the posterior mean under this Dirichlet prior; the Naive Bayes classifier in Section 36.11 uses the two-class case.

Closed-form posteriors are the exception rather than the rule: outside conjugate pairs, Bayesian inference requires numerical approximation (Section 36.8). Before tackling that, however, we need an answer to a more basic question about point estimates. From 13 flips we obtained $9/13$, a single number without an error bar: how far might $9/13$ be from the true bias, and does a coin with 11 heads in 13 flips really differ from one with 9? Section 36.5 takes up these questions.

36.4.6 Summary

Nearly every distribution in our catalogue shares the exponential-family form $h(\mathbf{x}) \exp(\eta^\top T(\mathbf{x}) - A(\eta))$, where data and parameters interact solely through the inner product $\eta^\top T(\mathbf{x})$, and maximizing entropy under fixed average statistics produces this exact form. Differentiating the log-partition function A generates moments: ∇A is the mean of T and $\nabla^2 A$ is its covariance, which immediately proves that A is convex. Consequently, maximum likelihood is a convex optimization problem that matches the model's expected sufficient statistics to their sample averages, admitting a finite solution whenever that sample average is attainable. Pairing an exponential family with its canonical link yields a generalized linear model whose gradient is always input times prediction minus observation, unifying linear, logistic, softmax, and Poisson regression while explaining why non-canonical links such as the probit behave differently. The softmax emerges as

the mean map of the categorical family (as well as from Gumbel noise and the Boltzmann distribution). Finally, every exponential family admits a conjugate prior updated simply by adding sufficient statistics; for a coin, Beta(a, b) contributes a and b pseudo-counts to the posterior mean and $a - 1$ and $b - 1$ to the mode.

36.4.7 Exercises

1. **Exponential races.** Let $X_1, \dots, X_m$ be independent with $X_i \sim \text{Exp}(\lambda_i)$, for example the waiting times for the next event in m independent event streams.
   1. Show that $\min_i X_i \sim \text{Exp}(\sum_i \lambda_i)$.
   2. Show that the j-th stream fires first with probability $P(X_j = \min_i X_i) = \lambda_j / \sum_i \lambda_i$.
   3. Let $V_1, \dots, V_K$ be independent Exp(1) variables and let $\mathbf{z} \in \mathbb{R}^K$ be a vector of logits. Use (b) to show that $\text{argmax}_k (z_k - \log V_k)$ is distributed according to the softmax probabilities Equation 36.57. Because $-\log V_k$ has the standard Gumbel distribution, this is a second proof that Gumbel noise gives the softmax.
2. **Maximum entropy on the integers.** Consider distributions $(p_0, p_1, p_2, \dots)$ on $\{0, 1, 2, \dots\}$ with a fixed mean $m > 0$.
   1. Maximize the Shannon entropy $H[p] = -\sum_k p_k \log p_k$ subject to $\sum_k p_k = 1$ and $\sum_k k\, p_k = m$, using Lagrange multipliers (Section 35.6.1). Show that the stationary point is a geometric distribution $p_k = (1 - r)\, r^k$, and express r through m.
   2. Repeat the derivation for the entropy relative to $h(k) = 1/k!$, $H_h[p] = -\sum_k p_k \log(p_k / h(k))$, and show that the stationary point is Poisson(m).
   3. At $m = 4$, compute the Shannon entropy of both distributions numerically. Which one is larger, and why must it be? Compare $P(X \geq 15)$ under the two distributions. Which assumption about large counts does the choice of base measure encode?
   4. A stationary point of a Lagrangian need not be a maximum. Use the maximum-entropy proposition to prove that the two distributions of
      (a) and (b) are the global maximizers, by exhibiting each as a member of an exponential family whose mean equals m.
3. **Moments from the log-partition.** The moment property Equation 36.54 and its second-order form $\nabla^2 A = \text{Cov}(T)$ turn moment calculations into differentiation.
   1. For the Poisson, $A(\eta) = e^\eta$. Derive the mean and the variance from A. Which property of the function A makes them equal?
   2. For the Gaussian, the natural parameters are $\eta_1 = \mu/\sigma^2$ and $\eta_2 = -1/(2\sigma^2)$. Show that $A = -\eta_1^2/(4\eta_2) - \frac{1}{2}\log(-2\eta_2)$, and differentiate it to recover $\mathbb{E}[x]$, $\mathbb{E}[x^2]$, Var(x) and Cov(x, x^2).
   3. The Laplace distribution with a fixed location μ and scale b has the sufficient statistic $T(x) = |x - \mu|$. Write it in the form Equation 36.51, and use A to compute $\mathbb{E}\,|x - \mu|$ and Var$|x - \mu|$. Check the result against the Laplace variance $2b^2$.
   4. For n Gaussian observations, write the two moment-matching equations $\nabla A(\hat{\eta}) = \bar{T}$ and solve them for $\hat{\mu}$ and $\hat{\sigma}^2$. What happens for $n = 1$? Relate your answer to the condition that $\bar{T}$ be an attainable mean.
4. code **Probit and logit.** For two classes with the score difference $m = z_1 - z_2$, the logit model gives $P(y = 1) = \text{sigmoid}(m)$ and the probit model with noise scale s gives

$P(y=1) = \Phi(m/(\sqrt{2}\,s))$.

1. Show that the difference of two independent standard Gumbel variables has the CDF $\mathrm{sigmoid}(x)$. The standard Gumbel distribution has variance $\pi^2/6$. Find the scale c for which the distribution with CDF $\mathrm{sigmoid}(cx)$ has variance 1, like the standard Gaussian.
2. Find numerically the scale c that minimizes $\max_x |\,\mathrm{sigmoid}(cx) - \Phi(x)|$, and the value of this maximum. Why is this scale smaller than the one of (a)?
3. An example of class 2 lies on the wrong side of the decision boundary, with $m > 0$. Show that its negative log-likelihood $-\log \mathrm{sigmoid}(-m)$ under the logit model lies between m and $m + e^{-m}$, and that its derivative with respect to m is less than 1. Use the bounds $\frac{x}{1+x^2}\,\phi(x) \le 1 - \Phi(x) \le \frac{\phi(x)}{x}$ for $x > 0$ to show that its negative log-likelihood $-\log \Phi(-m/(\sqrt{2}\,s))$ under the probit model grows like $m^2/(4s^2)$, and that its derivative grows linearly in m.
4. Draw 100 inputs $x \sim \mathcal{N}(1,1)$ of class 1 and 100 inputs $x \sim \mathcal{N}(-1,1)$ of class 2, and let $m = wx + b$. Fit (w, b) by maximum likelihood under both models. Then add three examples at $x = 6$ labeled as class 2, and fit again. For each model, compute the ratio of the slope w with these mislabeled examples to the slope without them, and explain the difference between the models with (c). Compute $\log \Phi$ with a numerically stable routine such as `scipy.special.log_ndtr`.

5. **Minimal categorical family.** ● Code a categorical outcome as a one-hot vector $\mathbf{x} \in \{0,1\}^K$ with class probabilities $\mathbf{p} = \mathrm{softmax}(\mathbf{z})$, written as an exponential family with $T(\mathbf{x}) = \mathbf{x}$, natural parameters $\mathbf{z}$ and $A(\mathbf{z}) = \log \sum_k e^{z_k}$.
   1. Compute $\nabla^2 A(\mathbf{z})$ and show that it equals the covariance $\mathrm{diag}(\mathbf{p}) - \mathbf{p}\mathbf{p}^\top$ of the one-hot vector $\mathbf{x}$. Every one-hot vector satisfies $\mathbf{x}\mathbf{x}^\top = \mathrm{diag}(\mathbf{x})$. Show that $\nabla^2 A(\mathbf{z})\,\mathbf{1} = \mathbf{0}$, and that the gradient of the negative log-likelihood of a dataset is orthogonal to $\mathbf{1}$ at every $\mathbf{z}$. Gradient descent with a small step size runs on the logits from a starting point $\mathbf{z}_0$, and every class occurs in the data. Which function of $\mathbf{z}$ does gradient descent never change? Which of the minimizers does it therefore approach?
   2. Fix $z_K = 0$. This gives a minimal representation with $\eta = (z_1, \dots, z_{K-1})$ and $T(\mathbf{x}) = (x_1, \dots, x_{K-1})$. Show that $\nabla^2 A(\eta)$ is positive definite whenever every $p_k > 0$. Given class counts $n_1, \dots, n_K$, solve the moment-matching condition $\nabla A(\hat\eta) = \bar T$ for the maximum-likelihood $\hat\eta$. What happens when some class has $n_k = 0$?

6. code **Poisson regression.** ● A generalized linear model for counts makes the natural parameter of a Poisson an affine function of the features: $y_i \mid \mathbf{x}_i \sim \mathrm{Poisson}(e^{\eta_i})$ with $\eta_i = \mathbf{w}^\top \mathbf{x}_i + b$.
   1. Use the negative log-likelihood Equation 36.55 to write the loss as a function of $(\mathbf{w}, b)$, and compute its gradient and Hessian. Show that the loss is convex. Which equations does a zero gradient impose on the fitted means $\hat m_i = e^{\hat\eta_i}$?
   2. Generate $n = 1000$ examples with $\mathbf{x}_i \sim \mathcal{N}(\mathbf{0}, \mathbf{I}_2)$, $\mathbf{w} = (-0.3, 1.0)$ and $b = 0.5$. Starting from zero, fit $\mathbf{u} = (\mathbf{w}, b)$ with Newton's method, which repeats the step $\mathbf{u} \leftarrow \mathbf{u} - \left(\nabla^2 L(\mathbf{u})\right)^{-1} \nabla L(\mathbf{u})$ for the loss L, and check the equations of (a) at the solution.
   3. Multiply each rate by an independent $g_i \sim \mathrm{Gamma}(1,1)$, which has mean 1, so that the counts become overdispersed while their conditional means stay e^{η_i}. Fit the Poisson model to 200 such datasets and to 200 datasets without the multiplier. Compare the mean and the standard deviation of the estimates across datasets.

Explain the behavior of the mean with the equations of (a).

7. **Conjugate priors from natural parameters.** ● For a scalar natural parameter η, the conjugate prior Equation 36.60 reads $p(\eta) \propto \exp\big(\nu\,\eta - \kappa\,A(\eta)\big)$, and n observations update it by $\nu \to \nu + \sum_i T(x_i)$ and $\kappa \to \kappa + n$.
   1. For the Poisson, $\eta = \log\lambda$ and $A(\eta) = e^{\eta}$. Change variables from η to λ, including the Jacobian, and show that the prior on λ is a Gamma distribution. Identify its shape and rate, and check that the update reproduces the Gamma posterior of Section 36.4.5.3.
   2. For the Bernoulli, $\eta = \log\frac{\theta}{1-\theta}$ and $A(\eta) = \log(1+e^{\eta})$. Show that the prior on θ is Beta$(\nu, \kappa-\nu)$, and find the values of (ν,κ) for which it is normalizable.
   3. Take the flat prior on the logit, $\nu = \kappa = 0$. Which density on θ does it induce, and is that density normalizable? After x heads in n flips, when is the posterior normalizable, and what is its mean? Compare it with the mean of the posterior Equation 36.59 under the uniform prior Beta$(1,1)$ on θ. What does the comparison say about the phrase "a flat prior"?

36.5 Statistics

Every model in this book is eventually judged by a number computed on held-out data: a test accuracy, a validation loss, or a benchmark score. We tend to treat that number as an intrinsic property of the model, yet it also reflects the particular examples that happened to land in the test set. Suppose a classifier answers 912 of 1,000 test questions correctly, giving an accuracy of 91.2%. On a different set of 1,000 questions from the same source, it would score slightly higher or slightly lower, so 91.2% is really one realization of a random variable. Now suppose a second model answers 921 of those same questions correctly. Is it actually better, or did this particular batch of questions happen to favor it?

Statistics answers questions like this by asking how an estimate would fluctuate across all the alternative samples we never drew. Those fluctuations follow a distribution, the *sampling distribution* of the estimate, and every tool in this section is built to describe it or reason with it. Its center and spread tell us whether an estimate is systematically off target and how widely it scatters. A hypothesis test checks whether an observed difference fits comfortably inside the scatter that chance alone would produce. A confidence interval collects all parameter values that the data cannot rule out. And when no closed-form formula describes the sampling distribution, the bootstrap simulates it directly from the data in hand.

In deep learning, these tools matter because almost every engineering decision rests on comparing noisy numbers: which model to deploy, which hyperparameter wins, or whether a new idea helped. Acting on a gap smaller than its own noise is no better than flipping a coin. Whereas the earlier sections of this chapter described how data behave when the underlying distribution is known, this section works in the opposite direction, turning observed data into estimates with honest error bars. The data need not even come from the outside world. When we approximate an expectation by averaging samples generated on a computer (a Monte Carlo estimate), the result is just another sample mean, so the same theory gives its error bar (Section 36.5.1.7). Likewise, test accuracy behaves like a coin tossed once per question, so the coin from earlier sections (9 heads in 13 flips) will reappear along the way as a sanity check.

If you remember only three ideas from this section, make them these: mean squared error splits cleanly into squared bias plus variance; a p-value is uniformly distributed when the

null hypothesis is true; and a confidence interval is simply the set of parameter values that a hypothesis test fails to reject. On a first reading, you can safely skim the discussions of consistency and the delta method. Keep one caveat in mind throughout: every guarantee below assumes that the estimator or test was chosen *before* looking at the data. At the end of the section, we will see how easily those guarantees break down when they are not.

```
from scipy.stats import norm
```

```
rng = np.random.default_rng()
```

36.5.1 Estimators and Their Sampling Distribution

Before we can decide how much to trust that 91.2% accuracy, we need a precise name for the rule that produced it and a simple probability model for how the test questions arrived.

36.5.1.1 Estimators and the Sample Proportion

An *estimator* is any rule that turns a sample $x_1, \dots, x_n$ into a number $\hat{\theta}_n$ intended to approximate an unknown parameter θ. The sample mean, the sample median, and the fraction of correct answers on a test set are all estimators. Because the data are random, $\hat{\theta}_n$ is random too, and its distribution across repeated samples of size n is its *sampling distribution*.

To model our classifier's test accuracy, let $x_i = 1$ if the model answers question i correctly and $x_i = 0$ otherwise. Assuming the test questions are drawn independently from a large pool on which the classifier's true accuracy is θ, the indicators x_i are i.i.d. Bernoulli random variables with mean θ. Our estimate is the sample proportion $\hat{\theta}$ of correct answers among the n questions. By linearity of expectation, $\mathbb{E}[\hat{\theta}] = \theta$, and as in Section 1.6, its variance is $\mathrm{Var}(\hat{\theta}) = \theta(1-\theta)/n$. In other words, the sample proportion is centered right on the true accuracy, and its typical error shrinks at the rate $1/\sqrt{n}$. Plugging in $\theta = 0.912$ and $n = 1{,}000$ gives a standard deviation of $\sqrt{0.912 \cdot 0.088 / 1000} \approx 0.0090$, or about 0.9 percentage points. By the central limit theorem, the sampling distribution is closely approximated by a Gaussian, so roughly 95% of test sets yield an accuracy within ± 1.8 percentage points of θ.

Rather than relying on formulas alone, we can observe this sampling distribution directly. Let us simulate 10,000 independent test sets of 1,000 questions for a classifier whose true accuracy is 0.912, and check the mean and spread of the resulting accuracies.

```
theta, n = 0.912, 1000
acc = rng.binomial(n, theta, 10000) / n      # accuracy on 10,000 test sets
print(f'mean {acc.mean():.4f}, std {acc.std():.4f}, '
      f'formula {np.sqrt(theta * (1 - theta) / n):.4f}')
```

```
mean 0.9119, std 0.0089, formula 0.0090
```

Just as the formula predicted, the simulated accuracies cluster around 0.912 with a standard deviation of 0.0090. That cloud of 10,000 numbers, representing accuracies on hypothetical test sets we never drew, is the object that every tool in this section reasons about.

Now return to our second model, which answered 921 questions correctly. If its score had been measured on a separate, independent test set of 1,000 questions, the observed gap of 0.009 between the two accuracies would have a standard deviation of $\sqrt{0.0090^2 + 0.0085^2} \approx 0.012$, where 0.0085 comes from evaluating the same formula at 0.921. A gap of 0.009 is only about 0.7 of its own standard error, nowhere near enough to separate the two models. Here, however, both models were evaluated on the *same* 1,000 questions. In Section 36.5.2.6, we will see how much sharper the comparison becomes once we take advantage of that shared test set.

36.5.1.2 Bias and Variance

An estimator can miss the truth in two distinct ways: it can aim off center on average, or it can scatter widely around its own average. The *bias* of $\hat{\theta}_n$ measures the first kind of error, the systematic offset between the estimator's average value and the true parameter,

$$\mathrm{Bias}(\hat{\theta}_n) = \mathbb{E}[\hat{\theta}_n] - \theta, \tag{36.61}$$

where the expectation is taken over the random draw of the sample. When the bias is zero for every possible value of θ, we call the estimator *unbiased*. The *variance* measures the second kind of error, how widely $\hat{\theta}_n$ fluctuates around its own center, and its square root is the *standard error*,

$$\mathrm{Var}(\hat{\theta}_n) = \mathbb{E}\big[(\hat{\theta}_n - \mathbb{E}[\hat{\theta}_n])^2\big], \qquad \mathrm{se}(\hat{\theta}_n) = \sqrt{\mathrm{Var}(\hat{\theta}_n)}. \tag{36.62}$$

Put simply, bias tells us where the estimates land on average, while standard error tells us how far a typical draw lands from that average. In our accuracy example, the sample proportion has bias 0 and standard error 0.009. Notice that variance is measured around the estimator's own mean, not around θ: a stubborn rule that ignores the data completely and always outputs 0.5 has zero variance, so all of its error comes from bias.

36.5.1.3 The Sample Variance

The sample proportion is unbiased, which might make unbiasedness seem automatic. Yet a biased estimator shows up immediately as soon as we try to compute a standard error from data. The standard error of a sample mean depends on the population variance σ^2, which is almost never known and must be estimated from the same sample. The most natural choice is the *plug-in variance*, the average squared deviation from the sample mean $\bar{x}$,

$$s_0^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2.$$

Surprisingly, s_0^2 is systematically too small. Why? Because $\bar{x}$ is the unique point that minimizes the sum of squared deviations for *this* particular sample. Measuring deviations from the true mean μ, a different point, always yields a larger sum, so replacing μ by $\bar{x}$ shrinks the numerator.

The smallest nontrivial sample, $n = 2$, makes the size of this shrinkage obvious. Here the sample mean is simply the midpoint of x_1 and x_2, so both deviations equal $\pm(x_1 - x_2)/2$, giving

$$\sum_{i=1}^{2}(x_i - \bar{x})^2 = \frac{(x_1 - x_2)^2}{2}.$$

Because x_1 and x_2 are independent with mean μ and variance σ^2, their difference $x_1 - x_2$ has mean 0 and variance $2\sigma^2$. The right-hand side therefore has expectation σ^2. If we divide by $n = 2$, we get $\sigma^2/2$ on average, only half the true variance. Dividing by $n-1 = 1$ instead recovers σ^2 on average. Remarkably, the exact same $n - 1$ correction removes the bias for every sample size n.

Proposition (unbiased sample variance). *Let $x_1, \dots, x_n$ be i.i.d. with mean μ and finite variance σ^2, and let $n \geq 2$. Then*

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2 \qquad \text{satisfies} \qquad \mathbb{E}[s^2] = \sigma^2. \tag{36.63}$$

Proof. To compare the deviations around $\bar{x}$ with those around μ, write $x_i - \bar{x} = (x_i - \mu) - (\bar{x} - \mu)$ and expand the square. Using $\sum_i (x_i - \mu) = n(\bar{x} - \mu)$ to simplify the cross term gives

$$\begin{aligned}\sum_{i=1}^{n}(x_i - \bar{x})^2 &= \sum_{i=1}^{n}(x_i - \mu)^2 - 2(\bar{x} - \mu)\cdot n(\bar{x} - \mu) + n(\bar{x} - \mu)^2 \\ &= \sum_{i=1}^{n}(x_i - \mu)^2 - n(\bar{x} - \mu)^2.\end{aligned}$$

The cross term is twice the final term with the opposite sign, so centering at $\bar{x}$ instead of μ subtracts $n(\bar{x} - \mu)^2$ from the sum of squares. Now take expectations of both sides. Each term $\mathbb{E}[(x_i - \mu)^2]$ equals σ^2, so the first sum has expectation $n\sigma^2$. Meanwhile, the sample mean has variance σ^2/n, so $\mathbb{E}[n(\bar{x} - \mu)^2] = \sigma^2$. The sum of squared deviations therefore has expectation $(n-1)\sigma^2$, and dividing by $n - 1$ yields $\mathbb{E}[s^2] = \sigma^2$. $\blacksquare$

A helpful way to remember this factor is through *degrees of freedom*. The n deviations $x_i - \bar{x}$ are constrained to sum to zero, so only $n - 1$ of them can vary freely: estimating $\bar{x}$ from the data uses up one degree of freedom. The plug-in variance s_0^2 has bias $\frac{n-1}{n}\sigma^2 - \sigma^2 = -\sigma^2/n$, which matters at small n and vanishes as the sample grows.

One caveat is worth noting: unbiasedness does not survive nonlinear transformations. By Jensen's inequality applied to the concave square-root function, $\mathbb{E}[s] \leq \sqrt{\mathbb{E}[s^2]} = \sigma$, with strict inequality whenever s is not constant. For Gaussian data at $n = 2$, the sample standard deviation is $s = |x_1 - x_2|/\sqrt{2}$, whose expectation is $\mathbb{E}[s] = \sigma\sqrt{2/\pi} \approx 0.80\,\sigma$. Thus, taking the square root of an unbiased variance estimate still underestimates the standard deviation by 20% on average.

In numerical libraries, you choose between these two denominators with a "delta degrees of freedom" flag: `np.var(x, ddof=0)` divides by n, while `np.var(x, ddof=1)` divides by $n - 1$.

36.5.1.4 The Bias–Variance Decomposition

To see bias and variance interact, let us construct an estimator that deliberately trades one for the other. Suppose we observe $n = 30$ noisy measurements $x_i \sim \mathcal{N}(\theta, \sigma^2)$ with $\theta = 1$ and $\sigma = 4$. The ordinary sample mean $\bar{x}$ is unbiased, with variance $\sigma^2/n = 16/30 \approx 0.533$ (standard error 0.73). Now consider the *shrunk mean* $c\,\bar{x}$ for a factor $0 < c < 1$, which pulls the sample mean toward 0. Pulling toward zero introduces a bias of $c\theta - \theta = (c-1)\theta$, but it also compresses the variance to $c^2\sigma^2/n$. At $c = 0.65$, the bias is -0.35, while the standard error drops sharply from 0.73 to 0.47. Figure 36.22 shows both sampling distributions side by side.

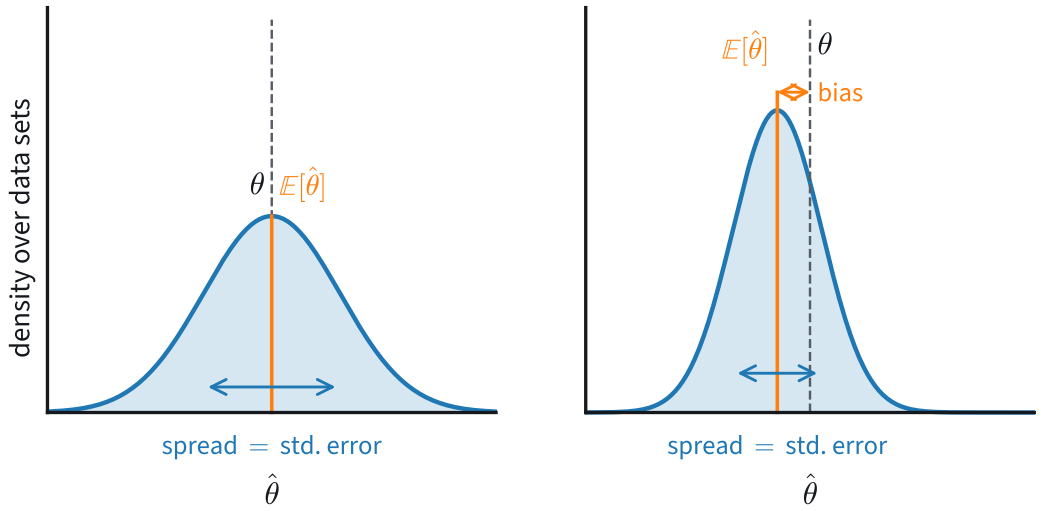


Figure 36.22: Sampling distributions of two estimators of $\theta = 1$ from 30 draws of $\mathcal{N}(1, 16)$. Left: the sample mean $\bar{x}$, with bias 0 and standard error 0.73. Right: the shrunk mean $0.65\,\bar{x}$, with bias -0.35 and standard error 0.47. Bias is the offset of the center from θ; standard error is the width.

Which estimator is actually better? The shrunk mean is systematically off target, yet because it scatters much less, any single draw from it often lands closer to θ than $\bar{x}$ does. To settle the comparison, we need a single score that accounts for both errors at once. The *mean squared error* (MSE) is the average squared distance from the estimator to the true parameter,

$$\mathrm{MSE}(\hat{\theta}_n) = \mathbb{E}\big[(\hat{\theta}_n - \theta)^2\big]. \tag{36.64}$$

Conveniently, the MSE decomposes cleanly into the two quantities we just defined.

Proposition (bias–variance decomposition). *If $\hat{\theta}_n$ has finite variance, then*

$$\mathrm{MSE}(\hat{\theta}_n) = \mathrm{Bias}(\hat{\theta}_n)^2 + \mathrm{Var}(\hat{\theta}_n). \tag{36.65}$$

Proof. Let $m = \mathbb{E}[\hat{\theta}_n]$ be the center of the estimator, and add and subtract m inside the square:

$$(\hat{\theta}_n - \theta)^2 = (\hat{\theta}_n - m)^2 + 2(\hat{\theta}_n - m)(m - \theta) + (m - \theta)^2.$$

Now take expectations term by term. The first term gives $\mathrm{Var}(\hat{\theta}_n)$, and the third is the constant $\mathrm{Bias}(\hat{\theta}_n)^2$. The cross term vanishes because $m - \theta$ is a constant and $\mathbb{E}[\hat{\theta}_n - m] = 0$ by definition of m. ■

There is a nice geometric way to picture this identity: it is Pythagoras' theorem in disguise. If we treat random variables as vectors equipped with the inner product $\langle U, V \rangle = \mathbb{E}[UV]$, then the zero-mean fluctuation $\hat{\theta}_n - m$ is orthogonal to every constant a, since $\langle \hat{\theta}_n - m,\, a \rangle = a\,\mathbb{E}[\hat{\theta}_n - m] = 0$. Writing the total error $\hat{\theta}_n - \theta$ as the sum of this fluctuation and the constant bias $m - \theta$ therefore breaks it into two perpendicular legs of lengths se and $|\mathrm{Bias}|$, whose squared lengths sum to the squared hypotenuse MSE. From this viewpoint, $\mathbb{E}[X]$ is the orthogonal projection of X onto the subspace of constants, the same projection we met in Section 33.1 with expectation playing the role of the dot product. Figure 36.23 illustrates this right triangle.

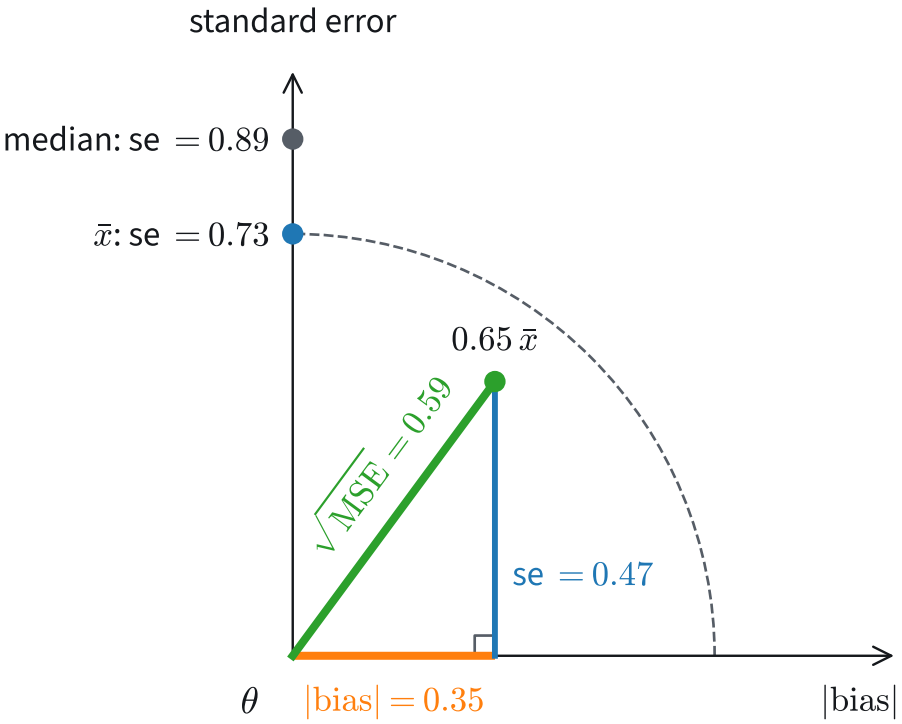


Figure 36.23: The bias–variance decomposition as Pythagoras' theorem for estimators of $\theta = 1$ from 30 draws of $\mathcal{N}(1, 16)$. Each estimator sits at horizontal coordinate $|\mathrm{bias}|$ and vertical coordinate se, so its distance from the origin is $\sqrt{\mathrm{MSE}}$. Shrinking $\bar{x}$ to $0.65\,\bar{x}$ adds a horizontal leg of 0.35 but cuts the vertical leg from 0.73 to 0.47, reducing the hypotenuse to 0.59 and placing it inside the dashed circle of estimators as accurate as $\bar{x}$. The sample median is unbiased but lies outside the circle.

With the decomposition in hand, we can rank estimators quantitatively. For our 30-

observation problem, the formula predicts an MSE of 0.533 for the sample mean (all variance) and $0.123 + 0.225 = 0.348$ for the shrunk mean $0.65\,\bar{x}$. What about the sample median? By symmetry of the Gaussian it is unbiased, though we do not yet have a formula for its variance. Let us simulate 10,000 data sets of size 30 and compare all three estimators empirically.

```
theta, sigma, n = 1.0, 4.0, 30
data = rng.normal(theta, sigma, (10000, n))   # 10,000 data sets of size 30
estimates = {'mean': data.mean(axis=1),
             '0.65 * mean': 0.65 * data.mean(axis=1),
             'median': np.median(data, axis=1)}
```

For each estimator in `estimates`, we now compute its empirical squared bias, variance, and total MSE across the 10,000 simulated datasets and compare the sum Bias^2 + Var against our analytical predictions:

```
predicted = {'mean': sigma**2 / n,            # bias^2 + variance, by formula
             '0.65 * mean': (0.35 * theta)**2 + 0.65**2 * sigma**2 / n,
             'median': None}                  # no formula yet
print(f'{"estimator":>12} {"bias^2":>7} {"var":>6} {"MSE":>6} {"predicted":>9}')
for name, est in estimates.items():
    bias2 = (est.mean() - theta) ** 2
    var = est.var()                           # a plain average: divides by the count
    mse = ((est - theta) ** 2).mean()
    pred = '-' if predicted[name] is None else f'{predicted[name]:.3f}'
    print(f'{name:>12} {bias2:7.3f} {var:6.3f} {mse:6.3f} {pred:>9}')
```

```
   estimator  bias^2    var    MSE predicted
        mean   0.000  0.532  0.532     0.533
0.65 * mean   0.125  0.225  0.350     0.348
      median   0.000  0.805  0.805         -
```

The simulation confirms the formulas: the sample mean has an MSE near 0.53, while the shrunk mean lands at $0.12 + 0.23 = 0.35$. The sample median trails at about 0.80; even though it is unbiased, its variance is roughly half again as large as that of $\bar{x}$. Notice the takeaway: a biased estimator can clearly outperform an unbiased one, and being unbiased is no guarantee of accuracy.

36.5.1.5 The Shrunk Mean and the Trade-off

This comparison immediately raises a practical question: how much bias should we accept in exchange for lower variance? For the shrunk mean $c\,\bar{x}$, we can solve this trade-off in closed form. Substituting its bias and variance into Equation 36.65 gives

$$\mathrm{MSE}(c) = (1-c)^2\theta^2 + c^2\frac{\sigma^2}{n}.$$

As we shrink c toward zero, the squared bias grows while the variance falls. Differentiating with respect to c and setting $-2(1-c)\theta^2 + 2c\,\sigma^2/n = 0$ yields the optimal shrinkage factor and its minimum error,

$$c^\star = \frac{\theta^2}{\theta^2 + \sigma^2/n}, \qquad \mathrm{MSE}(c^\star) = \frac{\theta^2\,\sigma^2/n}{\theta^2 + \sigma^2/n} < \frac{\sigma^2}{n}.$$

Notice that the optimal shrinkage always beats the unshrunk sample mean, and it helps most when the noise σ^2/n is large relative to the signal θ^2. For our running numbers ($\theta = 1$, $\sigma = 4$, $n = 30$), the formula gives $c^\star = 1/1.533 \approx 0.65$, the factor we chose earlier, and cuts the MSE from 0.533 down to 0.348. Figure 36.24 plots both error components and their sum against c. Here the classic U-shaped curve is exact, since it is simply the sum of a decreasing parabola and an increasing one.

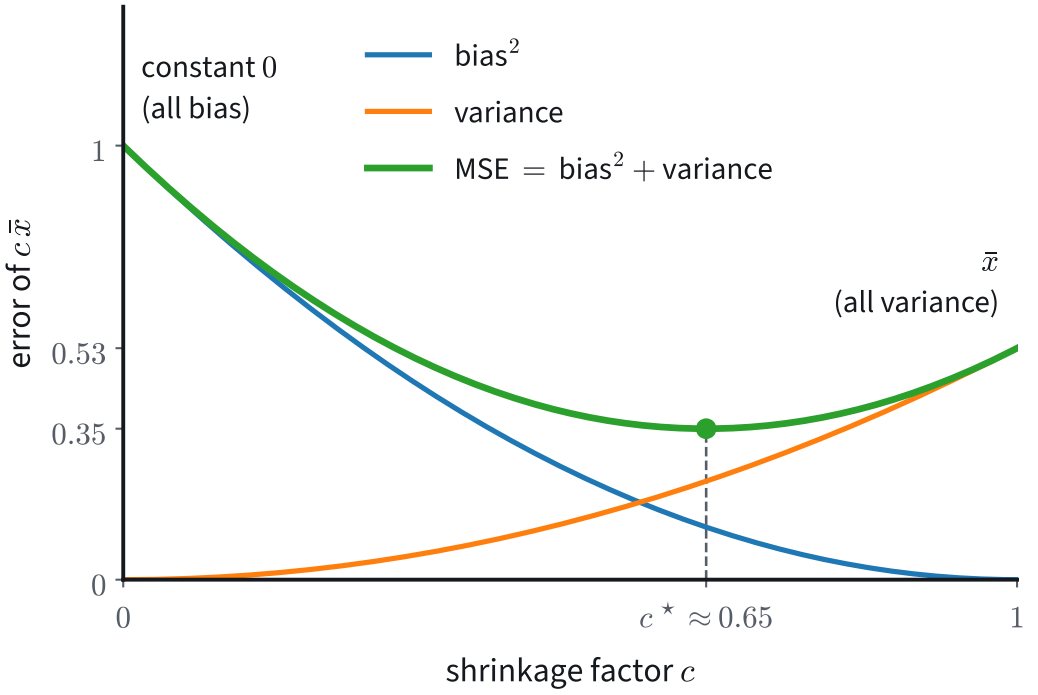


Figure 36.24: Squared bias and variance of the shrunk mean $c\,\bar{x}$ as functions of c, for $\theta = 1$, $\sigma = 4$, and $n = 30$. Their sum, the MSE, reaches its minimum at $c^\star \approx 0.65$, dropping to 0.35 compared with 0.53 for the unshrunk sample mean at $c = 1$.

Of course, there is a catch: $c^\star$ depends on the unknown parameter θ, so we cannot compute it directly from data. Worse, picking a fixed shrinkage factor without knowing θ can backfire badly. If the true mean were $\theta = 10$, using the same $c = 0.65$ would incur a bias of -3.5 and an MSE of $12.25 + 0.23 \approx 12.5$, more than twenty times worse than the 0.533 of $\bar{x}$. Shrinking toward 0 helps only when the truth actually lies near 0. Exercise 1 explores what happens when we plug an estimate of θ into $c^\star$, and exercise 9 reveals a famous surprise: in three or more dimensions, a similar data-driven shrinkage beats the sample mean for *every* value of θ.

This same bias–variance trade-off governs supervised prediction. At any fixed input x, the prediction $\hat{f}(x)$ of a model trained on a random data set is an estimator of the true regression function $f^*(x)$. Its expected squared error on a fresh label equals the MSE of that

estimator plus the irreducible variance of the label noise (Section 2.6). Across many model families, adding flexibility reduces bias while increasing variance, and regularization acts much like the shrinkage factor c (Geman et al. 1992). Keep in mind, though, that test error in modern overparameterized models does not always trace a simple U-shape: beyond the interpolation threshold where a model fits its training data exactly, test error can descend a second time (Section 36.10.3.2).

36.5.1.6 Consistency and Efficiency

Bias and variance evaluate an estimator at a fixed sample size n. A separate question is what happens as we gather more and more data: does the estimate eventually lock onto the truth? We say that an estimator $\hat{\theta}_n$ is *consistent* if it *converges in probability* to θ, meaning

$$P(|\hat{\theta}_n - \theta| > \varepsilon) \to 0 \quad \text{as } n \to \infty, \text{ for every } \varepsilon > 0,$$

written $\hat{\theta}_n \xrightarrow{P} \theta$. Intuitively, for any tolerance $\varepsilon > 0$, the probability of missing θ by more than ε shrinks to zero as the sample grows.

The bias–variance decomposition gives us a quick way to check consistency. Applying Markov's inequality (Section 36.1) to the nonnegative random variable $(\hat{\theta}_n - \theta)^2$ with threshold ε^2 bounds the probability of a large error directly by the MSE:

$$P(|\hat{\theta}_n - \theta| > \varepsilon) \le \frac{\mathrm{MSE}(\hat{\theta}_n)}{\varepsilon^2} = \frac{\mathrm{Bias}(\hat{\theta}_n)^2 + \mathrm{Var}(\hat{\theta}_n)}{\varepsilon^2}.$$

Consequently, if both the bias and the variance vanish as $n \to \infty$, the estimator is guaranteed to be consistent. Neither condition is enough on its own. Keeping only the first observation, $\hat{\theta}_n = x_1$, is unbiased for every n, yet its variance stays stuck at σ^2 so it never converges. Conversely, $\bar{x} + 1$ has variance $\sigma^2/n \to 0$, yet it converges to the wrong value. The plug-in variance s_0^2 illustrates the happy middle ground: even though it is biased at every finite n, its bias $-\sigma^2/n$ and variance (assuming a finite fourth moment) both vanish as n grows, so it is consistent.

Applying this bound to the sample mean yields the foundational result that makes averaging work.

Proposition (weak law of large numbers). *If $x_1, \dots, x_n$ are i.i.d. with mean μ and finite variance σ^2, then the sample mean $\bar{x}_n = \frac{1}{n}\sum_{i=1}^{n} x_i$ satisfies*

$$P(|\bar{x}_n - \mu| > \varepsilon) \le \frac{\sigma^2}{n\,\varepsilon^2} \quad \text{for every } \varepsilon > 0, \qquad \text{so} \qquad \bar{x}_n \xrightarrow{P} \mu. \tag{36.66}$$

Proof. By linearity of expectation, $\mathbb{E}[\bar{x}_n] = \mu$, so the bias is zero. Because the variables are independent, the variance of their sum is the sum of their variances, giving $\mathrm{Var}(\bar{x}_n) = n\sigma^2/n^2 = \sigma^2/n$. Substituting $\mathrm{MSE} = \sigma^2/n$ into the Markov bound above produces the inequality, whose right-hand side tends to zero as $n \to \infty$. ■

In fact, the law holds under the weaker assumption of a finite mean $\mathbb{E}|x_1| < \infty$, though the proof is longer (Wasserman 2013). This simple theorem underpins much of machine

learning: it explains why a Monte Carlo estimate of an integral converges, why accuracy on a large test set reflects true accuracy, and why average training loss tracks expected loss. That said, the bound itself is quite loose. For our 91.2%-accurate classifier on 1,000 questions, it only guarantees that the test accuracy misses θ by more than 3 percentage points with probability at most $0.0803/(1000{\cdot}0.03^2) \approx 0.09$, whereas the Gaussian approximation puts that probability near 0.001. Its real strength is generality: it assumes nothing beyond finite variance. In Section 36.9, we will derive far tighter bounds when stronger tail assumptions hold.

Consistency is only a baseline; it does not distinguish between two consistent estimators. Among unbiased estimators, the one with smaller variance is more *efficient*, meaning it achieves the same precision with fewer samples. For Gaussian data, both the sample mean and the sample median are unbiased for μ, but for large n the median's variance is roughly $\frac{\pi}{2}\,\sigma^2/n$ (exercise 6). The median therefore requires about $\pi/2 \approx 1.57$ times as much data to match the mean's standard error, giving it an efficiency of $2/\pi \approx 64\%$. Our $n = 30$ simulation already hinted at this ratio: 0.80 against 0.53. By contrast, the sample proportion achieves the smallest possible variance of any unbiased estimator of a Bernoulli parameter, a fundamental lower limit known as the Cramér–Rao bound (Section 36.6.3.4).

36.5.1.7 Monte Carlo Estimation

Up to now, our samples came from external data: test questions, noisy measurements, or training runs. Yet the law of large numbers does not care where the samples come from, which turns the sample mean into a powerful computational tool. Most expectations we encounter in deep learning (an expected loss, a normalizing constant, or the gradient of either) are integrals with no closed form, often spanning thousands of dimensions where numerical grids are hopelessly expensive. Even when we cannot build a grid, we can usually draw samples. The idea goes back to the earliest electronic computers (Metropolis and Ulam 1949): to approximate $\mathbb{E}[g(X)]$ for a random variable X with density p, draw independent samples $x_1, \dots, x_n$ from p and average the function values,

$$\mathbb{E}[g(X)] \approx \frac{1}{n}\sum_{i=1}^{n} g(x_i). \tag{36.67}$$

You can think of a **Monte Carlo** estimate as a Riemann sum whose evaluation points are chosen at random with equal weight $1/n$, automatically placing more points where the density p is high. By the law of large numbers Equation 36.66, this average is unbiased and converges to $\mathbb{E}[g(X)]$ as $n \to \infty$. Its convergence rate is governed by its variance: if $X_1, \dots, X_n$ are i.i.d. with density p and $\sigma^2 = \mathrm{Var}(g(X))$ is finite, then

$$\mathrm{Var}\Big(\frac{1}{n}\sum_{i=1}^{n} g(X_i)\Big) = \frac{\sigma^2}{n}. \tag{36.68}$$

There is nothing mathematically new here: this is just the σ^2/n variance of a sample mean from the law of large numbers, with $g(X_i)$ playing the role of x_i (Section 36.2). What is remarkable is the consequence. The root-mean-square error shrinks as $\sigma/\sqrt{n}$. On the face of it, that is slow: gaining one extra decimal digit of accuracy requires 100 times as many samples. Yet the dimension of x never appears in the exponent, and that dimension-independence is the entire reason sampling works in high dimensions. The one essential

requirement is finite variance; as we saw in Section 36.1, heavy tails can derail an average (see exercise 5).

Compare this with a deterministic grid. Reaching grid spacing ϵ in d dimensions requires $N = \epsilon^{-d}$ evaluation points. The left-endpoint rule of Section 34.6 has error proportional to $\epsilon = N^{-1/d}$, and a second-order rule such as the *midpoint rule* (evaluating each cell at its center) improves this to $\epsilon^2 = N^{-2/d}$. Monte Carlo, meanwhile, keeps its $N^{-1/2}$ rate in every dimension. In one dimension, the midpoint grid wins easily; at $d = 4$, the exponents tie; by $d = 10$, the grid slows to $N^{-1/5}$ while Monte Carlo still converges at $N^{-1/2}$. Although the numerator σ can still grow with d, the exponent never degrades (exercise 7).

Let us see this $1/\sqrt{n}$ rate in action on a concrete integral: $\int_0^1 e^{-x^2} dx = \mathbb{E}[e^{-U^2}]$ for U uniform on $[0, 1]$. Its exact value is $\frac{\sqrt{\pi}}{2}\operatorname{erf}(1) = 0.746824$. Below, we track the running average across powers of ten, comparing the actual error against the predicted scale $\sigma/\sqrt{n}$, where $\sigma = \operatorname{sd}(e^{-U^2}) \approx 0.2$ is estimated from the samples.

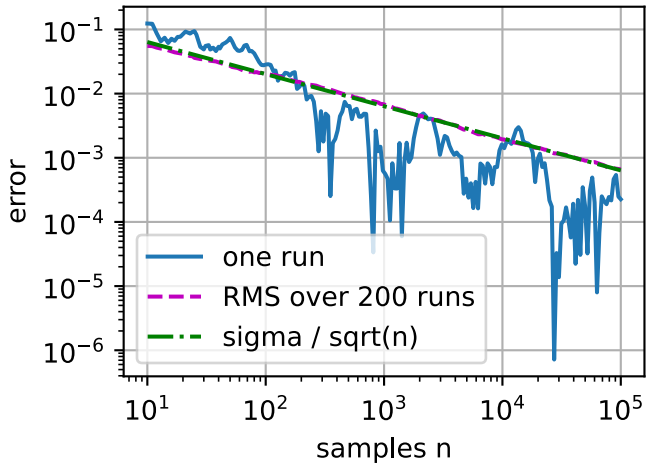
```
exact = math.sqrt(math.pi) / 2 * math.erf(1)
g = np.exp(-rng.uniform(size=10**5)**2)             # g(U), U uniform on [0, 1]
running = np.cumsum(g) / np.arange(1, 10**5 + 1)    # average of the first n
sigma = g.std()
for n in [10, 100, 1000, 10000, 100000]:
    print(f'n={n:<6}  estimate={running[n - 1]:.5f}  '
          f'error={abs(running[n - 1] - exact):.1e}  sigma/sqrt(n)={sigma / n**0.5:.1e}')
print(f'exact (sqrt(pi)/2) erf(1) = {exact:.6f}')
```

```
n=10      estimate=0.87045  error=1.2e-01  sigma/sqrt(n)=6.4e-02
n=100     estimate=0.77742  error=3.1e-02  sigma/sqrt(n)=2.0e-02
n=1000    estimate=0.74604  error=7.8e-04  sigma/sqrt(n)=6.4e-03
n=10000   estimate=0.74536  error=1.5e-03  sigma/sqrt(n)=2.0e-03
n=100000  estimate=0.74705  error=2.3e-04  sigma/sqrt(n)=6.4e-04
exact (sqrt(pi)/2) erf(1) = 0.746824
```

The observed errors track the predicted $\sigma/\sqrt{n}$ scale closely, though each row is still just one random realization: a single error rarely exceeds $\sigma/\sqrt{n}$ by more than a factor of two or three, and occasionally drops well below it. To see how the error behaves continuously across sample sizes, let us plot one running average at 200 log-spaced values of n alongside the root-mean-square error over 200 independent runs, which is the exact quantity predicted by Equation 36.68.

```
n = np.unique(np.logspace(1, 5, 200).astype(int))
sq = np.zeros(len(n))
for _ in range(200):                                # 200 independent runs
    g_run = np.exp(-rng.uniform(size=10**5)**2)
    sq += (np.cumsum(g_run)[n - 1] / n - exact)**2
rms = np.sqrt(sq / 200)                             # root-mean-square error
ratio = rms * np.sqrt(n) / sigma
print(f'RMS error / (sigma/sqrt(n)) over all n: {ratio.min():.2f} to {ratio.max():.2f}')
d2l.plot(n, [np.abs(running[n - 1] - exact), rms, sigma / np.sqrt(n)],
         'samples n', 'error', xscale='log', yscale='log',
         legend=['one run', 'RMS over 200 runs', 'sigma / sqrt(n)'])
```

```
RMS error / (sigma/sqrt(n)) over all n: 0.87 to 1.08
```



Notice the difference between the two curves. A single running average wanders above and below the theoretical line because its values at nearby n share almost all the same samples, and it plunges sharply whenever the error crosses zero. By contrast, the root-mean-square error averaged over 200 independent runs stays within about 20% of $\sigma/\sqrt{n}$ at every n, falling steadily with slope $-\frac{1}{2}$ across all four orders of magnitude.

36.5.2 Hypothesis Testing

An error bar tells us how uncertain a single estimate is, but most decisions in machine learning compare two of them: did a new architecture, optimizer, or regularization trick actually beat the baseline? In this subsection, we first build a comparison test from scratch by simulation, then derive its closed-form counterpart, and finally ask how much data an experiment needs to reliably detect a real gain.

36.5.2.1 Comparison of Two Models

Training a neural network is inherently random: weight initialization, data shuffling, and dropout masks all vary with the random seed, so running the exact same training script twice produces slightly different test accuracies. Suppose we train model A and model B across 20 seeds each and record their test accuracies. Imagine that seed-to-seed noise shifts a single run by roughly one percentage point, while model B is truly better by 0.8 percentage points. Given only those 40 numbers, can we tell that B is better? The natural summary is the difference between the two mean accuracies, which will serve as our *test statistic*.

Every hypothesis test begins by formulating the skeptical claim it tries to overturn. Here, the *null hypothesis* H_0 asserts that both lists of accuracies come from the same distribution, so the two models perform equally well on average. The *alternative hypothesis* H_A asserts that their distributions differ. We then ask: if H_0 were true, how surprising would it be to see a gap at least as large as ours? If the gap would be very unlikely under chance alone, we *reject* H_0; otherwise, we *fail to reject* it. That asymmetric phrasing is deliberate: failing to reject H_0 does not prove the models are equal, just as a verdict of "not guilty" in court does not prove innocence.

36.5.2.2 The Permutation Test

To judge whether an observed gap is surprising under H_0, we need to know its distribution when H_0 holds, called the *null distribution*. Remarkably, we can construct this distribution directly from the data without assuming Gaussianity or any other parametric form, using a *permutation test* (Fisher 1935). The logic is simple: if both lists truly came from the same distribution, then the labels "A" and "B" carry no information at all. Any reassignment of the 40 numbers into two groups of 20 would have been just as likely as the split we observed. By shuffling the labels, recomputing the mean gap, and repeating this R times, we map out the range of gaps that pure chance produces. The helper below vectorizes all R shuffles at once and counts how often a shuffled gap is at least as large in magnitude as the observed one.

```
def perm_p(acc_a, acc_b, rng, R=2000):
    """Two-sided permutation p-value and the R shuffled gaps."""
    k = len(acc_a)
    observed = acc_b.mean() - acc_a.mean()
    pooled = np.tile(np.concatenate([acc_a, acc_b]), (R, 1))
    shuffled = rng.permuted(pooled, axis=1)    # R relabelings
    gaps = shuffled[:, k:].mean(1) - shuffled[:, :k].mean(1)
    p = (1 + (np.abs(gaps) >= abs(observed)).sum()) / (R + 1)
    return p, gaps
```

Including the observed labeling as one of the $R + 1$ shuffles yields the *p-value*,

$$p = \frac{1 + \#\{r : |\text{gap}_r| \geq |\text{gap}_{\text{obs}}|\}}{R + 1},$$

which estimates the probability, under H_0, of observing a gap at least as extreme as the one we actually saw. A tiny p-value means that such a large gap would rarely occur if the labels were meaningless. Notice that this p-value is *two-sided*: large gaps in either direction count against H_0. (If we asked only whether B beats A, a one-sided test would count only shuffled gaps at least as large as the observed positive gap.)

Before looking at the data, we commit to a *significance level* α, conventionally $\alpha = 0.05$, and agree to reject H_0 whenever $p \leq \alpha$. Let us run one simulated experiment with the numbers above to see the test in action.

```
k = 20                                      # training seeds per model
acc_a = rng.normal(0.850, 0.010, k)         # per-seed accuracy, model A
acc_b = rng.normal(0.858, 0.010, k)         # model B: a true gap of 0.008
p, gaps = perm_p(acc_a, acc_b, rng, R=10000)
observed = acc_b.mean() - acc_a.mean()
print(f'observed gap {observed:.4f}, permutation p-value {p:.3f}')
```

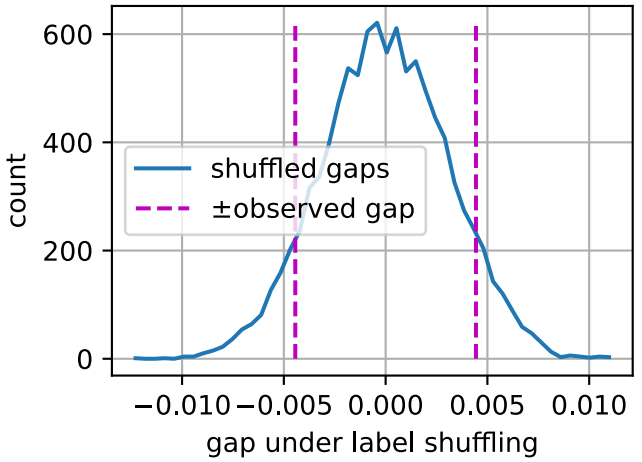
```
observed gap 0.0044, permutation p-value 0.158
```

Because each experiment draws fresh seeds, the observed gap fluctuates around the true 0.008 from run to run. In most runs, p drops below 0.05 and the test rejects H_0, yet in a noticeable minority of runs it fails to reject, as may happen here. With a full percentage point of seed noise, 20 seeds per model are not always enough to catch a 0.8-point gain. Shortly,

we will measure precisely how often this happens and see that it reflects the experiment's sample size rather than a defect in the test.

Plotting a histogram of the shuffled gaps alongside the observed gap (and its negative mirror image) lets us see the null distribution visually.

```
counts, edges = np.histogram(gaps, bins=50)
centers = (edges[:-1] + edges[1:]) / 2
top = counts.max()
d2l.plot([centers, [observed] * 2, [-observed] * 2],
         [counts, [0, top], [0, top]], 'gap under label shuffling', 'count',
         legend=['shuffled gaps', '±observed gap'],
         fmts=('-', 'm--', 'm--'))
```



Even though the permutation test assumed nothing about the shape of the data, the shuffled gaps settle into a neat bell curve centered at zero, with the observed gap typically landing out in one tail. Best of all, this exact same recipe works for *any* two-group summary statistic: a difference of medians, error rates, or validation losses. (When comparing means in closed form, its classical counterpart is Welch's t-test.) Every hypothesis test follows the same discipline: state H_0 and H_A, fix α and choose a sample size large enough to detect a meaningful effect, and only then collect the data, compute the p-value, and reject H_0 if $p \leq \alpha$.

36.5.2.3 The z-Test

While shuffling builds the null distribution empirically, assuming Gaussian data with a known variance gives us closed-form formulas for both the p-value and the detection rate. Suppose $x_1, \ldots, x_n$ are i.i.d. $\mathcal{N}(\mu, \sigma^2)$ with σ known, and we wish to test $H_0 : \mu = 0$. We standardize the sample mean by its standard error to form the test statistic

$$T = \frac{\sqrt{n}\,\bar{x}}{\sigma}.$$

Under H_0, $\bar{x} \sim \mathcal{N}(0, \sigma^2/n)$, so T follows a standard normal distribution. Write Φ for

the standard normal c.d.f. and z_q for its q-quantile, defined by $\Phi(z_q) = q$ (for instance, $z_{0.975} \approx 1.96$). The two-sided p-value is $2(1-\Phi(|T|))$, so the condition $p \le \alpha$ is equivalent to $|T| \ge z_{1-\alpha/2}$. At $\alpha = 0.05$, we reject H_0 whenever the sample mean lands more than 1.96 standard errors away from 0. Figure 36.25 highlights these two rejection tails.

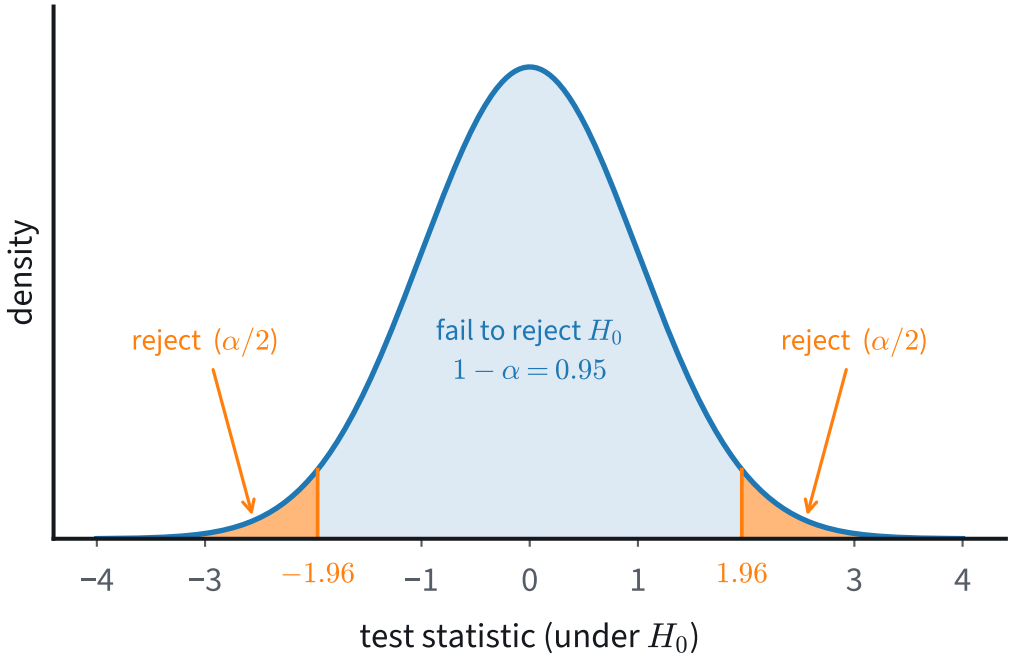


Figure 36.25: Rejection region of the two-sided z-test at $\alpha = 0.05$. Under H_0, the standardized statistic follows a standard normal distribution: the central region holds probability $1-\alpha = 0.95$, while the two tails beyond ± 1.96 together hold α. Observing a statistic in either tail would be rare under H_0, so we reject H_0.

Our two-model comparison has a direct two-sample version of this test. With k independent seeds per model and per-seed standard deviation σ, the mean gap $\bar{x}_B - \bar{x}_A$ has variance $2\sigma^2/k$, so $T = (\bar{x}_B - \bar{x}_A)/(\sigma\sqrt{2/k})$ is standard normal under H_0. Plugging in $\sigma = 0.01$ and $k = 20$ yields a standard error of 0.0032 for the gap. A true improvement of 0.008 therefore sits 2.5 standard errors above zero, past the 1.96 threshold, though not by a comfortable margin.

36.5.2.4 Two Kinds of Error and Power

Any hypothesis test can misfire in two ways, and good experimental design keeps both under control. A *type I error* (or false positive) occurs when we reject a true H_0, claiming an improvement that does not exist. A *type II error* (or false negative) occurs when we fail to reject a false H_0, missing a genuine improvement. Their probabilities are

$$\alpha = P(\text{reject } H_0 \mid H_0 \text{ true}), \qquad \beta = P(\text{fail to reject } H_0 \mid H_0 \text{ false}),$$

where β is evaluated at a specific alternative, such as a true gap of 0.8 percentage points.

The complement $1-\beta$ is called the *power* of the test: the probability that the experiment actually detects that real effect. Figure 36.26 summarizes the four possible outcomes.

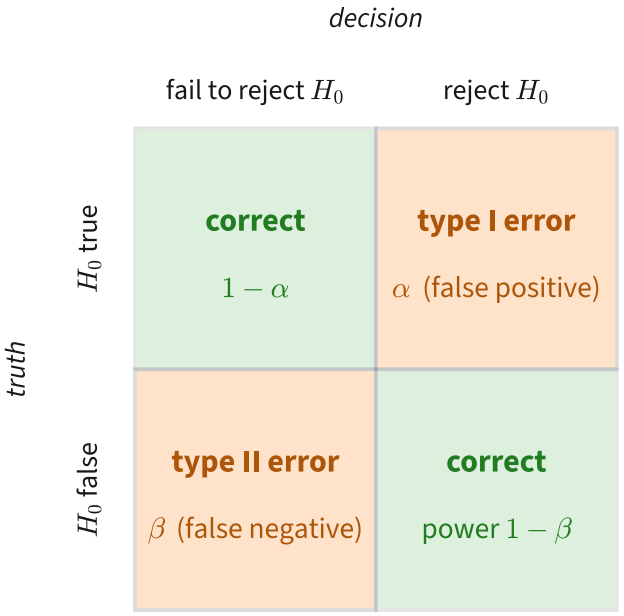


Figure 36.26: The 2×2 decision matrix of a hypothesis test, showing correct decisions on the diagonal, type I error (rate α) and type II error (rate β) off the diagonal, and statistical power ($1-\beta$) in the correct-rejection cell.

Why does the simple rule "reject when $p \le \alpha$" guarantee a type I error rate of exactly α? Because under H_0, the p-value is uniformly distributed on $[0,1]$. This fact is worth proving: not only does it explain what a significance level means, it is also the key to understanding multiple testing later on.

Proposition (p-values are uniform under the null). *Let the test statistic T have a continuous c.d.f. F under H_0, and let $p = 1 - F(T)$ (one-sided) or $p = 2\min\{F(T), 1-F(T)\}$ (two-sided). Then $P_{H_0}(p \le \alpha) = \alpha$ for every $\alpha \in [0,1]$.*

Proof. We first show that $U = F(T)$ is uniform on $[0,1]$. For any $u \in (0,1)$, let t_u be the largest value of t satisfying $F(t) = u$; such a value exists because F is continuous and runs from 0 to 1. Since F is nondecreasing, $F(T) \le u$ holds if and only if $T \le t_u$, which gives $P(U \le u) = F(t_u) = u$. For the one-sided p-value, it follows immediately that $P(1-U \le \alpha) = P(U \ge 1-\alpha) = \alpha$. For the two-sided p-value, the event $2\min\{U, 1-U\} \le \alpha$ means either $U \le \alpha/2$ or $U \ge 1-\alpha/2$; these two disjoint events each have probability $\alpha/2$, so their union has probability α. ∎

For the permutation p-value, an even simpler counting argument gives the same guarantee. Under H_0, the observed labeling and the R random shuffles are completely exchangeable, so the observed gap is equally likely to fall at any of the $R+1$ ranks. Consequently, $p \le \alpha$ happens only when the observed gap ranks among the $\lfloor \alpha(R+1) \rfloor$ largest in magnitude, yielding $P_{H_0}(p \le \alpha) \le \alpha$ (ties can only increase p). That is why the $+1$ in the numerator

and denominator matters: it ensures that a level-α test rejects a true null at most a fraction α of the time.

What happens to the distribution of p-values when H_0 is false? To see the contrast, let us simulate 2,000 two-sample z-tests with 20 seeds per model, first with no true gap and then with our true gap of 0.008, and histogram the resulting p-values into ten equal bins.

```
se = 0.010 * np.sqrt(2 / k)                 # se of the gap, k seeds per model
for gap in (0.0, 0.008):
    acc_a = rng.normal(0.850, 0.010, (2000, k))
    acc_b = rng.normal(0.850 + gap, 0.010, (2000, k))
    T = (acc_b.mean(axis=1) - acc_a.mean(axis=1)) / se
    p = 2 * norm.sf(np.abs(T))               # two-sided z-test p-values
    hist = np.histogram(p, bins=10, range=(0, 1))[0] / 2000
    print(f'gap {gap:.3f}:', ' '.join(f'{h:.2f}' for h in hist),
          f'  p <= 0.05: {np.mean(p <= 0.05):.2f}')
```

```
gap 0.000: 0.10 0.10 0.11 0.10 0.09 0.10 0.10 0.10 0.10 0.10   p <= 0.05: 0.05
gap 0.008: 0.82 0.08 0.04 0.02 0.01 0.01 0.01 0.01 0.01 0.01   p <= 0.05: 0.72
```

When H_0 holds (gap 0.000), the histogram is flat: roughly one-tenth of the p-values fall in each bin, and about 5% land at or below 0.05. When there is a real gap of 0.008, the p-values pile up sharply near zero: about 80% fall below 0.1, and 70% to 74% fall at or below 0.05. That rejection fraction is the power of the test, which our closed-form formula below predicts to be 0.72.

Before deriving that formula, it is worth guarding against two classic misreadings of a p-value (Wasserstein and Lazar 2016). First, a p-value is the probability of seeing data at least this extreme *assuming H_0 is true*; it is *not* the probability that H_0 is true (which would require a prior and Bayes' rule). Second, a large p-value does not prove that H_0 holds; it only tells us that the experiment failed to detect an effect, perhaps because it lacked the power to do so.

Let us compute that power explicitly for the one-sample z-test. When the true mean is $\mu \neq 0$, the statistic T is normal with mean $\sqrt{n}\,\mu/\sigma$ and variance 1. Defining the *standardized effect* $\delta = |\mu|/\sigma$ (the effect size measured in units of the noise), the test rejects whenever $|T| \geq z_{1-\alpha/2}$, which happens with probability

$$1 - \beta \approx \Phi\big(\delta\sqrt{n} - z_{1-\alpha/2}\big),$$

after dropping the negligible opposite tail $\Phi(-\delta\sqrt{n} - z_{1-\alpha/2})$. Thus power rises with the effect size δ and with $\sqrt{n}$. For a large effect $\delta = 1$, even $n = 8$ observations give power $\Phi(2.83 - 1.96) \approx 0.81$.

In our two-sample model comparison, the statistic has mean $\delta\sqrt{k/2}$ with $\delta = 0.008/0.010 = 0.8$, so the z-test power with k seeds per model is $\Phi(0.8\sqrt{k/2} - 1.96)$: about 0.72 with 20 seeds, and only 0.24 with 5 seeds. Because that formula assumes we know σ in advance, let us check how the permutation test, which estimates nothing ahead of time, performs across 400 fresh simulated experiments for each seed budget.

```python
def power(k, gap, rng, num_experiments=400):
    """Fraction of fresh experiments in which the permutation test rejects."""
    rejections = 0
    for _ in range(num_experiments):
        acc_a = rng.normal(0.850, 0.010, k)
        acc_b = rng.normal(0.850 + gap, 0.010, k)
        rejections += perm_p(acc_a, acc_b, rng)[0] <= 0.05
    return rejections / num_experiments

for k in (20, 5):
    print(f'{k:2d} seeds: rejects in {power(k, 0.008, rng):.2f} of runs, '
          f'formula {norm.cdf(0.8 * np.sqrt(k / 2) - 1.96):.2f}')
```

```
20 seeds: rejects in 0.70 of runs, formula 0.72
 5 seeds: rejects in 0.16 of runs, formula 0.24
```

Across 400 runs, the permutation test rejects in about 70% of experiments with 20 seeds and only about 20% with 5 seeds (each rate fluctuates by roughly two percentage points from run to run). Both numbers sit slightly below the z-test formula because the permutation test effectively estimates the noise from the data, which costs a small amount of power. Even with 20 seeds, then, our experiment misses a genuine 0.8-point gain roughly three times in ten; with only 5 seeds, it misses four times in five. Failing to reject with a power of 0.2 tells us almost nothing. In practice, experimenters typically aim for a power of at least 0.8.

36.5.2.5 Sample Size

Rather than discovering after the fact that an experiment was underpowered, we can plan the sample size up front. Setting the power formula equal to $1-\beta$ gives $\delta\sqrt{n} - z_{1-\alpha/2} = z_{1-\beta}$, and solving for n yields the sample size needed to detect a standardized effect δ at significance level α with power $1-\beta$:

$$n = \frac{\left(z_{1-\alpha/2} + z_{1-\beta}\right)^2}{\delta^2}. \tag{36.69}$$

Notice the steep $1/\delta^2$ scaling: cutting the target effect size in half requires four times as much data. For the standard benchmark of $\alpha = 0.05$ and power 0.8, we have $z_{0.975} \approx 1.960$ and $z_{0.8} \approx 0.842$, which simplifies to the handy rule of thumb $n \approx 7.85/\delta^2$. Figure 36.27 plots the resulting power curves across effect sizes.

When comparing two independent groups of k observations each with common standard deviation σ, the variance of the difference doubles to $\operatorname{Var}(\bar{x}_B - \bar{x}_A) = 2\sigma^2/k$. Accordingly, the required sample size per group doubles as well:

$$k = \frac{2\left(z_{1-\alpha/2} + z_{1-\beta}\right)^2}{\delta^2}, \qquad \delta = \frac{|\mu_B - \mu_A|}{\sigma}. \tag{36.70}$$

For our model comparison with $\delta = 0.8$, reaching 80% power requires $k = 2 \cdot 7.85/0.64 \approx 25$ seeds per model, which explains why 20 seeds fell just shy of the mark.

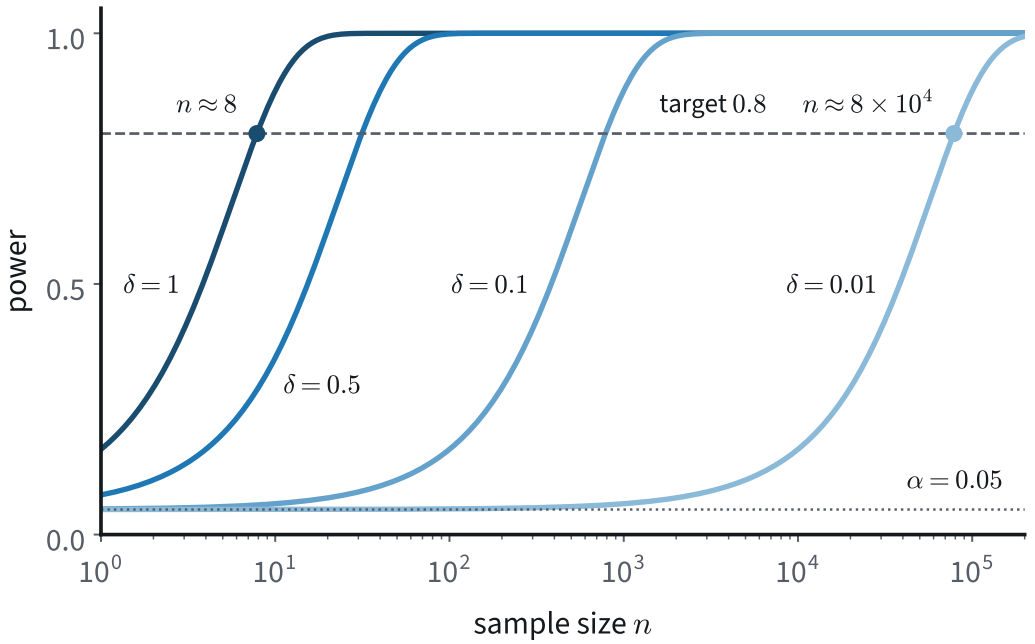


Figure 36.27: Power of the two-sided z-test at $\alpha = 0.05$ versus sample size n, with one curve per standardized effect δ. Each curve starts near α (dotted line), where the test rejects only by chance, and climbs toward 1. The dashed line marks the 0.8 power target, reached at $n \propto 1/\delta^2$: about 8 observations for $\delta = 1$, rising to about 78,500 for $\delta = 0.01$.

36.5.2.6 Paired Comparisons

Let us now return to our opening puzzle: two classifiers that scored 912 and 921 out of 1,000. Earlier we noted that treating their scores as independent left the gap buried in noise. Crucially, though, both models answered the *same* 1,000 questions. Some questions are easy for both models and others are hard for both, and an unpaired analysis lumps all of that shared question difficulty into the noise.

A *paired* comparison strips out that shared variation by working directly with the per-question differences $d_i \in \{-1, 0, 1\}$, defined as model B's correctness minus model A's on question i. Whenever both models agree, $d_i = 0$. Suppose they disagree on 41 questions: model B alone is right on 25, and model A alone is right on 16, giving the net lead of $+9$ (0.9 percentage points). Under the null hypothesis that the two models are interchangeable, each disagreement is like a fair coin flip, equally likely to favor A or B. The appropriate null simulation therefore flips the sign of each nonzero d_i at random, a procedure known as the *sign-flip permutation test*. Let us run it on these 41 disagreements.

```
d = np.r_[np.ones(25), -np.ones(16), np.zeros(959)]   # B only, A only, agree
nonzero = d[d != 0]                                   # agreements never change
signs = rng.choice([-1, 1], size=(100000, len(nonzero)))
flipped = (signs * nonzero).sum(axis=1) / len(d)      # gaps under H_0
p = (1 + (np.abs(flipped) >= abs(d.mean())).sum()) / (len(flipped) + 1)
print(f'gap {d.mean():.3f}, sign-flip p-value {p:.2f}')
```

```
gap 0.009, sign-flip p-value 0.21
```

The sign-flip test returns $p \approx 0.21$ (replacing random sign flips with the exact binomial probability gives McNemar's test). Compare this with an unpaired analysis of the exact same counts, which yields a gap of only 0.7 standard errors and $p \approx 0.47$. Pairing cuts the standard error of the gap in half, from 0.012 down to $\sqrt{41}/1000 \approx 0.006$, because the 959 questions on which the models agree no longer inject noise. Even so, a 25-to-16 split among 41 disagreements is still too small to be convincing: a nine-question margin on a 1,000-question test does not establish that model B is better.

Pairing matters most when you design an evaluation from scratch. Suppose two models have true accuracies of 90% and 91% and disagree on 4% of questions (B alone right on 2.5%, A alone on 1.5%). Then d_i has mean 0.01 and variance $\mathbb{E}[d_i^2] - 0.01^2 = 0.04 - 0.0001 = 0.0399$, giving a standardized effect of $\delta = 0.01/\sqrt{0.0399} \approx 0.050$. By Equation 36.69, a paired test reaches 80% power with about $7.85 \cdot 0.0399/0.01^2 \approx 3{,}100$ questions. By contrast, evaluating the two models on separate, independent test sets leaves an unpaired variance of $0.9 \cdot 0.1 + 0.91 \cdot 0.09 = 0.1719$ per pair of questions, requiring roughly $7.85 \cdot 0.1719/0.01^2 \approx 13{,}500$ questions *per model*.

In this example, pairing shrinks each test set by more than a factor of four and reduces total labeling effort by nearly a factor of nine, and the savings grow even larger when the two models agree more frequently. The practical rule is clear: always evaluate competing models on the same test set and analyze the per-question differences. The same principle applies to random seeds: when two models share the same training seeds, flip the signs of the per-seed differences (exercise 5).

36.5.2.7 Multiple Testing

Every significance guarantee so far, $P(\text{reject} \mid H_0) = \alpha$, assumes we run a *single* test chosen in advance. In machine learning, however, hyperparameter sweeps routinely run dozens of comparisons at once. Suppose you compare 20 new configurations against a baseline when none of them is actually better, testing each at $\alpha = 0.05$. If the tests are independent, the probability that at least one configuration falsely looks significant is $1 - 0.95^{20} \approx 0.64$. Cherry-picking only the winners, or repeatedly tweaking an analysis until $p \le \alpha$ (*p-hacking*), reliably turns pure noise into publishable-looking gains.

The simplest safeguard is the *Bonferroni correction* (Bonferroni 1936): when running m tests, lower each individual threshold to α/m. Writing A_j for the event that test j falsely rejects a true null, the union bound gives

$$P\Big(\bigcup_{j=1}^{m} A_j\Big) \le \sum_{j=1}^{m} P(A_j) = m \cdot \frac{\alpha}{m} = \alpha.$$

Notice how remarkably general this bound is: it caps the probability of making *any* false discovery at α without assuming anything about how the m tests are correlated. For 20 independent tests run at $0.05/20$, the chance of at least one false positive drops from 0.64 to $1 - (1 - 0.0025)^{20} \approx 0.049$.

Unfortunately, when m reaches the hundreds or thousands, dividing by m makes the Bonferroni threshold so strict that genuine discoveries get buried. In large-scale screening, statisticians instead control the *false discovery rate*, the expected fraction of false positives among the hypotheses we reject. The Benjamini–Hochberg procedure (Benjamini and Hochberg 1995) keeps this rate at or below α whenever the p-values are independent or positively dependent; exercise 2 introduces the rule and tests it in simulation.

36.5.3 Confidence Intervals

A p-value answers a binary question about a single candidate parameter value. In practice, we usually want more: the entire range of parameter values consistent with the data, reported directly as an error bar around our estimate.

36.5.3.1 Definition and Interpretation

A *confidence interval* is produced by a rule guaranteed to capture the fixed, unknown parameter in a specified fraction of repeated experiments (Neyman 1937). Formally, a $1 - \alpha$ confidence interval is a random interval C_n, computed from the data, satisfying

$$P_\theta(C_n \ni \theta) \ge 1 - \alpha \quad \text{for every } \theta. \tag{36.71}$$

We deliberately write $C_n \ni \theta$ rather than $\theta \in C_n$ to emphasize where the randomness sits: θ is a fixed number, while the interval C_n shifts from one data set to the next. The probability $1 - \alpha$ is a property of the *procedure*. Across many independent experiments, at least 95% of the resulting 95% confidence intervals will cover θ. Any single interval you compute either contains θ or misses it, and the interval itself carries no hint of which happened. Figure 36.28 illustrates this with 100 intervals drawn from 100 independent samples. Many intervals used in practice satisfy this guarantee only in the large-sample limit: an *asymptotic* $1 - \alpha$ interval satisfies $\liminf_{n\to\infty} P_\theta(C_n \ni \theta) \ge 1 - \alpha$ for every θ.

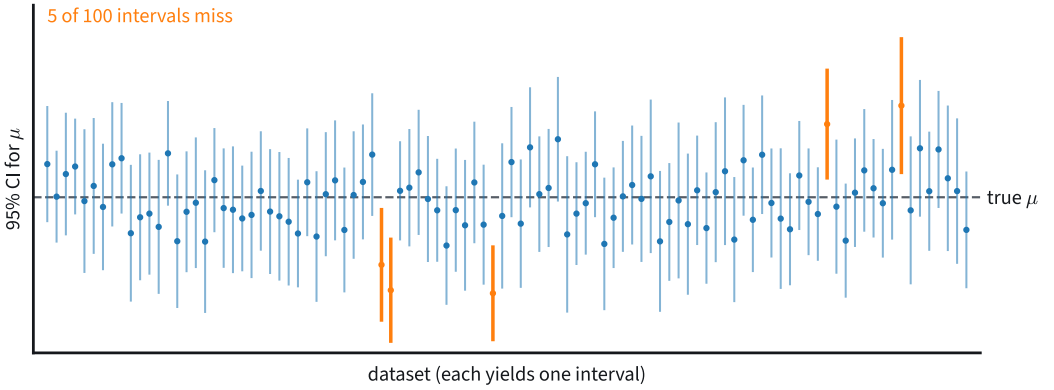


Figure 36.28: One hundred 95% confidence intervals, each constructed from its own independent sample, plotted against the fixed true mean (dashed line). Most intervals cover the truth while the orange ones miss. The 95% guarantee describes the long-run hit rate of the procedure; an individual interval gives no sign of whether it succeeded.

36.5.3.2 The Gaussian Interval

Let us start with the cleanest case, where every step is exact. Suppose $x_1, \dots, x_n$ are i.i.d. $\mathcal{N}(\mu, \sigma^2)$ with σ known. Because $\bar{x} \sim \mathcal{N}(\mu, \sigma^2/n)$, setting the half-width to $h = z_{1-\alpha/2}\,\sigma/\sqrt{n}$ ensures that $\bar{x}$ lands within h of μ with probability $1-\alpha$. Now flip that inequality around so it reads as a statement about μ:

$$|\bar{x} - \mu| \le h \quad \Longleftrightarrow \quad \mu \in [\bar{x} - h,\ \bar{x} + h].$$

Hence $P_\mu(\mu \in [\bar{x} - h, \bar{x} + h]) = 1 - \alpha$ for every μ. That simple algebraic reversal is the entire idea behind a confidence interval. At $\alpha = 0.05$, the interval is $\bar{x} \pm 1.96\,\sigma/\sqrt{n}$.

In real problems σ is rarely known, so we substitute the sample standard deviation s from Equation 36.63. Once s is estimated from the data, the studentized statistic $T = (\bar{x} - \mu)/(s/\sqrt{n})$ is no longer normal. For Gaussian data, it follows Student's t distribution with $n-1$ degrees of freedom (Student (Gosset 1908), which has heavier tails than the normal distribution to account for the extra randomness in s. (We take this distribution on faith here; exercise 4 derives it as a Gaussian with random precision.) Replacing $z_{1-\alpha/2}$ by the t-quantile $t_{n-1,1-\alpha/2}$ yields

$$\left[\bar{x} - t_{n-1,1-\alpha/2}\,\frac{s}{\sqrt{n}},\ \bar{x} + t_{n-1,1-\alpha/2}\,\frac{s}{\sqrt{n}}\right], \tag{36.72}$$

which achieves coverage $1-\alpha$ for Gaussian data at every $n \ge 2$. At small sample sizes, this distinction matters enormously: $t_{n-1,0.975}$ is 12.71 for $n = 2$, 2.776 for $n = 5$, and 1.984 for $n = 100$, converging toward 1.96 as n grows. The workhorse interval used throughout applied work simply plugs in the large-sample limit,

$$\left[\bar{x} - 1.96\,\frac{s}{\sqrt{n}},\ \bar{x} + 1.96\,\frac{s}{\sqrt{n}}\right], \tag{36.73}$$

an asymptotic 95% interval that is slightly too narrow at every finite n. For Gaussian data, its actual coverage is $P(|t_{n-1}| \le 1.96)$, which is close to nominal at $n = 100$ (0.947) but drops to 0.878 at $n = 5$.

What if the data are not Gaussian? By the central limit theorem (Section 36.3.2.3), $\sqrt{n}(\bar{x} - \mu)/\sigma$ converges to a standard normal whenever the variance is finite, and s converges to σ by the law of large numbers. Consequently, Equation 36.73 remains an asymptotic 95% interval for the mean of *any* distribution with finite variance. How large n needs to be depends on the shape of the data: skewed distributions need more samples (exercise 3), and accuracies close to 100% need far more, since many test sets produce a perfect score with zero sample variance (exercise 4). In all cases, the half-width scales as $1/\sqrt{n}$, so cutting an error bar in half takes four times the data.

Because a test accuracy is the mean of 0/1 indicators with $s^2 \approx \hat{\theta}(1-\hat{\theta})$, the same formula gives $\hat{\theta} \pm 1.96\sqrt{\hat{\theta}(1-\hat{\theta})/n}$, known as the *Wald interval*. Let us evaluate it for our 1,000-question classifier and for our running coin example (9 heads in 13 flips).

```
for n_correct, n in ((912, 1000), (9, 13)):
    theta_hat = n_correct / n
    half = 1.96 * np.sqrt(theta_hat * (1 - theta_hat) / n)
    print(f'{n_correct:3d} of {n:4d}: '
          f'({theta_hat - half:.3f}, {theta_hat + half:.3f})')
```

```
912 of 1000: (0.894, 0.930)
  9 of   13: (0.441, 0.943)
```

For the classifier, we obtain $91.2\% \pm 1.8$ percentage points. For the coin, the interval $(0.44, 0.94)$ comfortably includes $\frac{1}{2}$, confirming that 13 flips are far too few to conclude the coin is biased. Pinning its bias down to ± 0.05 would take roughly $1.96^2 \cdot 0.69 \cdot 0.31 / 0.05^2 \approx 330$ flips.

It is illuminating to contrast this with a Bayesian *credible interval* for the same coin. A Bayesian analysis treats θ itself as random, puts a prior on it, and reports an interval containing posterior probability $1 - \alpha$ given the observed flips. With a uniform prior, the posterior is Beta$(10, 5)$ (Section 36.4), whose central 95% credible interval (between the 2.5% and 97.5% posterior quantiles) is $(0.42, 0.87)$. Although the numbers look similar here, the two intervals make fundamentally different claims. The credible interval says that, under the chosen prior, θ lies in $(0.42, 0.87)$ with probability 0.95. The confidence interval says that its construction procedure traps the fixed true θ in at least 95% of repeated experiments, regardless of what θ is. Keep in mind, too, that the Wald interval meets its guarantee only asymptotically: with 13 flips, its actual coverage is around 91% at $\theta = \frac{1}{2}$ and drops below 50% at $\theta = 0.05$.

Because coverage is a frequency across repeated experiments, we can test it directly by simulation. Let us draw 1,000 independent data sets of $n = 100$ standard normal observations, compute the asymptotic interval Equation 36.73 on each, and count how many cover the true mean 0.

```
mu, n, trials = 0.0, 100, 1000
data = rng.normal(mu, 1.0, (trials, n))          # one row = one data set
mu_hat = data.mean(axis=1)
```

```
se = data.std(axis=1, ddof=1) / np.sqrt(n)
hits = np.abs(mu_hat - mu) <= 1.96 * se          # does the interval contain mu?
print(f'first interval ({mu_hat[0] - 1.96 * se[0]:.3f}, '
      f'{mu_hat[0] + 1.96 * se[0]:.3f}) contains 0: {hits[0]}')
print(f'{hits.sum()} of {trials} intervals contain the true mean '
      f'(nominal 950, exact 947)')
```

```
first interval (-0.299, 0.098) contains 0: True
951 of 1000 intervals contain the true mean (nominal 950, exact 947)
```

The first interval covers 0 in about 95% of runs, and the total hit count centers around 947 (typically within ± 14) rather than 950. That three-hit shortfall is not random noise. At $n = 100$, the exact t-quantile is $t_{99,0.975} = 1.984$ instead of 1.96, so the true coverage of the 1.96 interval is $P(|t_{99}| \le 1.96) = 0.947$. The ± 14 scatter around 947 is simply two binomial standard deviations, $2\sqrt{1000 \cdot 0.947 \cdot 0.053} \approx 14$. This same recipe gives the error bar on a mean accuracy across k training seeds: take the seed-to-seed standard deviation divided by $\sqrt{k}$, and multiply by the appropriate quantile.

36.5.3.3 Tests and Intervals

You may have noticed that the number 1.96 just appeared twice: once as the rejection threshold of the z-test, and again as the multiplier in the Gaussian confidence interval. That is no coincidence; tests and confidence intervals are two sides of the exact same coin.

Consider Gaussian data with known σ. A two-sided z-test of $H_0 : \mu = \mu_0$ fails to reject H_0 precisely when $|T| \le z_{1-\alpha/2}$, where $T = \sqrt{n}(\bar{x} - \mu_0)/\sigma$. Rearranging that inequality gives $|\bar{x} - \mu_0| \le h$ with $h = z_{1-\alpha/2}\,\sigma/\sqrt{n}$, which simply states that $\mu_0 \in [\bar{x} - h,\ \bar{x} + h]$. In short, *a 95% confidence interval is the set of null values that a 5% hypothesis test keeps*. Its coverage guarantee follows immediately from the test's type I error rate: for every μ, $P_\mu(\mu \in [\bar{x} - h, \bar{x} + h]) = 1 - P_\mu(\text{the test of } H_0 : \mu \text{ rejects}) = 1 - \alpha$.

What makes this duality so useful is that it works for *any* family of hypothesis tests. If for every candidate θ_0 we have a level-α test of $H_0 : \theta = \theta_0$, then collecting all values θ_0 that the test does not reject automatically forms a $1 - \alpha$ confidence set by the same one-line argument. Whenever an explicit formula fails or misbehaves, we can invert a test instead. For example, if a model answers 29 of 30 questions correctly, the Wald interval absurdly extends past an accuracy of 1, whereas inverting an exact binomial test produces a valid interval inside $[0, 1]$ (exercise 4).

This duality also suggests a better habit when comparing two models: report a confidence interval for the difference, such as $\bar{x}_B - \bar{x}_A \pm 1.96\,\sigma\sqrt{2/k}$ across k seeds. It excludes 0 if and only if the hypothesis test rejects, while also showing you how large the improvement might actually be.

36.5.3.4 The Delta Method

Often we care not about $\hat{\theta}$ itself, but about a smooth function $g(\hat{\theta})$ of it: the log-odds of an accuracy, the ratio of two losses, or a relative error reduction. How do we propagate our error bar through g? When g is differentiable at θ, a first-order Taylor expansion gives

$$g(\hat{\theta}) \approx g(\theta) + g'(\theta)\,(\hat{\theta} - \theta).$$

Locally around θ, the function g simply rescales small deviations by its slope $|g'(\theta)|$. Taking standard deviations and plugging in $\hat{\theta}$ for the unknown θ in g' yields the *delta method* (Wasserman 2013):

$$\operatorname{se}(g(\hat{\theta})) \approx \left|g'(\hat{\theta})\right| \operatorname{se}(\hat{\theta}). \tag{36.74}$$

Let us try it on a concrete number. An accuracy of $\hat{\theta} = 0.90$ on $n = 1{,}000$ questions has standard error $\operatorname{se}(\hat{\theta}) = \sqrt{0.9 \cdot 0.1/1000} \approx 0.0095$. Its log-odds $g(\hat{\theta}) = \log \frac{\hat{\theta}}{1-\hat{\theta}} \approx 2.20$ has derivative $g'(\hat{\theta}) = 1/(\hat{\theta}(1-\hat{\theta})) \approx 11.1$, so the standard error of the log-odds is roughly $11.1 \cdot 0.0095 \approx 0.105$.

The same linearization extends directly to a vector of estimates $\hat{\theta}$ with covariance matrix Σ, giving $\operatorname{Var}(g(\hat{\theta})) \approx \nabla g^{\top} \Sigma \, \nabla g$. Notice that the off-diagonal covariances in Σ matter: if you take the ratio of two losses measured on the same test set, the numerator and denominator fluctuate together.

Like any linear approximation, the delta method has clear limits. First, it requires $g'(\theta) \neq 0$: at a stationary point such as $g(\theta) = (\theta - \frac{1}{2})^2$ evaluated at $\theta = \frac{1}{2}$, the linear term vanishes and $g(\hat{\theta})$ fluctuates at the faster rate $1/n$ rather than $1/\sqrt{n}$. Second, $\hat{\theta}$ must stay inside a region where g is nearly linear, which breaks down for an accuracy very close to 100% (exercise 4). And when a statistic has no handy derivative at all, we need a completely different approach: resampling.

36.5.4 The Bootstrap

Every standard error we have computed so far came from an analytic formula. Yet many quantities we care about in practice, such as a sample median, an AUC, or a BLEU score, fluctuate from test set to test set just like accuracy does, without offering a tidy formula for their variance. The bootstrap solves this by simulating the sampling distribution directly from the data. Below, we first verify the method on a case whose answer we know, then apply it where no simple formula is available, and finally examine two instructive ways it can fail.

36.5.4.1 The Plug-in Principle

If we knew the true data-generating distribution F, finding the sampling distribution of *any* statistic would be effortless: just draw thousands of fresh data sets from F and recompute the statistic on each. Of course, we do not know F. The *bootstrap* (Efron 1979) makes one bold substitution: it replaces the unknown F with the *empirical distribution* $\hat{F}_n$, which places probability $1/n$ on each of the n observed data points. Drawing a new sample of size n from $\hat{F}_n$ is simply sampling n points from our data *with replacement*. The entire algorithm takes three steps:

1. Draw n points from the observed sample with replacement to create a *bootstrap resample* (some original points appear multiple times, others not at all).
2. Recompute the statistic $\hat{\theta}^{*}$ on that resample.
3. Repeat R times to collect replicates $\hat{\theta}^{*}_1, \dots, \hat{\theta}^{*}_R$.

The standard deviation of those R replicates estimates $\mathrm{se}(\hat{\theta})$, and their $\alpha/2$ and $1-\alpha/2$ quantiles form the *percentile confidence interval*. Figure 36.29 illustrates the workflow. Behind its simplicity lie two key assumptions: the observations are i.i.d., and the statistic varies smoothly under small perturbations of the data.

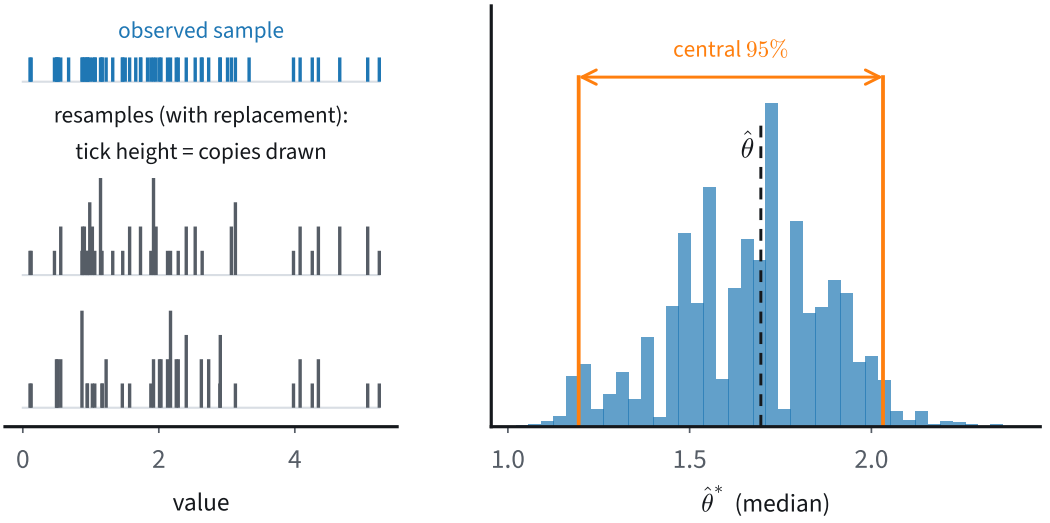


Figure 36.29: The ordinary bootstrap. Starting from one observed sample (top left), we draw many resamples of the same size with replacement (bottom left, where tick heights count how many times each observation was selected) and recompute the statistic, here the median. The histogram of replicates (right) approximates the sampling distribution around $\hat{\theta}$ (dashed line): its standard deviation estimates the standard error, and its central quantiles form a percentile interval. The histogram looks jagged because any resample's median must equal one of a few observed values; this mild lumpiness turns into outright failure for the maximum.

36.5.4.2 Bootstrap Standard Errors

Before trusting the bootstrap on a hard statistic, let us check it against a formula we already know. Our opening test set had 912 correct and 88 incorrect answers, with theoretical standard error 0.0090. Let us resample those 1,000 binary indicators 5,000 times and see if the bootstrap recovers the same number.

```
correct = np.r_[np.ones(912), np.zeros(88)]          # the opening's test set
idx = rng.integers(0, len(correct), (5000, len(correct)))  # 5,000 resamples
boot = correct[idx].mean(axis=1)                     # accuracy of each resample
print(f'bootstrap se {boot.std(ddof=1):.4f}, '
      f'formula {np.sqrt(0.912 * 0.088 / 1000):.4f}')
```

```
bootstrap se 0.0090, formula 0.0090
```

The two estimates match to within a few percent. Now let us turn to a statistic without a practical closed-form standard error: the median of 200 draws from the exponential distribution with mean 1, a right-skewed distribution whose true median $\log 2 \approx 0.69$ sits

below its mean. An asymptotic formula for the median's standard error does exist, namely $1/(2f(m)\sqrt{n})$, where $f(m)$ is the density evaluated at the median m (exercise 6), but in real life we do not know $f(m)$. In our synthetic experiment we *do* know it, $f(m) = e^{-\log 2} = \frac{1}{2}$, giving a theoretical benchmark of $1/\sqrt{200} \approx 0.071$ against which we can judge the bootstrap.

```
data = rng.exponential(1.0, 200)    # its median's se formula needs the unknown density
theta_hat = np.median(data)
idx = rng.integers(0, len(data), (10000, len(data)))  # resample with replacement
boot = np.median(data[idx], axis=1)                   # one median per resample
lo, hi = np.percentile(boot, [2.5, 97.5])
print(f'median {theta_hat:.3f}, bootstrap se {boot.std(ddof=1):.3f}, '
      f'95% percentile interval ({lo:.3f}, {hi:.3f})')
```

```
median 0.741, bootstrap se 0.090, 95% percentile interval (0.636, 0.993)
```

Without ever being told the density, the bootstrap typically lands near 0.07. Across different data sets, the sample median fluctuates around $\log 2$ (usually within 0.15), and the bootstrap standard error itself varies between roughly 0.05 and 0.11. Why so much variation? Because the median of any resample must equal one of the few original data points sitting near the middle of the sample, making the bootstrap distribution lumpy and sensitive to the random gaps between those points. Consequently, the percentile interval is somewhat coarse and occasionally barely covers $\log 2$. Even so, the median is smooth enough for the bootstrap to remain valid, since the underlying density at the median is strictly positive.

36.5.4.3 Failures of the Bootstrap

It is just as important to know when the bootstrap breaks. Both of its core assumptions, smoothness and independence, can fail in practice.

First, the bootstrap fails when a statistic depends too sharply on extreme data points. The classic counterexample is the sample maximum. Let $x_1, \dots, x_n$ be i.i.d. uniform on $[0, \theta]$, with estimator $\hat{\theta} = \max_i x_i$. Any bootstrap resample achieves the exact same maximum $\hat{\theta}$ unless all n draws happen to miss the single largest observation, which gives

$$P(\hat{\theta}^* = \hat{\theta}) = 1 - \left(1 - \frac{1}{n}\right)^n \;\to\; 1 - \frac{1}{e} \approx 0.63.$$

Thus, the bootstrap distribution piles 63% of its probability mass onto a single point, even though the true sampling distribution of $\hat{\theta}$ is continuous. Worse still, no resample can ever exceed $\hat{\theta}$, which is strictly below θ, so the percentile interval has zero chance of containing θ.

Second, the bootstrap fails when you resample units that are not independent, a common trap in modern NLP and vision benchmarks. Consider a reading-comprehension benchmark containing G passages with m questions per passage. If a model misunderstands a passage, it tends to miss several of its questions at once. Suppose correctness indicators for two questions on the same passage have correlation ρ, while different passages are independent. Then the overall accuracy has variance

$$\mathrm{Var}(\hat{\theta}) = \frac{\theta(1-\theta)}{Gm}(1 + (m-1)\rho),$$

which is inflated by a factor of $1 + (m-1)\rho$ compared with Gm independent questions. A naive bootstrap that resamples individual questions ignores this within-passage correlation and understates the standard error by $\sqrt{1 + (m-1)\rho}$. Even at $m = 20$ and a modest correlation of $\rho = 0.1$, that understates the error bar by a factor of $\sqrt{2.9} \approx 1.7$ (Miller 2024). The fix is straightforward: resample whole passages instead of individual questions (exercise 8).

We end this section with two important questions left open. First, we have studied how to evaluate and put error bars on a given estimator, but we have not yet asked how to *construct* a good estimator in the first place, or how tightly any estimator can pin down a parameter; Section 36.6 answers both questions through the likelihood function and its curvature. Second, every guarantee in this section assumed a single estimator or hypothesis test fixed *before* seeing the data. A model chosen because it achieved the highest score on a validation set does not satisfy that assumption: picking the winner from many candidates is multiple testing in disguise. In Section 36.10, we will ask how large a sample we need for our guarantees to hold simultaneously across an entire family of models.

36.5.5 Summary

Any number computed from random data is itself a random variable, and its sampling distribution across hypothetical alternative data sets determines how far we can trust it. Because mean squared error splits into squared bias plus variance, a biased estimator such as the shrunk mean can outperform the unbiased sample mean. Viewing an integral as an expectation turns the sample mean into Monte Carlo estimation, whose $\sigma/\sqrt{n}$ error rate never degrades with dimension, unlike deterministic grids. A hypothesis test rejects the null hypothesis when an observed difference would be rare by chance; since a p-value is uniform under the null, rejecting when $p \le \alpha$ controls the false-positive rate at α, while statistical power grows with the standardized effect times $\sqrt{n}$ and increases dramatically when competing models are paired on the same questions. A confidence interval is simply the set of parameter values that a hypothesis test fails to reject. Finally, when no analytic standard error exists, the bootstrap simulates the sampling distribution by resampling the data, working well for smooth statistics of independent observations while breaking down for extremes like the maximum and for clustered test sets.

36.5.6 Exercises

1. code **Shrinking the sample mean.** The text found that the shrunk mean $c\,\bar{x}$ of $x_1, \dots, x_n$ i.i.d. $\mathcal{N}(\theta, \sigma^2)$ has the smallest MSE at $c^\star = \theta^2/(\theta^2 + \sigma^2/n)$.
    1. Extend the text's simulation of the sample mean, the shrunk mean and the median to a grid of factors $c \in [0, 1]$ at $\theta = 1$, $\sigma = 4$ and $n = 30$. Plot the squared bias, the variance and the MSE against c with `d2l.plot`, and compare the empirical minimizer with $c^\star$.
    2. Shrink toward a point a instead of 0, with the estimator $a + c(\bar{x} - a)$. Compute its MSE and its optimal factor. Which a would you choose, and what does the answer say about the choice of 0 in the text?

3. $c^\star$ is not an estimator, because it contains θ. Plug in $\bar{x}$ for θ and s^2 for σ^2, giving $\hat{c} = \bar{x}^2/(\bar{x}^2 + s^2/n)$, and measure the MSE of $\hat{c}\,\bar{x}$ by simulation at $\theta \in \{0, 1, 3, 10\}$. Where does it beat $\bar{x}$, and where does it lose? Exercise 9 shows that in three or more dimensions a similar substitution wins for every θ.

Hint for (c): at $\theta = 0$ the plug-in factor is itself noisy; look at its distribution.

2. code **Multiple comparisons.** A hyperparameter sweep compares many configurations with one baseline, so it runs many tests at once.

   1. Twenty models, none of them better than the baseline, are each compared with it by an independent test at $\alpha = 0.05$. Compute the probability that at least one comparison is significant, and the same probability at the Bonferroni threshold $\alpha/20$. Show that the Bonferroni correction holds the probability of at least one false positive at or below α even when the tests are dependent.
   2. The Benjamini–Hochberg procedure (Benjamini and Hochberg 1995) sorts the m p-values, $p_{(1)} \le \cdots \le p_{(m)}$, finds the largest k with $p_{(k)} \le k\alpha/m$, and rejects the k hypotheses with the smallest p-values. Simulate a sweep of $m = 200$ configurations, of which $m_0 = 180$ have no effect. Draw the z-statistics from $\mathcal{N}(0, 1)$ for these and from $\mathcal{N}(3, 1)$ for the other 20, and compute the two-sided p-values `2 * norm.sf(np.abs(z))`. Over $1{,}000$ repetitions, report the average numbers of true and false discoveries and the average false-discovery proportion (the fraction of rejections that are false, or 0 without rejections) for the uncorrected threshold 0.05, for Bonferroni and for Benjamini–Hochberg. For independent tests, Benjamini–Hochberg guarantees an expected false-discovery proportion of at most $\alpha m_0/m$. Does the simulation agree?

3. code **Coverage at small n.** The interval Equation 36.73 uses the Gaussian quantile 1.96 in place of the t-quantile, and it assumes approximately Gaussian sample means.

   1. Run the coverage simulation of the text (1,000 intervals from standard normal data sets) with $n \in \{2, 5, 10, 30, 100\}$ and $100{,}000$ trials, and record the coverage of the interval. Then replace 1.96 by the 97.5% quantile of the t-distribution with $n-1$ degrees of freedom, which is 12.71, 2.776, 2.262, 2.045 and 1.984 for these n, and record the coverage again.
   2. Set $n = 2$ and replace 1.96 by 1.0, the 75% quantile of the t-distribution with one degree of freedom, so that the interval has 50% coverage. Compute the coverage separately for the intervals shorter and longer than the median width. Does the difference contradict Equation 36.71? Would you read a short interval as a sign of a precise estimate?
   3. Draw the data from the exponential distribution with mean 1 at $n = 10$, keeping the t-quantile 2.262. Measure the coverage, and count separately the intervals that lie entirely below and entirely above the true mean. Explain the asymmetry from the shape of the distribution.

4. code **Delta method near a boundary.** By Equation 36.74, an accuracy $\hat{\theta} = 0.90$ measured on $n = 1{,}000$ test examples has a log-odds $\log \frac{\hat{\theta}}{1-\hat{\theta}}$ with standard error about 0.105.

   1. Simulate $100{,}000$ test sets by drawing the number of correct predictions from the binomial distribution with $n = 1{,}000$ and $\theta = 0.9$. Compare the standard deviation of the log-odds with the delta-method value.

2. Repeat with $\theta = 0.999$. Estimate the fraction of test sets with $\hat{\theta} = 1$, compare it with its exact value, and state the log-odds of such a test set.
3. Which assumption behind the first-order expansion fails at $\theta = 0.999$? Compute the expected number of incorrect answers, $n(1-\theta)$, for both values of θ, and explain why this count, rather than n alone, decides whether the delta method and the Wald interval can be trusted.
4. A benchmark has 30 questions, and a model answers 29 of them correctly. Compute the Wald interval $\hat{\theta} \pm 1.96\sqrt{\hat{\theta}(1-\hat{\theta})/30}$ and note where it ends. Then invert a test: for each θ_0 on a fine grid of $[0.5, 1]$, compute the exact two-sided p-value of $H_0 : \theta = \theta_0$, $\min\{1, 2\min\{P_{\theta_0}(K \le 29),\ P_{\theta_0}(K \ge 29)\}\}$ with $K \sim \text{Binomial}(30, \theta_0)$, and collect the θ_0 that it keeps at $\alpha = 0.05$. Compare the two intervals, and their coverage at $\theta = 0.95$ over $100{,}000$ simulated benchmarks.

5. code **Paired comparisons.** ● Two models evaluated on the same test examples or the same seeds share part of their noise, and a paired test removes it.

   1. In the text the two models disagree on 41 questions, 25 to 16. Under H_0 the number of questions that only B gets right is $\text{Binomial}(41, \frac{1}{2})$. Compute the exact two-sided p-value and compare it with the Monte Carlo value of the sign-flip cell in Section 36.5.2.6. Then fix the net gap at 9 and vary the number of disagreements: compute p for 9 to 0, 17 to 8, 25 to 16 and 55 to 46. What decides the verdict, the gap or the number of disagreements?
   2. Model the per-seed accuracies with a seed effect that both models share: for seed s, the accuracies are $0.850 + u_s + e_{A,s}$ and $0.858 + u_s + e_{B,s}$, where $u_s \sim \mathcal{N}(0, 0.01^2)$ and $e_{A,s}, e_{B,s} \sim \mathcal{N}(0, 0.003^2)$ are independent. Implement the sign-flip test on the per-seed differences as an exact test: with k seeds, enumerate all 2^k sign patterns instead of drawing random ones. Over $1{,}000$ experiments with 10 seeds, compare its power with that of the unpaired permutation test `perm_p`.
   3. Repeat with 5 seeds. What is the smallest two-sided p-value that the exact sign-flip test can return, and what does it imply at $\alpha = 0.05$? A Monte Carlo version that draws 999 random sign patterns, as `perm_p` draws random shuffles, rejects in a few percent of these experiments. Why?

6. **Standard error of the median.** ● Let $x_1, \dots, x_n$ be i.i.d. with c.d.f. F, median m, and density $f(m) > 0$ at the median. The sample median $\hat{m}$ is at most $m + t$ exactly when at least half of the observations are at most $m + t$.

   1. The empirical c.d.f. $\hat{F}_n(m+t)$ has mean $F(m+t)$ and variance $F(m+t)(1 - F(m+t))/n$. Use the central limit theorem to show that $P(\hat{m} \le m + t) \approx \Phi(2\sqrt{n}\, f(m)\, t)$ for small t, and conclude that $\text{se}(\hat{m}) \approx 1/(2f(m)\sqrt{n})$. Relate the factor $1/f(m)$ to the slope g' in Equation 36.74.
   2. Evaluate the approximation for the exponential distribution with mean 1 at $n = 200$, and compare it with the bootstrap standard error that your run of the text's bootstrap of the median prints. Rerun the cell several times: the printed value moves between about 0.05 and 0.11. Why does the bootstrap estimate of a median's standard error vary this much from data set to data set, when the bootstrap estimate of a mean's standard error does not?
   3. Compute the ratio $\text{Var}(\hat{m})/\text{Var}(\bar{x})$ for large n for Gaussian data and for Laplace

data. Which estimator of the location would you choose for heavy-tailed data?

Hint for (b): a resample's median is one of a few observed values near $\hat{m}$, and the spacing of those values is random.

7. code **Grids against Monte Carlo.** ● Section 36.5.1.7 compares exponents: with N evaluations in d dimensions, a second-order grid has error of order $N^{-2/d}$ and Monte Carlo of order $N^{-1/2}$. Test this comparison on $f(\mathbf{x}) = \exp(-\|\mathbf{x}\|^2)$ over $[0,1]^d$, whose integral is $\left(\frac{\sqrt{\pi}}{2}\operatorname{erf}(1)\right)^d$ by Fubini's product rule (Section 34.6).

   1. Implement the midpoint rule on the full grid of m^d cubes and the Monte Carlo average Equation 36.67 over $N = m^d$ uniform samples. For $d \in \{1, 2, 4, 5, 6\}$ and N up to about 10^6, plot the relative error against N on log-log axes, using for Monte Carlo the root mean squared error over 20 repetitions. Do the slopes match the predicted exponents?
   2. At $d = 4$ the exponents tie. Which method is more accurate there? What sets the constant in front of each rate, and how does each constant change with d?
   3. Both errors can be predicted. The midpoint sum of f is a product of one-dimensional midpoint sums. The root mean squared error of Monte Carlo is $\operatorname{sd}(f(\mathbf{U}))/\sqrt{N}$ by Equation 36.68, for $\mathbf{U}$ uniform on $[0,1]^d$, and the product rule factors $\mathbb{E}[f(\mathbf{U})^2]$ into one-dimensional integrals. Compute both predicted relative errors for $d \in \{4, 5, 6\}$ at $N \approx 10^5$ and compare them with your measurements. At which d do the two methods break even? How reliably do 20 repetitions decide the comparison at that d?

8. code **Clustered test sets.** ● Evaluation sets often consist of clusters of related items, such as several questions about one passage, and the i.i.d. bootstrap then understates the standard error (Miller 2024). A test set has G clusters of m items each, its expected accuracy is θ, the correctness indicators of two items in the same cluster have correlation ρ, and items in different clusters are independent.

   1. Show that the average of m variables with common variance v and pairwise correlation ρ has variance $v(1 + (m-1)\rho)/m$, and conclude that the accuracy $\hat{\theta}$ on the test set has variance
   $$\operatorname{Var}(\hat{\theta}) = \frac{\theta(1-\theta)}{Gm}(1 + (m-1)\rho). \tag{36.75}$$
   2. Simulate $G = 100$ and $m = 20$, drawing a cluster accuracy $\theta_g \sim \operatorname{Beta}(6.3, 2.7)$ for each cluster and conditionally independent items given θ_g. These choices give $\theta = 0.7$ and $\rho = 1/(6.3 + 2.7 + 1) = 0.1$. Over 300 test sets, compare the standard deviation of $\hat{\theta}$ with the average i.i.d. bootstrap standard error, which resamples items as in the text's bootstrap of the accuracy, and with the average cluster bootstrap standard error, which resamples whole clusters, using $R = 1{,}000$ resamples for each. By which factor does the i.i.d. bootstrap understate the standard error? Compare the factor with the prediction of (a).

9. code **Stein's paradox.** ●● Observe $\mathbf{x} \sim \mathcal{N}(\theta, \mathbf{I}_d)$ and measure an estimator by its risk $\mathbb{E}\|\hat{\theta} - \theta\|^2$, the MSE with squares replaced by squared norms. The estimator $\mathbf{x}$ has risk d for every θ. James and Stein (1961) showed that for $d \geq 3$ the estimator
$$\hat{\theta}_{\text{JS}} = \left(1 - \frac{d-2}{\|\mathbf{x}\|^2}\right)\mathbf{x} \tag{36.76}$$

has smaller risk for every θ.

1. Prove Stein's lemma (Stein 1981): if $z \sim \mathcal{N}(\theta, 1)$ and h is differentiable with $\mathbb{E}|h'(z)| < \infty$, then $\mathbb{E}[(z-\theta)h(z)] = \mathbb{E}[h'(z)]$. Integrate by parts.
2. For an estimator $\mathbf{x} + \mathbf{g}(\mathbf{x})$ with $\mathbf{g}$ weakly differentiable (here: differentiable away from the origin, with $\mathbb{E}\|\mathbf{g}(\mathbf{x})\|^2 < \infty$), show that its risk is

$$d + \mathbb{E}\|\mathbf{g}(\mathbf{x})\|^2 + 2\sum_{i=1}^{d} \mathbb{E}\left[\frac{\partial g_i}{\partial x_i}(\mathbf{x})\right]. \tag{36.77}$$

   Use this identity to show that $\widehat{\theta}_{\mathrm{JS}}$ has risk $d - (d-2)^2\,\mathbb{E}\big[1/\|\mathbf{x}\|^2\big]$. Where does the argument need $d \geq 3$?
3. Simulate the risks of $\mathbf{x}$, of $\widehat{\theta}_{\mathrm{JS}}$, and of the positive-part estimator $\max\big(0, 1 - (d-2)/\|\mathbf{x}\|^2\big)\mathbf{x}$ for $d = 10$ and $\|\theta\| \in \{0, 1, 2, 5, 10, 20\}$, and plot them against $\|\theta\|$.
4. Write the optimal factor $c^\star$ for $c\,\mathbf{x}$ in d dimensions, the analogue of the text's $c^\star$. Which function of the data replaces the unknown quantity in $\widehat{\theta}_{\mathrm{JS}}$, and why does this substitute become accurate as d grows? Compare with exercise 1c.

10. [extended] **Peeking at a running test.** ●● An A/B test, or an evaluation during training, is often watched while the data arrive and stopped as soon as the result looks significant (Johari et al. 2017). Let $x_1, x_2, \dots$ be i.i.d. $\mathcal{N}(\mu, 1)$, let $H_0 : \mu = 0$, and let $S_n = \sum_{i \leq n} x_i$. The z-test at time n rejects when $|S_n|/\sqrt{n} \geq 1.96$.

    1. Simulate $4{,}000$ runs of length $N = 10{,}000$ under H_0. Estimate the probability that at least one of K equally spaced looks rejects, for $K \in \{1, 2, 5, 10, 100, 1000, 10000\}$. The law of the iterated logarithm implies that this probability tends to one as monitoring continues indefinitely. How fast does it grow in your table?
    2. Put the prior $\mathcal{N}(0, \tau^2)$ on μ and define the mixture likelihood ratio

$$\Lambda_n = \int \prod_{i=1}^{n} \frac{p(x_i \mid \mu)}{p(x_i \mid 0)}\, \mathcal{N}(\mu; 0, \tau^2)\, d\mu. \tag{36.78}$$

       Show that $\Lambda_n = (1 + n\tau^2)^{-1/2} \exp\!\big(\tau^2 S_n^2 / (2(1 + n\tau^2))\big)$, and that under H_0 it is a martingale: $\mathbb{E}[\Lambda_{n+1} \mid x_1, \dots, x_n] = \Lambda_n$.
    3. Ville's inequality, which we take on faith, states that a nonnegative martingale with $\Lambda_0 = 1$ satisfies $P(\sup_n \Lambda_n \geq 1/\alpha) \leq \alpha$. Conclude that rejecting at the first n with $\Lambda_n \geq 1/\alpha$ has type I error at most α, however often the data are inspected. Derive the rejection boundary for $|S_n|$, and compare its growth in n with that of the fixed boundary $1.96\sqrt{n}$.
    4. Check the type I error of the sequential test by simulation for $\tau^2 \in \{0.01, 0.1, 1\}$. Then set $\mu = 0.2$, and take the fixed-sample z-test whose n follows from Equation 36.69 for power 0.8 as the reference. For each of the three values of τ^2, report the power of the sequential test by that n, its median stopping time, and its power by N. How does the choice of τ^2 affect these numbers? What does a test that is valid at every look give up, and what does it gain?

36.6 Maximum Likelihood

Suppose we flip a coin thirteen times and see nine heads and four tails. Anyone would estimate the probability of heads as $9/13 \approx 0.69$. Yet two natural questions follow immediately: why that particular number, and how much should we trust it? Could the coin still be fair?

One principle answers both questions: choose the parameters under which the observed data are most probable. This is the principle of *maximum likelihood*, and it underlies nearly every training loss in this book. For the coin, it returns $9/13$. For a neural network, it turns an assumption about noise in the targets into the loss we minimize: squared error and cross-entropy, which earlier chapters adopted because they work well, both arise in this way. Remarkably, the same loss also tells us how tightly the data pin down the parameters. The more sharply the loss curves around its minimum, the smaller the error bar, and for our coin that curvature will tell us whether thirteen flips are enough to rule out a fair coin.

Earlier, Section 36.5 measured the error of estimators given in closed form, such as a sample mean or a sample proportion. Here the estimator is defined by an optimization problem, and its error bar comes directly out of the loss itself. We first show how the standard losses of earlier chapters follow from maximum likelihood. We then turn to the heart of the section: how accurate is the maximum-likelihood estimate? Its main result, that the estimation error becomes Gaussian as the sample grows with a variance set by the curvature of the loss, is the one to remember. Finally, seeing three heads in a row shows what goes wrong when data are scarce, and how a prior belief held before seeing the data turns into a regularizer. Models in which part of every observation is hidden follow in Section 36.7.

36.6.1 The Maximum Likelihood Principle

A probabilistic model assigns a probability to every data set it could have generated. Once the data are in hand, we turn this viewpoint around: hold the observed data fixed and ask how probable they are under each candidate setting of the parameters.

Let θ denote the parameters of a model and X the observed data set. The **likelihood** is the probability of the observed data (or, for continuous data, its density) viewed as a function of the parameters, $P(X \mid \theta)$. The **maximum likelihood estimate** (MLE) is the parameter setting that maximizes this function,

$$\hat{\theta} = \operatorname*{argmax}_{\theta} P(X \mid \theta). \tag{36.79}$$

Among all candidate parameters, the MLE is the one under which what we actually observed was most probable. Notice that this rule treats θ as an unknown constant rather than a random variable: it assigns probabilities to data sets, never to parameter values. That distinction will matter when we introduce priors in Section 36.6.4.

36.6.1.1 The Coin

A biased coin is the simplest model on which we can work the principle all the way through. Its single parameter is the probability θ of heads. In Section 36.3 we called this number p; now that it is an unknown to be estimated, we write θ. Because the flips are independent, their probabilities multiply, so a specific sequence with n_H heads and n_T tails has probability

$$P(X \mid \theta) = \theta^{n_H}(1-\theta)^{n_T}. \tag{36.80}$$

Viewed as a function of θ, this expression is the likelihood. For our thirteen-flip sequence HHHTHTTHHHHHT, with $n_H = 9$ heads and $n_T = 4$ tails, it is $\theta^9(1-\theta)^4$. A fair coin assigns this sequence probability $2^{-13} \approx 1.2 \times 10^{-4}$, whereas a coin with $\theta = 9/13$ assigns it 3.3×10^{-4}, about 2.7 times as much.

To locate the maximum, we take the logarithm first. Because the logarithm is strictly increasing, it leaves the location of the maximum unchanged while turning the product into a sum that is effortless to differentiate. Setting the derivative to zero gives

$$\frac{d}{d\theta}\Big[9\log\theta + 4\log(1-\theta)\Big] = \frac{9}{\theta} - \frac{4}{1-\theta} = 0 \quad\Longrightarrow\quad 9(1-\theta) = 4\theta \quad\Longrightarrow\quad \hat{\theta} = \frac{9}{13}.$$

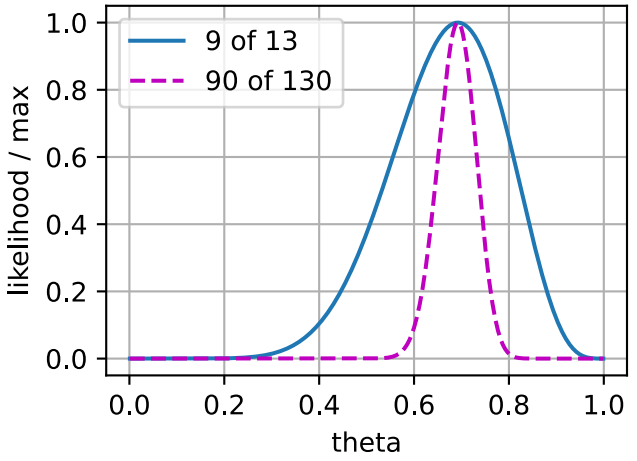
The endpoints $\theta = 0$ and $\theta = 1$ assign probability zero to any sequence containing both heads and tails, so the maximum must lie in the interior $(0,1)$, and this stationary point is indeed the global maximizer. Repeating the same calculation for the general likelihood Equation 36.80 yields

$$\frac{n_H}{\theta} - \frac{n_T}{1-\theta} = 0 \quad\Longrightarrow\quad \hat{\theta} = \frac{n_H}{n_H + n_T}.$$

The MLE of a coin's bias is simply the observed frequency of heads, just as intuition demands. Look closely at the derivative, however: encoding each flip as $x \in \{0,1\}$ (with $x = 1$ for heads and $x = 0$ for tails), the derivative is a sum of one term per flip, $x/\theta - (1-x)/(1-\theta)$. Each term is the gradient of a single observation's log-probability, and it will take center stage when we measure the accuracy of $\hat{\theta}$.

Now suppose the same frequency had come from ten times as many flips, 90 heads in 130. The point estimate cannot change, since it depends only on the ratio. What changes is how sharply the data single it out. Below, we plot both likelihoods, dividing each by its maximum so that the two curves share a single vertical axis.

```
theta = np.linspace(0.001, 0.999, 999)

def likelihood(theta, n_H, n_T):  # divided by its maximum
    lik = theta**n_H * (1 - theta)**n_T
    return lik / lik.max()

d2l.plot(theta, [likelihood(theta, 9, 4), likelihood(theta, 90, 40)], 'theta',
         'likelihood / max', legend=['9 of 13', '90 of 130'])
```



Both curves peak at $9/13 \approx 0.69$, but the second is about three times narrower. Later in this section we will turn the width of this peak into a precise error bar.

36.6.1.2 The Negative Log-Likelihood

The coin gave us a closed-form answer. A model with millions of parameters has none, so we must optimize numerically, and in floating-point arithmetic the raw likelihood is unusable. With n independent data points, it is a product of n probabilities,

$$P(X \mid \theta) = \prod_{i=1}^{n} p(x_i \mid \theta),$$

each lying in $[0, 1]$. A billion factors near $1/2$ multiply to roughly $(1/2)^{10^9}$, which underflows to zero in every floating-point format long before we can compute a gradient.

Taking the logarithm turns that product into a sum that stays comfortably in range:

$$\log\big((1/2)^{10^9}\big) = 10^9 \cdot \log(1/2) \approx -6.93 \times 10^8.$$

Maximizing the likelihood is therefore equivalent to maximizing the *log-likelihood* $\log P(X \mid \theta)$. Because optimization routines are written to minimize losses rather than maximize objectives, we flip the sign and define the **negative log-likelihood** (NLL):

$$\ell(\theta) = -\log P(X \mid \theta) = -\sum_{i=1}^{n} \log p(x_i \mid \theta). \tag{36.81}$$

The NLL is a sum of one term per observation, measuring the surprise of the model at each data point. For the coin it reads $\ell(\theta) = -\big(n_H \log\theta + n_T \log(1-\theta)\big)$, and nothing stops us from minimizing it by gradient descent as if no closed form existed. Below, we run gradient descent on nearly nine million flips and compare the result with $n_H/(n_H + n_T)$.

```
n_H = 8675309
n_T = 256245
theta = torch.tensor(0.5, requires_grad=True)
lr = 1e-9
for _ in range(100):
    loss = -(n_H * torch.log(theta) + n_T * torch.log(1 - theta))
    loss.backward()
    with torch.no_grad():
        theta -= lr * theta.grad
    theta.grad.zero_()
theta, n_H / (n_H + n_T)
```

```
(tensor(0.9713, requires_grad=True), 0.9713101437890875)
```

Gradient descent lands on 0.9713, matching the observed frequency to all printed digits. The learning rate 10^{-9} looks tiny, and summing the loss forces it on us: the gradient of a loss summed over nearly nine million flips carries a factor of n, so the largest stable step size shrinks in proportion (Section 35.1.2). This is why minibatch losses throughout this book average over examples rather than summing them.

Taking the logarithm also gives the gradient a clean additive form,

$$\nabla_\theta \, \ell(\theta) = -\sum_{i=1}^{n} \frac{\nabla_\theta \, p(x_i \mid \theta)}{p(x_i \mid \theta)},$$

with one independent term per example, whereas applying the product rule directly to the likelihood would produce n products of $n - 1$ factors. As a result, a random minibatch of examples yields an unbiased estimate of $\nabla \ell$, which is what makes stochastic gradient descent possible.

36.6.2 Losses as Negative Log-Likelihoods

For the coin, we derived the NLL directly from a probabilistic model. In earlier chapters, by contrast, we wrote down training losses such as squared error and cross-entropy directly and used them because they worked. Most of those standard losses turn out to be negative log-likelihoods in disguise, which means that choosing a loss function amounts to choosing a probabilistic model of the data.

36.6.2.1 Cross-Entropy and the Entropy Floor

A classifier's training loss is an average NLL, and its training curve often levels off at a positive value rather than reaching zero. To understand what that value represents and how low it can possibly go, let us rewrite the average NLL as a sum over distinct outcomes rather than over individual examples. Let $\hat{p}_{\text{data}}$ be the *empirical distribution* of the sample: an outcome x that appears n_x times among the n observations receives probability $\hat{p}_{\text{data}}(x) = n_x/n$, its observed frequency. For our coin sequence, $\hat{p}_{\text{data}}(\text{H}) = 9/13$ and $\hat{p}_{\text{data}}(\text{T}) = 4/13$.

We now regroup the sum over data points into a sum over *distinct outcomes*, pooling all observations with $x_i = x$. On the coin, the nine heads pool into nine copies of $\log p_\theta(\text{H})$, so heads enters the average with weight $9/13$ and tails with weight $4/13$. In general,

$$\frac{1}{n}\,\ell(\theta) = -\frac{1}{n}\sum_{i=1}^{n}\log p_\theta(x_i) = -\sum_x \frac{n_x}{n}\,\log p_\theta(x)$$
$$= -\sum_x \hat{p}_{\text{data}}(x)\,\log p_\theta(x) = \text{CE}(\hat{p}_{\text{data}},\, p_\theta)\,.$$

The right-hand side is the *cross-entropy* from the empirical distribution to the model, $\text{CE}(p,q) = -\sum_x p(x)\log q(x)$. Because scaling by the positive constant $1/n$ does not move the minimizer, maximum likelihood for discrete outcomes is equivalent to minimizing cross-entropy. One line of regrouping explains why a classification loss carries that name.

How low can this cross-entropy go? It splits cleanly into a piece fixed by the data and a piece controlled by the model. Recall from Section 36.4.2 that the entropy of a distribution p on a finite set is $H(p) = -\sum_x p(x)\log p(x)$, its average surprise. The Kullback–Leibler (KL) divergence $D_{\text{KL}}(p\,\|\,q) = \sum_x p(x)\log\frac{p(x)}{q(x)}$ measures the average extra surprise of using q when p holds (Section 37.1 develops both in full). Throughout, we use natural logarithms, so all three quantities are measured in *nats*. Adding and subtracting $\sum_x p(x)\log p(x)$ gives

$$\text{CE}(p,q) = -\sum_x p(x)\log q(x) = -\sum_x p(x)\log p(x) + \sum_x p(x)\log\frac{p(x)}{q(x)} = H(p) + D_{\text{KL}}(p\,\|\,q).$$

Notice that the entropy depends on p alone; only the KL divergence depends on the model q. By Gibbs' inequality (Section 36.4.2), the KL divergence is never negative and vanishes only when $q = p$.

Setting $p = \hat{p}_{\text{data}}$ and $q = p_\theta$, we see that the training loss can never drop below the entropy of the observed frequencies, and it touches that floor only if the model family can reproduce those frequencies exactly. All that training can reduce is the KL divergence from the empirical frequencies to the model. For our coin, the floor is $H(9/13) = -\frac{9}{13}\log\frac{9}{13} - \frac{4}{13}\log\frac{4}{13} = 0.617$ nats. A fair-coin model has cross-entropy $\log 2 = 0.693$ nats, leaving a KL gap of 0.076 nats; a model with $\theta = 0.9$ leaves a gap of 0.164 nats. Figure 36.30 plots the entire curve.

The same identity applies to a classifier. A classifier models $p_\theta(y \mid \mathbf{x})$, and its average NLL averages $-\log p_\theta(y_i \mid \mathbf{x}_i)$ over the training set. Grouping examples by their input recovers the cross-entropy decomposition at each distinct input, with the empirical label frequencies at that input playing the role of $\hat{p}_{\text{data}}$ (Exercise 1). For one-hot labels and a softmax output, this average NLL is precisely the softmax cross-entropy loss of Section 3.1. In a typical data set, every training input appears only once, so the empirical label distribution at each input is one-hot and the training-loss floor is zero regardless of how noisy the labels are. The irreducible floor that practitioners encounter appears on held-out data, where the expected loss cannot fall below the conditional entropy of the labels given the inputs (Section 37.1.3.1).

A language model is the same classifier with a very large set of labels: the input is the text so far, the label is the next token, and training minimizes the average NLL per token. It is usually reported through its exponential, the *perplexity* (Section 8.3.2), which is the number of equally likely tokens that would cause the same average surprise. Here the held-out floor is far from zero. Most sentences can continue in many reasonable ways, so the conditional

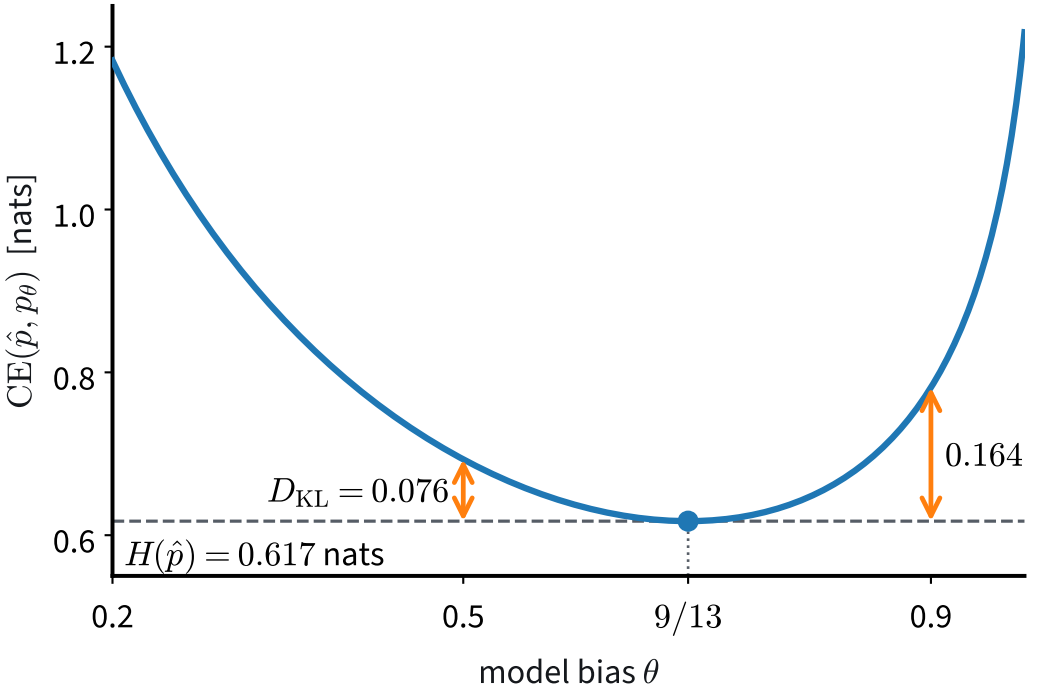


Figure 36.30: Per-flip cross-entropy for the coin data (9 heads and 4 tails) as a function of the model bias θ. The loss never falls below the entropy of the observed frequencies, 0.617 nats (dashed line), touches that floor at the MLE $9/13$, and exceeds it at any other θ by the KL divergence from the data frequencies to the model (orange): 0.076 nats at $\theta = 1/2$ and 0.164 nats at $\theta = 0.9$.

entropy of the next token is large, and no model, however big, can push its held-out loss below it.

36.6.2.2 Densities

For continuous data, defining the likelihood takes one extra step. The probability of observing any single exact real number is zero, so taken literally the likelihood would be 0 for every θ and could not distinguish one parameter setting from another. Instead, we ask for each observation to match only within a small tolerance ϵ, and then note that ϵ drops out of the optimization.

For i.i.d. observations $x_1, \dots, x_n$ with density $p(x \mid \theta)$, the probability that every X_i falls in an interval of width ϵ around x_i is, to first order in ϵ,

$$P\big(X_i \in [x_i, x_i + \epsilon] \ \forall i \mid \theta\big) \approx \epsilon^n \prod_{i=1}^{n} p(x_i \mid \theta).$$

Taking the negative logarithm gives

$$-\log P\big(X_i \in [x_i, x_i + \epsilon] \ \forall i \mid \theta\big) \approx -n \log \epsilon - \sum_i \log p(x_i \mid \theta).$$

The tolerance ϵ enters only through the additive constant $-n \log \epsilon$, which does not depend on θ. Requiring four digits of precision or four hundred shifts that constant without moving the minimizer, so we drop it and minimize $-\sum_i \log p(x_i \mid \theta)$, just as in the discrete case. Keep in mind that a density value is not itself a probability: only the product $\epsilon\, p(x_i \mid \theta)$ approximates a probability, so p carries the inverse units of x and can exceed 1.

The population form of the cross-entropy identity carries over directly to continuous data. The average negative log-density on the sample remains our objective, and its expectation under the true data-generating density $p_\star$ is

$$\mathbb{E}_{x \sim p_\star}\big[-\log p(x \mid \theta)\big] = h(p_\star) + D_{\mathrm{KL}}(p_\star \,\|\, p_\theta),$$
$$h(p_\star) = -\int p_\star(x) \log p_\star(x)\, dx,$$

by the same add-and-subtract argument with integrals replacing sums. Because the *differential entropy* $h(p_\star)$ does not depend on θ, maximum likelihood at the population level still minimizes a KL divergence. The finite-sample identity of the discrete case has no continuous counterpart, since the empirical distribution of n distinct real numbers is not a density.

36.6.2.3 Gaussian Noise and Squared Error

The standard regression loss emerges from the same principle as soon as we specify a model for noise in the targets. Section 2.1.3 derived this connection for linear regression, and nothing in that derivation relies on the model being linear. Suppose each target equals the model's prediction plus Gaussian noise of *fixed* variance, $y_i \sim \mathcal{N}(\widehat{y}_i, \sigma^2)$ with $\widehat{y}_i = f_\theta(\mathbf{x}_i)$. Summing the negative logarithm of the Gaussian density $(2\pi\sigma^2)^{-1/2} \exp(-(y-\widehat{y})^2/(2\sigma^2))$ over n independent examples gives

$$-\log \prod_{i=1}^{n} \mathcal{N}(y_i;\, \hat{y}_i,\, \sigma^2) = \frac{1}{2\sigma^2} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 + \frac{n}{2} \log(2\pi\sigma^2). \tag{36.82}$$

Because the variance σ^2 is fixed, the second term is constant in θ and the leading factor $1/(2\sigma^2)$ is a positive constant. Minimizing this NLL is therefore equivalent to minimizing $\sum_i (y_i - \hat{y}_i)^2$: squared-error regression is maximum likelihood under the assumption of Gaussian residuals. The same recipe connects many familiar losses, including the four listed below, to the NLL of a conditional distribution $p(y \mid \mathbf{x})$. Each row follows by taking the negative logarithm of the distribution and dropping constants that do not depend on θ.

Noise model $p(y \mid \mathbf{x})$	Negative log-likelihood $-\log p(y \mid \mathbf{x})$	Deep-learning loss
Gaussian, fixed variance σ^2	$\frac{1}{2\sigma^2}(y - \hat{y})^2 + \text{const}$	mean squared error (MSE)
Bernoulli	$-y \log \hat{p} - (1-y) \log(1-\hat{p})$	binary cross-entropy (BCE)
Categorical (one-hot y)	$-\sum_k y_k \log \hat{p}_k$	softmax cross-entropy
Laplace, fixed scale s	$\frac{1}{s}\lvert y - \hat{y}\rvert + \text{const}$	mean absolute error (MAE)

For all of these losses, choosing a loss function is choosing a noise model. Not every useful loss arises this way, however: the hinge loss of support vector machines has no normalizing constant independent of the model's output, so it is not the NLL of any conditional distribution, even up to an additive constant (Exercise 3).

The choice of noise model matters the moment an outlier enters the data. Suppose we fit a single constant c to five sensor readings $\{1, 2, 3, 4, 100\}$, where the last reading comes from a faulty sensor. Under Gaussian noise we minimize $\frac{1}{2}\sum_i (y_i - c)^2$, whose derivative $-\sum_i (y_i - c)$ vanishes at the sample mean, $c = 22$. Under Laplace noise we minimize $\sum_i \lvert y_i - c\rvert$, a sum of V-shaped curves whose slope at c equals the number of readings below c minus the number above. That slope is negative to the left of the median and positive to the right, so the minimum sits at the median, $c = 3$. Had the fifth reading been 5, both noise models would have returned 3. As Figure 36.31 illustrates, the faulty reading drags the Gaussian fit all the way to 22 while leaving the Laplace fit untouched.

36.6.3 Accuracy of the Maximum Likelihood Estimate

We can now return to our opening question: how sure should we be about $9/13$? For a sample proportion, Section 36.5 gave the standard error $\sqrt{\hat{\theta}(1-\hat{\theta})/n}$, which estimates the standard deviation of $\hat{\theta}$ across repeated samples. In a general model, however, the MLE is not a simple sample average, so we need a general way to read an error bar directly off the loss. Two ingredients do the job: the per-example gradient we met for the coin, and the curvature of the NLL.

36.6.3.1 The Score and Fisher Information

The **score** of a single observation is the gradient of its log-probability with respect to the parameters,

$$s(x; \theta) = \nabla_\theta \log p(x \mid \theta).$$

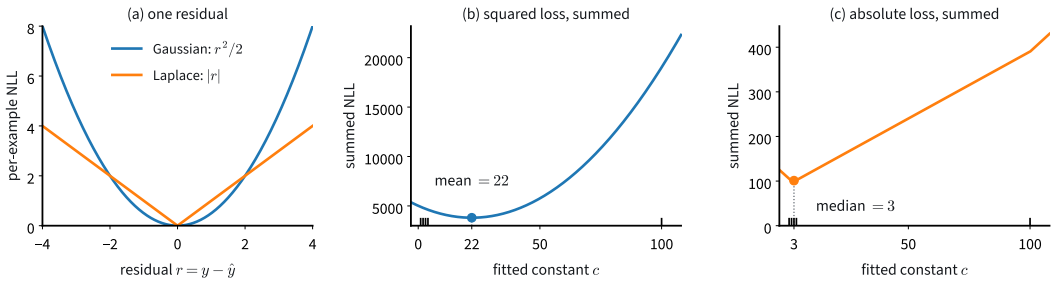


Figure 36.31: Fitting a single constant c to the readings $1, 2, 3, 4, 100$. (a) Per-example negative log-likelihood as a function of the residual: quadratic for Gaussian noise, V-shaped for Laplace noise. (b) The summed squared loss forms a parabola whose minimum, the mean 22, is pulled far toward the outlier. (c) The summed absolute loss has a kink at every reading (black ticks) and reaches its minimum at the median 3; every point pulls on c with one unit of slope, no matter how far away it lies.

This generalizes the coin's per-flip gradient to any probabilistic model, and the equation $\sum_i s(x_i; \hat{\theta}) = \mathbf{0}$ is the first-order optimality condition that locates the MLE.

A remarkable property of the score is that, at the true parameter, its expected value is zero. To see why, differentiate the normalization condition $\int p(x \mid \theta)\, dx = 1$ under the integral sign:

$$\begin{aligned} \mathbf{0} &= \nabla_\theta \int p(x \mid \theta)\, dx = \int \nabla_\theta\, p(x \mid \theta)\, dx \\ &= \int s(x;\theta)\, p(x \mid \theta)\, dx = \mathbb{E}_\theta\big[\, s(x;\theta)\,\big], \end{aligned}$$

where the third equality uses the log-derivative identity $\nabla_\theta\, p = p\, \nabla_\theta \log p = p\, s$. In words, averaged over data drawn from the model, the gradient of the log-likelihood vanishes at the parameter that generated the data. Because the score has mean zero, its covariance matrix equals its second moment. That covariance is the **Fisher information** (Fisher 1925),

$$\begin{aligned} I(\theta) &= \mathrm{Var}_\theta\big[\, s(x;\theta)\,\big] = \mathbb{E}_\theta\big[\, s(x;\theta)\, s(x;\theta)^\top\,\big] \\ &= -\,\mathbb{E}_\theta\big[\nabla_\theta^2 \log p(x \mid \theta)\big]. \end{aligned} \tag{36.83}$$

The middle equality reflects the mean-zero property. The last equality is the *information identity*: the variance of the score equals the expected curvature of the NLL. It follows by differentiating $\mathbb{E}_\theta[s] = \mathbf{0}$ once more, again moving the derivative inside the integral. The product rule splits the derivative into two terms,

$$\begin{aligned} \mathbf{0} &= \nabla_\theta \int s(x;\theta)\, p(x \mid \theta)\, dx \\ &= \int \nabla_\theta^2 \log p(x \mid \theta)\; p(x \mid \theta)\, dx + \int s(x;\theta)\, \big(\nabla_\theta\, p(x \mid \theta)\big)^\top dx. \end{aligned}$$

Substituting $\nabla_\theta\, p = p\, s$ in the second integral turns it into $\mathbb{E}_\theta[\, s\, s^\top]$, leaving

$$\mathbb{E}_\theta\big[\nabla_\theta^2 \log p(x \mid \theta)\big] + \mathbb{E}_\theta\big[\, s\, s^\top \,\big] = \mathbf{0},$$

which rearranges directly to the last equality of Equation 36.83. Intuitively, a sharply curved NLL pins the parameter down tightly, whereas a flat one leaves it loosely determined. The Fisher information measures the curvature of a single observation's NLL, averaged over data drawn from the model.

Both derivations moved a derivative inside an integral without comment. The results that follow rest on a short list of *regularity conditions*, which we collect once here:

- the density is twice continuously differentiable in θ;
- its support, the set of x with $p(x \mid \theta) > 0$, does not depend on θ;
- differentiation and integration may be interchanged (the tool of Section 34.7), which justifies the second equality in each derivation above and is the step a moving support would break;
- the true parameter lies in the interior of the parameter space;
- the Fisher information is finite and nonsingular.

Exercises 6 and 7 show concretely what breaks when the Fisher information is singular or when the support moves with the parameter.

Let us compute $I(\theta)$ on three models of increasing richness. First, for the coin, a single flip $x \in \{0, 1\}$ has log-likelihood $\log p(x \mid \theta) = x \log \theta + (1 - x) \log(1 - \theta)$, so its score and second derivative are

$$s(x; \theta) = \frac{x}{\theta} - \frac{1 - x}{1 - \theta}, \qquad \frac{d}{d\theta}\, s(x; \theta) = -\frac{x}{\theta^2} - \frac{1 - x}{(1 - \theta)^2}.$$

Taking $-\mathbb{E}[\,\cdot\,]$ with $\mathbb{E}[x] = \theta$ gives the per-flip Fisher information and its inverse divided by n,

$$I(\theta) = \frac{1}{\theta} + \frac{1}{1 - \theta} = \frac{1}{\theta(1 - \theta)}, \qquad \frac{I(\theta)^{-1}}{n} = \frac{\theta(1 - \theta)}{n}. \tag{36.84}$$

Notice that the per-flip information is smallest at $\theta = 1/2$: a fair coin is the hardest to pin down, because each flip carries the least information about the bias.

Next, a two-parameter model illustrates the matrix form. For a Gaussian with mean μ and standard deviation σ, the NLL of one observation is $\log \sigma + (x - \mu)^2/(2\sigma^2)$ up to a constant. Its second derivative with respect to μ is $1/\sigma^2$, while its second derivative with respect to σ is $-1/\sigma^2 + 3(x - \mu)^2/\sigma^4$, which has expectation $2/\sigma^2$. The mixed derivative $2(x - \mu)/\sigma^3$ has expectation zero, so the off-diagonal entries vanish and $I(\mu, \sigma) = \operatorname{diag}(1/\sigma^2,\, 2/\sigma^2)$. The NLL curves twice as sharply in σ as in μ, meaning the data pin down the scale more tightly than the mean. Figure 36.32 shows this bowl for 30 draws.

For both the coin and the Gaussian, the MLE is a sample average, so we could have worked out its spread without Fisher information. A model whose MLE is not an average shows what the theory adds. The *Cauchy location model* has density

$$p(x \mid \theta) = \frac{1}{\pi\big(1 + (x - \theta)^2\big)},$$

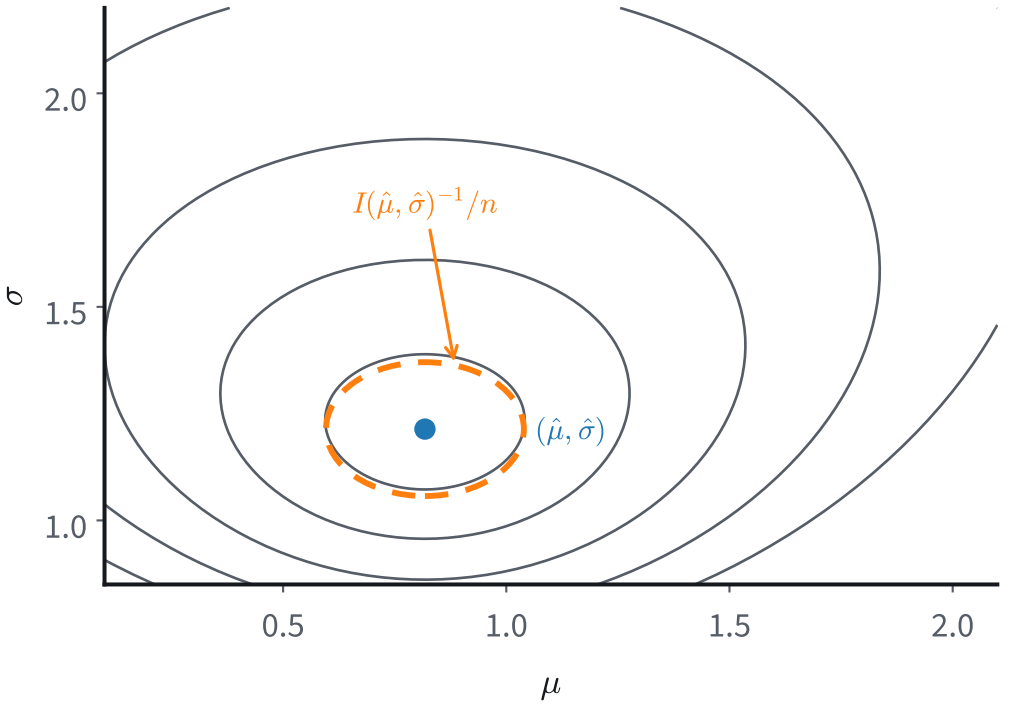


Figure 36.32: Negative log-likelihood of a Gaussian's parameters (μ, σ) for $n = 30$ draws from $\mathcal{N}(1, 1.5^2)$, with the MLE marked by the dot. The dashed ellipse shows $I(\hat{\mu}, \hat{\sigma})^{-1}/n$: it closely traces the innermost contour, where the NLL sits one half above its minimum, and it is wider along μ than along σ.

a symmetric bell centered at θ whose tails are so heavy that, as Section 36.1 showed, its mean does not exist. Writing $u = x - \theta$, the score is $s = 2u/(1+u^2)$ and its derivative is $\partial_\theta s = -2(1-u^2)/(1+u^2)^2$. We can find the Fisher information at sight by a trigonometric substitution: let $u = \tan\varphi$. Since $d\varphi = du/(1+u^2)$, the angle φ of a standard Cauchy variable is uniform on $(-\pi/2, \pi/2)$, and

$$s = \frac{2\tan\varphi}{1+\tan^2\varphi} = \sin 2\varphi, \qquad \text{so} \qquad I = \mathbb{E}\big[\sin^2 2\varphi\big] = \frac{1}{2}.$$

Each Cauchy observation therefore carries half a unit of information about its center, the same as a Gaussian observation with $\sigma^2 = 2$. Notice also that the score never exceeds 1 in magnitude and decays to zero for far-away points, which is why extreme outliers barely budge the MLE.

Even when no closed-form trick exists, automatic differentiation computes both forms of I directly. The cell below draws $10{,}000$ Cauchy observations at $\theta^\star = 0$ and prints the mean score, the variance of the score, and minus the mean curvature. Using the uniform-angle representation, the Monte Carlo standard errors of these three estimates are about 0.007, 0.004, and 0.008.

```
def log_p(theta, x):  # log density of the Cauchy location model
    return -torch.log1p((x - theta)**2) - np.log(np.pi)

x = torch.tensor(np.random.default_rng().standard_cauchy(10_000),
                 dtype=torch.float32)  # theta* = 0
score = torch.func.vmap(torch.func.grad(log_p), in_dims=(None, 0))
curv = torch.func.vmap(torch.func.grad(torch.func.grad(log_p)),
                       in_dims=(None, 0))
s, h = score(torch.tensor(0.0), x), curv(torch.tensor(0.0), x)
print(f'mean score {s.mean():.3f}, variance of score {s.var():.3f}, '
      f'-mean curvature {-h.mean():.3f}')
```

```
mean score 0.004, variance of score 0.498, -mean curvature 0.502
```

`torch.func.grad` differentiates the log-density of one observation, and `vmap` maps it over all $10{,}000$ observations; applying `grad` twice gives the curvature.

The three sample estimates land within two or three standard errors of their theoretical values 0, 0.5, and 0.5.

36.6.3.2 The Error Bar of the Coin

The Fisher information gives us the error bar for $9/13$ that we asked for at the start of the section. In the next subsection we will show that, for large n, the MLE has variance approximately $1/(nI(\theta))$. Taking that result on trust for a moment, we can apply it right away. For the coin, $1/(nI) = \theta(1-\theta)/n$ by Equation 36.84, and plugging in $\hat\theta$ for the unknown θ yields the estimated standard error

$$\widehat{\mathrm{se}} = \sqrt{\frac{\hat\theta(1-\hat\theta)}{n}} = \sqrt{\frac{(9/13)(4/13)}{13}} = 0.128.$$

Thus $\hat{\theta} = 0.69 \pm 0.13$. The approximate 95% confidence interval $\hat{\theta} \pm 1.96\,\widehat{\mathrm{se}} = [0.44,\ 0.94]$ comfortably contains $1/2$, so thirteen flips cannot distinguish this coin from a fair one. Pinning θ down to ± 0.05 at 95% confidence would require roughly $1.96^2\,\hat{\theta}(1-\hat{\theta})/0.05^2 = 1.96^2 \cdot 0.213/0.05^2 \approx 330$ flips.

Crucially, the exact same number comes straight out of the loss itself. The second derivative of the coin's NLL evaluated at $\hat{\theta}$ is

$$\ell''(\hat{\theta}) = \frac{n_H}{\hat{\theta}^2} + \frac{n_T}{(1-\hat{\theta})^2} = \frac{9}{(9/13)^2} + \frac{4}{(4/13)^2} = 61.0,$$

and $1/\sqrt{61.0} = 0.128$. For the coin, this agreement is exact: since $n_H = n\hat{\theta}$ and $n_T = n(1-\hat{\theta})$, the observed curvature equals $n/(\hat{\theta}(1-\hat{\theta})) = nI(\hat{\theta})$. For any model that satisfies our regularity conditions, the same relationship holds approximately at large n. After fitting a model, we can compute the curvature of the NLL at its minimum (its Hessian) and take one over its square root as the error bar. For a vector of parameters, the individual error bars are the square roots of the diagonal entries of the inverse Hessian. A deep neural network violates these regularity conditions, and Section 36.6.3.5 explains what survives of this recipe there. Curvature also explains the likelihood plot at the opening of the section: ten times the data multiplies the curvature by ten and narrows the peak by $\sqrt{10} \approx 3.2$.

36.6.3.3 Consistency and Asymptotic Normality

Why is the variance of the MLE approximately $1/(nI)$? To answer that, we first need assurance that the MLE heads toward the right value at all, and then a description of how it fluctuates around that target. Three tools from earlier sections do the work. From Section 36.5 we use consistency ($\hat{\theta} \xrightarrow{P} \theta^\star$) and the weak law of large numbers. From Section 36.3 we use the central limit theorem, which states that a sum of n i.i.d. mean-zero terms with variance v, divided by $\sqrt{n}$, converges in distribution ($\xrightarrow{d}$) to $\mathcal{N}(0, v)$.

The cross-entropy identity of Section 36.6.2.1 explains why the MLE approaches the true parameter. Suppose the data are drawn from the model with true parameter $\theta^\star$. Because the average NLL is an average of i.i.d. terms, the law of large numbers implies that it converges to the population NLL,

$$\mathbb{E}_{\theta^\star}\big[-\log p(x \mid \theta)\big] = H(p_{\theta^\star}) + D_{\mathrm{KL}}(p_{\theta^\star} \,\|\, p_\theta),$$

with differential entropy h replacing H for continuous densities (Section 36.6.2.2). By Gibbs' inequality (Section 36.4.2), this objective attains its minimum where the KL divergence vanishes, namely at $\theta = \theta^\star$. That minimizer is unique if the model is **identifiable**, meaning distinct parameter vectors always define distinct distributions. Neural networks are not identifiable: permuting the hidden units of a layer along with their incoming and outgoing weights changes the parameter vector without changing the function computed. For such models, consistency is a statement about the fitted distribution, or about the equivalence class of parameters producing it, rather than about a single parameter vector.

We state the consistency guarantee without proof (Wasserman 2013). The MLE is consistent when four conditions hold: the data are i.i.d. from $p_{\theta^\star}$; the model is identifiable; the population optimum is well separated, so no θ far from $\theta^\star$ comes arbitrarily close to the optimal

value (on a compact parameter set with a continuous objective, uniqueness suffices); and the average NLL converges to the population NLL uniformly in θ. Uniformity is the subtle requirement that is easy to overlook: the law of large numbers guarantees convergence at each fixed θ, and pointwise convergence alone does not force the minimizers to converge. For the coin, of course, no theorem is needed: $\hat{\theta} = n_H/n \to \theta^\star$ directly by the law of large numbers.

Consistency tells us where $\hat{\theta}$ lands. To get an error bar, we need the size and shape of its fluctuations around $\theta^\star$. For a scalar parameter θ, the derivation takes three short steps, and its one clever idea is to Taylor-expand the first-order optimality condition rather than the NLL itself. Write $S_n(\theta) = \sum_i s(x_i;\theta)$ for the total score.

I. By definition, the MLE solves $S_n(\hat{\theta}) = 0$. Expanding $S_n(\hat{\theta})$ to first order around $\theta^\star$ gives

$$0 = S_n(\hat{\theta}) \approx S_n(\theta^\star) + S_n'(\theta^\star)\,(\hat{\theta} - \theta^\star).$$

II. Solving for the estimation error and multiplying by $\sqrt{n}$ writes it as a ratio of two averages,

$$\sqrt{n}\,(\hat{\theta} - \theta^\star) \approx \frac{S_n(\theta^\star)/\sqrt{n}}{-S_n'(\theta^\star)/n}.$$

III. The numerator is a sum of n i.i.d. scores, each with mean zero and variance $I(\theta^\star)$, divided by $\sqrt{n}$; by the central limit theorem it converges in distribution to $\mathcal{N}(0, I(\theta^\star))$. The denominator is the average of n i.i.d. terms $-\partial_\theta s(x_i;\theta^\star)$ whose expectation is $I(\theta^\star)$ by the information identity; by the law of large numbers it converges to $I(\theta^\star)$. Dividing a Gaussian of variance I by the constant I produces a Gaussian of variance $I/I^2 = 1/I$.

Notice how cleanly the two forms of Fisher information meet here: the numerator uses the variance of the score, while the denominator uses the expected curvature. The information identity makes both equal to the same number I, which is why the final formula is so simple. The vector case follows the exact same algebra with matrices, yielding the covariance $I^{-1} I I^{-1} = I^{-1}$. A rigorous proof also bounds the Taylor remainder, which is the role of the integrable curvature bound in the proposition below.

Proposition (asymptotic normality). *Under the regularity conditions, the conditions for consistency above, and a bound $\|\nabla_\theta^2 \log p(x \mid \theta)\| \le M(x)$ with $\mathbb{E}_{\theta^\star}[M(x)] < \infty$ for all θ near $\theta^\star$,*

$$\sqrt{n}\,(\hat{\theta} - \theta^\star) \;\xrightarrow{d}\; \mathcal{N}\!\left(\mathbf{0},\; I(\theta^\star)^{-1}\right) \tag{36.85}$$

(Wasserman 2013). We will not prove this beyond the three steps above, but it is the central result of the section: the error bar, the efficiency of the MLE, and the limits of the theory all rest on it. In plain terms, the estimation error of the MLE is approximately Gaussian, shrinks at the rate $1/\sqrt{n}$, and has covariance given by the inverse Fisher information. At finite n we use the approximation $\mathrm{Cov}(\hat{\theta}) \approx I(\theta^\star)^{-1}/n$, which is the error bar we applied to the coin. In particular, halving the error bar requires four times as much data. When the model is misspecified, the variance and curvature forms of I no longer coincide, and the covariance becomes the "sandwich" formula of Section 36.6.3.5.

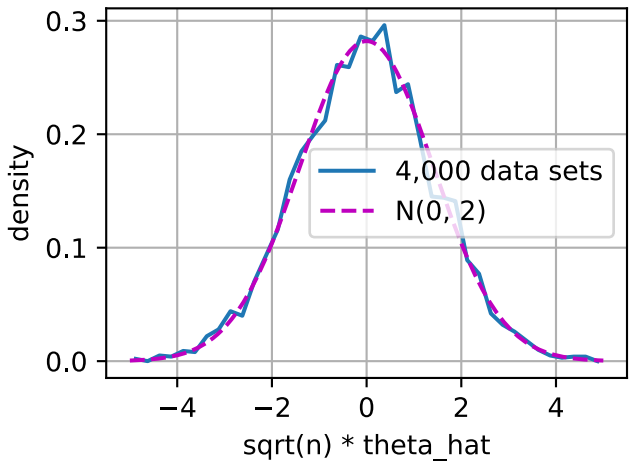
To test asymptotic normality in a nontrivial setting, we need a model whose MLE is not a sample mean, and the Cauchy location model fits the bill. Its likelihood equation $\sum_i 2u_i/(1+u_i^2) = 0$ has no closed-form solution, so we solve it numerically with **Fisher scoring**, which is Newton's method with the expected curvature $nI = n/2$ substituted for the observed curvature:

$$\theta \leftarrow \theta + \frac{S_n(\theta)}{nI} = \theta + \frac{2}{n} S_n(\theta).$$

Replacing the observed curvature by nI is more than a convenience: the expected curvature is always positive, whereas the observed curvature of a Cauchy NLL can turn negative away from the minimum, causing a plain Newton step to head uphill. We initialize each run at the sample median, which is already close to the solution, and fit $4{,}000$ data sets of $n = 100$ observations at $\theta^\star = 0$ using ten scoring steps each. If the asymptotic approximation is already accurate at $n = 100$, the histogram of $\sqrt{n}\,\hat\theta$ should match $\mathcal{N}(0, 1/I) = \mathcal{N}(0, 2)$.

```
reps, n = 4000, 100
x = np.random.default_rng().standard_cauchy((reps, n))  # theta* = 0
theta = np.median(x, axis=1)
for _ in range(10):  # Fisher scoring: theta += S_n / (n I) with I = 1/2
    u = x - theta[:, None]
    theta += (2 / n) * (2 * u / (1 + u**2)).sum(axis=1)
z = np.sqrt(n) * theta  # sqrt(n) (theta_hat - theta*)
hist, edges = np.histogram(z, bins=40, range=(-5, 5), density=True)
grid = np.linspace(-5, 5, 201)
d2l.plot([(edges[:-1] + edges[1:]) / 2, grid],
         [hist, np.exp(-grid**2 / 4) / np.sqrt(4 * np.pi)],
         'sqrt(n) * theta_hat', 'density', legend=['4,000 data sets', 'N(0, 2)'])
print(f'standard deviation of sqrt(n) * theta_hat: {z.std():.2f}')
```

```
standard deviation of sqrt(n) * theta_hat: 1.43
```



The histogram tracks the predicted curve in center, width, and height. Across runs, the standard deviation of $\sqrt{n}\,\hat\theta$ lands between roughly 1.40 and 1.47, close to the asymptotic

limit $\sqrt{2} \approx 1.41$; at $n = 100$ the variance still sits a few percent above its limiting value.

36.6.3.4 The Cramér–Rao Bound

Is a variance of $1/(nI)$ good, or could a cleverer estimator beat it? As it turns out, the Fisher information also sets a hard floor that no estimator which is right on average can break through. Recall from Section 36.5 that an estimator $\tilde{\theta}$ is *unbiased* if $\mathbb{E}_\theta[\tilde{\theta}] = \theta$ for every θ.

Proposition (Cramér–Rao bound). *Under the regularity conditions, any unbiased estimator $\tilde{\theta}$ of a scalar parameter from n i.i.d. observations satisfies*

$$\operatorname{Var}(\tilde{\theta}) \;\geq\; \frac{1}{n\, I(\theta)} \tag{36.86}$$

(Cramér 1946; Radhakrishna Rao 1945). In other words, no unbiased estimator can be more precise than the curvature of the likelihood permits. The proof takes just three lines, and its single idea is to correlate the estimator with the score.

Proof. Let $S_n = \partial_\theta \log P(X \mid \theta)$ be the score of the entire data set, a sum of n independent per-example scores; it has mean zero and variance $nI(\theta)$. Unbiasedness states that $\int \tilde{\theta}(X)\, P(X \mid \theta)\, dX = \theta$. Differentiating both sides under the integral sign and substituting $\partial_\theta P = P\, S_n$ yields

$$1 = \int \tilde{\theta}(X)\, S_n\, P(X \mid \theta)\, dX = \mathbb{E}[\tilde{\theta}\, S_n] = \operatorname{Cov}(\tilde{\theta},\, S_n),$$

where the last equality uses $\mathbb{E}[S_n] = 0$. Applying the Cauchy–Schwarz inequality for covariances gives $1 \leq \operatorname{Var}(\tilde{\theta})\operatorname{Var}(S_n) = \operatorname{Var}(\tilde{\theta})\, nI(\theta)$. ∎

Unbiasedness enters through the "1" on the left-hand side, while fixed support enters when we differentiate under the integral sign; Exercise 7 exhibits an unbiased estimator that beats the bound when the support moves with θ. For a vector parameter, the bound generalizes to $\operatorname{Cov}(\tilde{\theta}) \succeq I(\theta)^{-1}/n$ in the positive semidefinite order.

The MLE's limiting variance Equation 36.85 matches this floor exactly, making the MLE *asymptotically efficient*: in the large-sample limit, no unbiased estimator achieves smaller variance. For the coin, the floor is attained at every finite n, not merely in the limit, because n_H/n is an unbiased sample mean with variance $\theta(1-\theta)/n = 1/(nI)$. In general, however, the MLE reaches the floor only as $n \to \infty$. At finite n it can be biased, in which case the bound does not apply to it, as we will see next for the Gaussian variance.

How much does asymptotic efficiency matter in practice? On the Cauchy location model, the sample median is a natural competitor and the sample mean is the naive one; by symmetry, both the MLE and the median are unbiased. The Cramér–Rao floor for n times the variance is $1/I = 2$, whereas the asymptotic theory of the median predicts $\pi^2/4 \approx 2.47$. Before reading the output below, pause to guess what the sample mean will do. The cell reuses our $4{,}000$ data sets.

```
print(f'n Var(MLE) = {n * theta.var():.2f}, '
      f'n Var(median) = {n * np.median(x, axis=1).var():.2f}, '
```

```
        f'n Var(mean) = {n * x.mean(axis=1).var():.3g}')
```

```
n Var(MLE) = 2.06, n Var(median) = 2.50, n Var(mean) = 7.79e+05
```

The MLE lands between roughly 1.9 and 2.2, right near the floor of 2, while the median sits near 2.5; the MLE achieves the same error bar with about 20% fewer observations. The sample mean, meanwhile, exhibits an enormous variance that fluctuates wildly from run to run: the average of n independent Cauchy variables is still Cauchy, so averaging never helps at all.

36.6.3.5 Limits of the Theory

The guarantees above hold under the stated regularity conditions and in the limit $n \to \infty$. Each of the first four cases below breaks one of those conditions and is worked out in the exercises. The fifth case, a misspecified model, is the regime every deep-learning practitioner lives in, so we examine it here.

- *Finite-sample bias.* The MLE of a Gaussian's variance divides the sum of squared residuals by n, underestimating σ^2 on average by the factor $(n-1)/n$ (Section 36.5). The MLE is optimal in the limit, not unbiased at every n.
- *The boundary.* When a true parameter sits near the edge of its range, such as a coin with bias 0.95, the Gaussian approximation needs far more data to become accurate: what governs it is the expected count of the rare outcome, not n alone. On the boundary itself, the MLE may not even exist (Exercise 4).
- *A support that moves with the parameter.* For the uniform distribution on $[0, \theta]$, the MLE is $\max_i x_i$. Its error shrinks at rate $1/n$ rather than $1/\sqrt{n}$, and an unbiased rescaling beats the Cramér–Rao floor (Exercise 7).
- *Singular information.* The rescaling symmetry of a ReLU network gives the Fisher matrix I a null vector, so I^{-1} does not exist (Exercise 6).

Misspecification. When no member p_θ of the model family matches the true data-generating distribution, the MLE converges to the parameter that minimizes the KL divergence to the truth (White 1982). The expected score still vanishes at that limit, but the variance of the score, B, and the expected curvature of the NLL, A, no longer agree. Step III of our derivation then yields variance B/A^2 instead of $1/I$, or in matrix form the *sandwich* covariance $\mathbf{A}^{-1}\mathbf{B}\mathbf{A}^{-1}$ (Exercise 8). Consider, for example, fitting a Poisson model with rate λ to overdispersed counts that have mean m and variance $m + m^2/\alpha$. Because the Poisson MLE is the sample mean, it still recovers m, with true variance $(m + m^2/\alpha)/n$. Yet the Poisson model's Fisher information $I = 1/\lambda$ predicts variance m/n, which is too small by the factor $1 + m/\alpha$, which is 3 when $m = 4$ and $\alpha = 2$.

A deep neural network is both misspecified and non-identifiable, so we cannot attach classical error bars to individual weights. What the theory still supports are statements about the fitted predictive distribution, which parameter symmetries leave unchanged. The Fisher matrix itself will reappear twice later in this book: as the local curvature (Hessian) of the KL divergence between neighboring distributions in a parametric family (Section 37.3), and as the preconditioner used by natural-gradient optimization (Section 35.5).

36.6.4 MAP Estimation and Regularization

Thirteen flips gave us a sensible estimate; three flips can break maximum likelihood completely. Suppose three flips come up HHH. The MLE is $\widehat{\theta} = 3/3 = 1$, which assigns probability zero to tails, so the very first tail we observe later incurs an infinite NLL. Any count-based classifier that encounters a feature-class combination it never saw during training faces the exact same pathology. Nobody who has ever handled a coin believes that three heads prove tails is impossible. We need a principled way to express, before seeing any data, that coins are rarely completely one-sided.

A probability distribution over the parameters captures such prior knowledge. Maximum likelihood had no room for it, because it placed probabilities only on data sets. Introducing a **prior** $p(\theta)$, Bayes' rule combines the prior with the likelihood to produce a distribution over the parameters given the observed data:

$$p(\theta \mid X) = \frac{P(X \mid \theta)\, p(\theta)}{P(X)}, \qquad P(X) = \int P(X \mid \theta)\, p(\theta)\, d\theta. \tag{36.87}$$

The left-hand side is the **posterior**, representing our updated belief after seeing the data. The denominator is the **evidence**, the marginal probability of the data averaged over all parameter values; crucially, it does not depend on θ. The **maximum a posteriori** (MAP) estimate is the mode (peak) of the posterior. Because the evidence is constant in θ, maximizing the posterior and taking negative logarithms gives

$$\begin{aligned} \widehat{\theta}_{\text{MAP}} &= \operatorname*{argmax}_{\theta}\ P(X \mid \theta)\, p(\theta) \\ &= \operatorname*{argmin}_{\theta}\ \Big[\underbrace{-\textstyle\sum_i \log p(x_i \mid \theta)}_{\text{data fit (NLL)}}\ \underbrace{-\log p(\theta)}_{\text{penalty}} \Big]. \end{aligned} \tag{36.88}$$

MAP estimation therefore minimizes the familiar NLL plus an additive penalty, the negative log-prior. A prior that assigns low probability to extreme parameter values penalizes them, playing the exact role of a regularizer. For HHH, a prior equivalent to adding one imaginary head and one imaginary tail (the Beta$(2,2)$ prior below) yields $\widehat{\theta}_{\text{MAP}} = (3{+}1)/(3{+}2) = 4/5$. A subsequent tail then costs $-\log(1/5) = 1.6$ nats, a finite surprise rather than an infinite one.

36.6.4.1 Gaussian Prior and Weight Decay

Weight decay, the standard regularizer in deep learning, is the MAP penalty of a Gaussian prior. This holds for weight decay in its L_2-penalty form; the decoupled variant used with Adam (Section 9.5.3) is no longer exactly a MAP penalty. Section 2.7 derived the correspondence for linear regression; here we state it in the notation of this section for any model with Gaussian target noise. Assume each parameter is a priori small and independent, $\theta_j \sim \mathcal{N}(0, \tau^2)$, where τ reflects the typical magnitude we expect. The prior density is proportional to $\exp(-\|\theta\|_2^2/(2\tau^2))$, so its negative logarithm equals $\|\theta\|_2^2/(2\tau^2)$ up to a constant. Combining this with the Gaussian NLL Equation 36.82, the MAP objective Equation 36.88 becomes

$$\hat{\theta}_{\text{MAP}} = \operatorname*{argmin}_{\theta} \frac{1}{2\sigma^2} \sum_i (y_i - \hat{y}_i)^2 + \frac{1}{2\tau^2} \|\theta\|_2^2.$$

Multiplying through by the positive constant σ^2/n leaves the minimizer unchanged and recovers the weight-decay objective of Section 2.7,

$$\frac{1}{n} \sum_{i=1}^{n} \frac{1}{2} \left(y_i - \hat{y}_i\right)^2 + \frac{\lambda}{2} \|\theta\|_2^2, \qquad \lambda = \frac{\sigma^2}{n\tau^2}.$$

Leaving the bias b unpenalized, as this book usually does, corresponds to a flat prior on b. Notice that the penalty weight λ in the averaged loss shrinks as $1/n$: the prior matters most when data are scarce and fades away as the sample grows. Section 35.7.3.2 interprets the same penalty equivalently as a constraint on the norm of the weights.

Replacing the Gaussian prior with a *Laplace* prior $p(\theta) \propto e^{-\|\theta\|_1/s}$ of scale s produces the L_1 penalty $\|\theta\|_1/s$, whose sharp kink at zero encourages exact zeros and thus sparse parameter vectors. Figure 36.33 illustrates this tug-of-war in one dimension: the NLL pulls toward the MLE, the negative log-prior pulls toward the prior mean, and the MAP estimate settles in between.

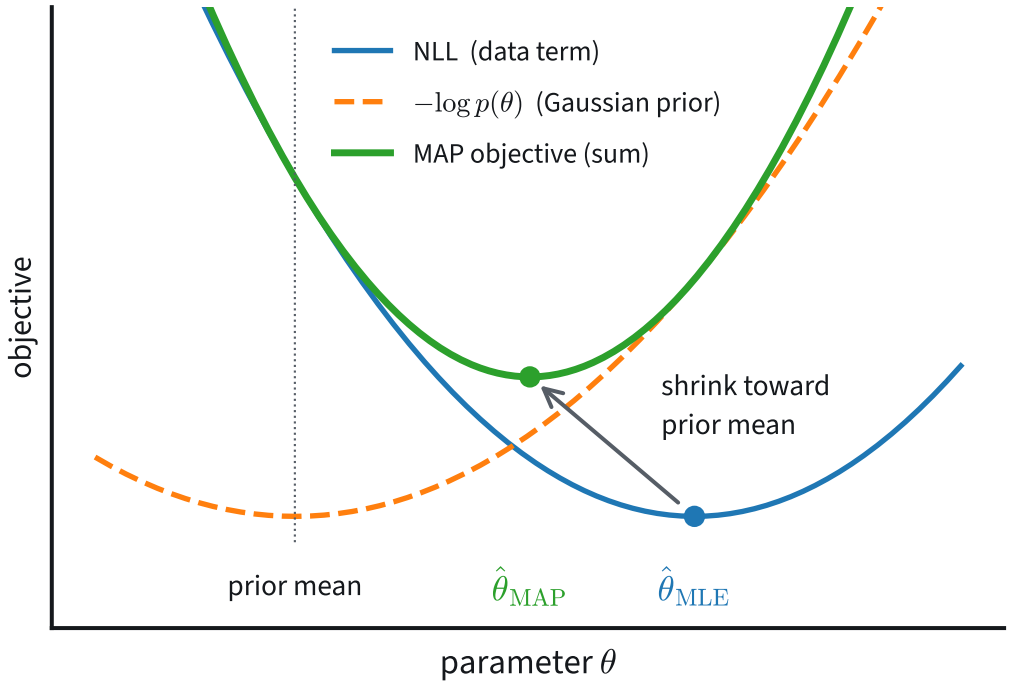


Figure 36.33: Maximum a posteriori balances data against prior. The negative log-likelihood (blue) is minimized at the MLE; adding the negative log of a Gaussian prior centered at the prior mean (orange, dashed) produces the MAP objective (green), pulling the minimum from the MLE toward the prior mean. A tighter prior (smaller τ) pulls harder, whereas as $\tau \to \infty$ the prior flattens and MAP reduces to the MLE.

The same diagram makes both limiting cases immediate. As the prior flattens ($\tau \to \infty$ or $s \to \infty$), the penalty disappears; and as the sample size n grows, the NLL deepens in proportion to n while the prior stays fixed. In either limit, MAP converges back to the MLE.

36.6.4.2 Beta Prior and Pseudo-Counts

For the coin, the parameter θ lives on $[0, 1]$, making the Beta family the natural prior. A Beta(a, b) prior with $a, b > 0$ has density proportional to $\theta^{a-1}(1-\theta)^{b-1}$. Multiplying by the coin likelihood Equation 36.80 simply adds the observed counts to the exponents, so the posterior remains in the Beta family, the conjugate update of Section 36.4:

$$p(\theta \mid X) \;\propto\; \theta^{\,n_H+a-1}\,(1-\theta)^{\,n_T+b-1}.$$

This posterior has the exact functional form of a coin likelihood with $n_H + a - 1$ heads and $n_T + b - 1$ tails. Whenever both exponents are positive, maximizing it is identical to finding the coin's MLE: "head exponent over total exponent":

$$\hat{\theta}_{\text{MAP}} \;=\; \frac{n_H + a - 1}{n + a + b - 2}, \qquad n = n_H + n_T. \tag{36.89}$$

For the posterior mode, then, the prior acts as $a - 1$ imaginary heads and $b - 1$ imaginary tails appended to the sample before taking the empirical frequency. This matches the convention in Section 36.4: a and b are the *pseudo-counts* of the posterior mean, whereas $a - 1$ and $b - 1$ are those of the posterior mode. A Beta$(2, 2)$ prior adds one imaginary flip of each outcome to the mode, shifting our $9/13 \approx 0.692$ to $(9+1)/(13+2) = 2/3$ and HHH to $4/5$. A strong Beta$(30, 30)$ prior anchors the MAP estimate near $1/2$ until the observed flips outweigh its 58 imaginary ones.

These two summaries meet in a classic formula. Under the flat prior Beta$(1, 1)$, the posterior mean is Laplace's rule of succession, $(n_H+1)/(n+2)$ (Laplace 1814), widely known in count-based classifiers as add-one smoothing. That same ratio is the MAP estimate under Beta$(2, 2)$, whereas MAP under the flat Beta$(1, 1)$ prior just returns the plain MLE. Add-one smoothing is therefore a posterior mean under a flat prior, not a flat-prior MAP estimate.

36.6.4.3 Posterior Mode and Posterior Distribution

The posterior mode has a subtle weakness: it depends on how we parameterize the model. Under a nonlinear change of variables, a probability density is multiplied by a Jacobian factor, which shifts its peak even though the probabilities of all regions stay identical. A Beta$(1, 1)$ prior is flat in θ, yet the exact same belief expressed in terms of the log-odds $\eta = \log \frac{\theta}{1-\theta}$ is not flat, and taking the MAP estimate in η produces $10/15$ rather than $9/13$ (Exercise 5). The likelihood has no Jacobian because its values are data probabilities attached directly to points in parameter space. Consequently, the MLE is **equivariant**: if $\widehat{\theta}$ is the MLE of θ, then $g(\widehat{\theta})$ is the MLE of $g(\theta)$ for any one-to-one transformation g (Wasserman 2013). For instance, the MLE of the coin's log-odds is simply $\log \frac{9/13}{4/13} = \log \frac{9}{4}$. This is why maximum likelihood is defined without a prior, rather than as "MAP with a flat prior", whose answer would shift whenever we rewrote the parameter. Equivariance has a trade-off of its own: as the same exercise shows, no estimator can be both equivariant and unbiased under every reparameterization.

An even deeper limitation is that MAP, just like the MLE, collapses our knowledge into a single parameter vector. The full posterior $p(\theta \mid X)$ captures how uncertain the parameters remain after seeing the data. Making predictions by averaging over that uncertainty, $\int p(x \mid \theta)\, p(\theta \mid X)\, d\theta$, accounts for parameter uncertainty, whereas plugging in the mode discards it. Because such integrals rarely have closed forms, approximating them is the focus of Section 36.8.

A more immediate obstacle comes first. Everything in this section assumed that we can write down the probability of each observation directly as a function of the parameters. When each data point comes from one of several hidden sources and we do not observe which one, that probability becomes a sum over the hidden possibilities, and the logarithm of a sum no longer splits cleanly into simple terms. Section 36.7 shows what to optimize instead.

36.6.5 Summary

Maximum likelihood selects the parameters under which the observed data are most probable; in practice, this means minimizing the negative log-likelihood, a sum of per-example surprises. Most standard training losses arise this way from a choice of noise model: squared error from Gaussian noise, absolute error from Laplace noise, and cross-entropy from discrete outcomes, where the average loss can never fall below the entropy of the empirical frequencies. The curvature of the same loss quantifies precision: under regularity conditions, the MLE's error is approximately Gaussian with variance $1/(nI)$, where the Fisher information I is both the variance of the score and the expected curvature of the NLL, matching the Cramér–Rao lower bound for unbiased estimators. These classical guarantees break down when a model is misspecified or non-identifiable, and deep neural networks are both. Finally, introducing a prior turns MAP estimation into NLL plus a regularizer: a Gaussian prior yields weight decay and a Beta prior adds pseudo-counts, though the MAP estimate still reduces the posterior to a single point.

36.6.6 Exercises

1. **The cross-entropy floor.** For discrete outcomes the average NLL equals $H(\hat{p}_{\text{data}}) + D_{\text{KL}}(\hat{p}_{\text{data}} \,\|\, p_\theta)$ by Section 36.6.2.1, so it cannot fall below the empirical entropy.
   1. Evaluate this floor in nats for the coin sequence with 9 heads and 4 tails, and the KL gap of the fair-coin model $\theta = 1/2$.
   2. A classifier models $p(y \mid \mathbf{x})$ and can represent any distribution over labels separately for each distinct input. Show that its smallest achievable average NLL on a training set is the empirical conditional entropy $\sum_{\mathbf{x}} \hat{p}_{\text{data}}(\mathbf{x})\, H(\hat{p}_{\text{data}}(\cdot \mid \mathbf{x}))$. Evaluate it for a binary task in which each of 100 distinct inputs occurs 10 times and carries the wrong label in exactly one of its occurrences.
   3. Now every training input occurs once, as in a typical image data set. What is the floor of (b)? Why does a training loss near zero then say nothing about how noisy the labels are? Which quantity that we can measure has the population conditional entropy $H(Y \mid X)$ as its floor?
2. **Learned noise variance.** The Gaussian NLL Equation 36.82 holds the variance σ^2 fixed. Here the variance is learned as well.
   1. Let one shared σ^2 be a parameter alongside θ. Minimize the NLL over σ^2 for fixed θ and show that the remaining objective is $\frac{n}{2} \log\left(\frac{1}{n}\sum_i (y_i - \hat{y}_i)^2\right)$ plus a constant. Is the minimizer over θ still the least-squares fit?

2. A network outputs a mean $\hat{y}(\mathbf{x})$ and a variance $\hat{\sigma}^2(\mathbf{x}) > 0$ for each input. Up to an additive constant, its NLL on the training set is

$$\sum_{i=1}^{n} \Big[\frac{(y_i - \hat{y}(\mathbf{x}_i))^2}{2\hat{\sigma}^2(\mathbf{x}_i)} + \frac{1}{2} \log \hat{\sigma}^2(\mathbf{x}_i) \Big].$$

   Show that this objective is unbounded below if the network can fit one training point exactly while shrinking its variance. Which property of the Gaussian mixture likelihood in Section 36.7 has the same cause?
3. Compute the gradient of example i's term with respect to $\hat{y}(\mathbf{x}_i)$. Suppose that the predicted variance tracks the squared residual, $\hat{\sigma}^2(\mathbf{x}_i) \approx r_i^2$ with $r_i = y_i - \hat{y}(\mathbf{x}_i)$. How does the size of the gradient then depend on $|r_i|$, and how does that compare with squared error? What does this weighting do to the fit of the mean in regions where the fit is currently poor (Seitzer et al. 2022)?

3. **Losses that are not likelihoods.** Let $\phi(y, o)$ be a loss of a label y and a model output o, and let $Z(o) = \sum_y e^{-\phi(y,o)}$. If $Z(o)$ does not depend on o, then $e^{-\phi(y,o)}/Z$ is a conditional distribution of y given o whose NLL is ϕ plus a constant, so minimizing ϕ is maximum likelihood.
   1. For labels $y \in \{-1, +1\}$, show that the logistic loss $\phi(y, o) = \log(1 + e^{-yo})$ has $Z(o) = 1$, and identify the probability that the model assigns to $y = +1$.
   2. Compute $Z(o)$ for the hinge loss $\phi(y, o) = \max(0, 1 - yo)$ of support vector machines, separately for $|o| \le 1$ and for $|o| > 1$. Conclude that the hinge loss is not the NLL, up to a constant, of any conditional distribution of y given o. Which term must be added to the hinge loss to obtain the NLL of $e^{-\phi(y,o)}/Z(o)$, and how does that term behave as $|o|$ grows?
   3. For a real target y and a prediction $\hat{y}$, take any loss of the residual, $\phi(y, \hat{y}) = \rho(y - \hat{y})$ with $\int e^{-\rho(r)}\, dr < \infty$, such as the Huber loss. Show that $Z(\hat{y}) = \int e^{-\rho(y-\hat{y})}\, dy$ does not depend on $\hat{y}$. Why is the condition of this problem automatic for such regression losses but restrictive for classification losses?

4. code **Boundaries: separable data and rare outcomes.** Maximum likelihood misbehaves when the best parameter sits on the edge of its range.
   1. Two labeled points, $x = -1$ with $y = 0$ and $x = +1$ with $y = 1$, are fit by logistic regression $P(y = 1 \mid x) = \text{sigmoid}(wx)$ without a bias. Show that the NLL is $2\log(1 + e^{-w})$, that it decreases for every w and tends to 0, and that no finite w attains it. Which coin sequence of the text has the same problem, and what does $w \to \infty$ predict for a new point at $x = 0.1$?
   2. Add the prior $w \sim \mathcal{N}(0, \tau^2)$. Show that the MAP estimate solves $2\,\text{sigmoid}(-w) = w/\tau^2$, and solve it numerically for $\tau = 1$. Generalize: for any linearly separable data set the logistic NLL has infimum 0 and no minimizer, and a Gaussian prior restores a finite MAP estimate.
   3. For the coin, the skewness of $\hat{\theta} = n_H/n$ is $(1 - 2\theta)/\sqrt{n\theta(1-\theta)}$. Evaluate it for $\theta = 0.7$ and for $\theta = 0.95$ at $n = 400$, and find the n at which $\theta = 0.95$ reaches the skewness that $\theta = 0.7$ has at $n = 400$. For θ near 1, express the skewness through the expected number of tails $n(1 - \theta)$. Why is it that count, and not n, which decides whether the Gaussian limit is accurate?
   4. Simulate $20{,}000$ data sets of $n = 400$ flips (`rng.binomial(n, theta, 20000) / n`), and histogram $\sqrt{n}(\hat{\theta} - \theta)$ for $\theta = 0.7$ and $\theta = 0.95$ against $\mathcal{N}(0, \theta(1 - \theta))$. Describe how the $\theta = 0.95$ histogram departs from the curve.

5. **MAP in log-odds.** Parameterize the coin by its log-odds $\eta = \log \frac{\theta}{1-\theta}$, so that $\theta =$

sigmoid(η) and $\frac{d\theta}{d\eta} = \theta(1-\theta)$. Keep the uniform prior Beta$(1,1)$ on θ and the sequence with 9 heads and 4 tails.

1. Use the change-of-variables formula for densities, $p_\eta(\eta) = p_\theta(\theta)\,|d\theta/d\eta|$ of Section 36.2 to find the prior density of η that this belief implies. Show that it is the logistic density $e^{-\eta}/(1+e^{-\eta})^2$, the distribution of $\log \frac{U}{1-U}$ for a uniform U, and explain the result with inverse-transform sampling, without a calculation.
2. Find the MAP estimate of η and map it back to θ. Compare it with the MAP estimate in the θ parameterization, with the MLE $\log \frac{9}{4}$ of Section 36.6.4.3, and with Laplace's rule of succession.
3. Repeat (b) for a general prior Beta(a,b) and counts n_H, n_T. Which summary of the posterior over θ does the log-odds MAP estimate reproduce?
4. The MLE is equivariant, so the MLE of θ^2 is $\widehat{\theta}^2$. Show that its bias is $\mathbb{E}[\widehat{\theta}^2] - \theta^2 = \mathrm{Var}(\widehat{\theta}) = \theta(1-\theta)/n$, and evaluate it at $\theta = 9/13$, $n = 13$. Conclude that no estimator can be both equivariant under all smooth one-to-one reparameterizations and unbiased for all of them.

6. **Symmetries of a ReLU network.** ● Consider the network $f_\theta(x) = \sum_{j=1}^{h} a_j\,\mathrm{ReLU}(w_j x)$ with a scalar input x and parameters $\theta = (a_1, w_1, \dots, a_h, w_h)$, and the regression model $y \sim \mathcal{N}(f_\theta(x), \sigma^2)$ with fixed σ^2. Identifiability is defined in Section 36.6.3.
   1. Show that replacing (a_j, w_j) by $(c\,a_j,\ w_j/c)$ for any $c > 0$ leaves f_θ unchanged. How many distinct parameter vectors with the same function does permuting the hidden units produce when no two units coincide?
   2. Show from Equation 36.83 that the Fisher information of this model is $I(\theta) = \mathbb{E}_x\big[\nabla_\theta f_\theta(x)\,\nabla_\theta f_\theta(x)^\top\big]/\sigma^2$. Differentiate the rescaling of (a) with respect to c at $c = 1$ to find a nonzero vector $\mathbf{v}$ with $I(\theta)\mathbf{v} = \mathbf{0}$. Which hypothesis behind Equation 36.85 fails?
   3. The Hessian entry of the mean squared error $\frac{1}{n}\sum_i (y_i - f_\theta(x_i))^2$ with respect to a_j is $\frac{2}{n}\sum_i \mathrm{ReLU}(w_j x_i)^2$. How does this entry change under the rescaling of (a)? Compute the entry for w_j, and explain why for a ReLU unit it contains no residual term (away from the kinks $w_j x_i = 0$). How does it change under the rescaling? What does this imply for measures of the flatness of a minimum that are based on Hessian eigenvalues (Dinh, Pascanu, et al. 2017)?

7. **A support that moves.** ● Let $x_1, \dots, x_n$ be independent and uniform on $[0, \theta]$ with unknown $\theta > 0$. This model violates one of the regularity conditions of Section 36.6.3.
   1. Write the likelihood as a function of θ and show that the MLE is $\widehat{\theta} = \max_i x_i$. Why does setting a derivative to zero not find it?
   2. Show that $P(n(\theta - \widehat{\theta}) > t) = \left(1 - \frac{t}{n\theta}\right)^n$ for $0 \le t \le n\theta$, and find the limit as $n \to \infty$. At which rate in n does the error shrink, and is the limit Gaussian?
   3. On the support, the score is $s(x;\theta) = -1/\theta$. Compute $\mathbb{E}_\theta[s]$. Which step in the derivation of the mean-zero score fails (compare exercise 1 of Section 34.7)?
   4. The unbiased estimator $\frac{n+1}{n}\widehat{\theta}$ has variance $\theta^2/(n(n+2))$. Compare it with the value $1/(n\,\mathbb{E}_\theta[s^2])$ that Equation 36.86 would give as a floor. Which step of the proof of the bound fails here?

8. [extended] **Misspecification and the sandwich.** ●● This problem carries out the heuristic derivation of asymptotic normality when the two forms of the Fisher information differ. When no member of the model family is the data-generating law, the MLE converges to the pseudo-true parameter $\theta^\dagger$ that minimizes the population KL

divergence (Section 36.6.3.5), and its spread is no longer I^{-1}/n (White 1982). Write $\mathbf{A} = -\mathbb{E}[\nabla_\theta^2 \log p(x \mid \theta^\dagger)]$ and $\mathbf{B} = \mathbb{E}[s(x;\theta^\dagger)\, s(x;\theta^\dagger)^\top]$, with expectations under the data-generating law.

1. Expand the first-order condition $\sum_i s(x_i;\widehat{\theta}) = \mathbf{0}$ to first order around $\theta^\dagger$, and apply the central limit theorem to $\frac{1}{\sqrt{n}}\sum_i s(x_i;\theta^\dagger)$ to argue that
$$\sqrt{n}\,(\widehat{\theta} - \theta^\dagger) \xrightarrow{d} \mathcal{N}\big(\mathbf{0},\ \mathbf{A}^{-1}\mathbf{B}\mathbf{A}^{-1}\big). \tag{36.90}$$
Why does $\mathbb{E}[s(x;\theta^\dagger)] = \mathbf{0}$ hold although the model is wrong? Show that the covariance reduces to the one in Equation 36.85 when the model is correct.
2. Fit a Poisson model with rate λ to counts from a negative binomial distribution with mean m and shape α, whose variance is $m + m^2/\alpha$. Find $\widehat{\lambda}$, $\lambda^\dagger$, $\mathbf{A}$, and $\mathbf{B}$. By which factor does the inverse Fisher information understate the variance of $\widehat{\lambda}$?
3. Fit the Poisson regression $\log \lambda(x) = w_0 + w_1 x$ with one feature by Newton's method, which for this model reads $\mathbf{w} \leftarrow \mathbf{w} + (\mathbf{Z}^\top \operatorname{diag}(\lambda_i)\mathbf{Z})^{-1}\mathbf{Z}^\top(\mathbf{y} - \lambda)$, with rows $(1, x_i)$ in $\mathbf{Z}$ and $\lambda_i = \lambda(x_i)$. An observation is now a pair (x_i, y_i), and $\log p(y_i \mid x_i, \mathbf{w})$ takes the place of $\log p(x \mid \theta)$ in $\mathbf{A}$ and $\mathbf{B}$; estimate both by sample averages at the fitted $\widehat{\mathbf{w}}$. Generate $2{,}000$ data sets of $n = 500$ pairs with $x \sim \mathcal{N}(0,1)$, $\mathbf{w} = (1, 0.5)$, and negative-binomial counts with $\alpha = 2$: draw a rate from the Gamma distribution with shape α and mean $\lambda(x)$, then a Poisson count with that rate. Compare three standard errors of $\widehat{w}_1$: from the inverse Fisher information, from the sandwich, and from the spread of $\widehat{w}_1$ across data sets. Which fraction of the nominal 95% intervals of the first two covers $w_1 = 0.5$? Repeat with $\alpha = 10^6$.
4. Exercise 6c of Section 36.4 shows that the Poisson fit still recovers $\mathbf{w}$ when the counts are overdispersed. Which part of the model would have to be wrong for $\mathbf{w}^\dagger$ to differ from $\mathbf{w}$? Which parameter does the sandwich interval of (c) cover in that case?

36.7 Latent Variables, EM, and the ELBO

Every likelihood we met in Section 36.6 was a clean product of factors, one per observation, so taking logarithms turned it into a sum we could differentiate term by term. That convenience rested on a quiet assumption: that we get to observe everything the model generates.

In practice, we often miss a crucial piece of the data. Imagine collecting 500 measurements drawn from two distinct populations, 300 from one and 200 from the other, only to discover that nobody recorded which population produced which measurement. If those labels had survived, fitting the model would take one line per population: a mixing fraction, a mean, and a variance. Without them, the probability of each measurement becomes a sum over the two populations it might have come from. Unfortunately, the logarithm of a sum does not split apart; every parameter ends up entangled in every term, and setting the gradient to zero leaves equations with no closed-form solution. The same obstacle appears whenever a model explains observations through hidden structure: the cluster of a customer, the topic of a document, or the random code a generator turns into an image. Such unobserved quantities are called *latent variables*.

The central idea of this section is simple and surprisingly effective: guess what we cannot see using probabilities rather than hard labels, and then fit the model as if those soft guesses were real data. What makes this idea work is that our guess does not even have to be right. For *any* probabilistic guess, the resulting easy objective lies below the true log-likelihood, and it touches the log-likelihood precisely when the guess equals the posterior distribution over the hidden labels. This lower bound is the key result to remember. Expectation–maximization climbs it directly, while variational inference and variational autoencoders climb it approximately whenever the exact posterior is out of reach.

We will develop this one idea from start to finish: first setting up mixture models, then deriving the lower bound and measuring its gap, and finally running the resulting algorithm on our 500 points, once from a starting point where it recovers both populations and once from a symmetric start where it gets stuck.

36.7.1 Mixture Models

Let us translate this missing-label puzzle into symbols. A probabilistic model with a latent variable z specifies a joint distribution over the hidden and observed quantities,

$$p(x, z \mid \theta) = p(z \mid \theta)\, p(x \mid z, \theta),$$

of which we observe only x. To compute the likelihood of what we actually saw, we must sum out all possible values of the hidden variable (or integrate them out when z is continuous), giving the *marginal* likelihood:

$$p(x \mid \theta) = \sum_{z} p(z \mid \theta)\, p(x \mid z, \theta). \tag{36.91}$$

In words, the model's probability for x adds up every way the model could have generated x, one term for each value of the hidden variable.

The canonical example is the **Gaussian mixture model** (GMM). Here the latent variable $z \in \{1, \dots, K\}$ selects component k with mixing probability π_k, and the observation is then drawn from that component's Gaussian, $x \mid z{=}k \sim \mathcal{N}(\mu_k, \sigma_k^2)$. Putting the n observations together, the negative log-likelihood (NLL) is

$$\ell(\theta) = -\sum_{i=1}^{n} \log\Big(\sum_{k=1}^{K} \pi_k\, \mathcal{N}(x_i;\, \mu_k, \sigma_k^2)\Big), \qquad \theta = (\pi_k, \mu_k, \sigma_k)_{k=1}^{K}.$$

All the trouble comes from the sum sitting *inside* the logarithm. The logarithm stops at the mixture and never reaches the individual Gaussian densities, where it would otherwise cancel the exponential and leave a simple quadratic. Compare this with the *complete-data* problem: if an oracle handed us every label z_i, the likelihood would separate cleanly by component, and maximum likelihood would reduce to estimators we already know. Each mixing weight π_k would simply be the fraction of points belonging to component k, just like the coin-bias estimator $\widehat{\theta} = n_H/n$ in Section 36.6, while μ_k and σ_k^2 would be the ordinary sample mean and variance of those points. Because the problem is hard only when the labels are missing, a natural strategy suggests itself: estimate the missing labels as probabilities, and then solve the easy complete-data problem.

What does such a probabilistic guess look like for a single point? Consider one observation at $x = 0.5$ under two equally weighted components, $\mathcal{N}(-1, 1)$ and $\mathcal{N}(1, 1)$. Their densities at $x = 0.5$ are proportional to $e^{-(1.5)^2/2} = e^{-1.125}$ and $e^{-(0.5)^2/2} = e^{-0.125}$, so Bayes' rule assigns the left component the posterior probability

$$\frac{e^{-1.125}}{e^{-1.125} + e^{-0.125}} = \frac{1}{1 + e} = 0.27.$$

Rather than forcing a hard choice, Bayes' rule splits the point 27% to 73% between the two components. Move the observation far to the right, say to $x = 3$, and the posterior becomes 99.8% certain it belongs to the right component; place it at the midpoint $x = 0$, and the split is an even 50%–50%. The soft guess is confident where the data are clear and appropriately undecided where the components overlap (Figure 36.34).

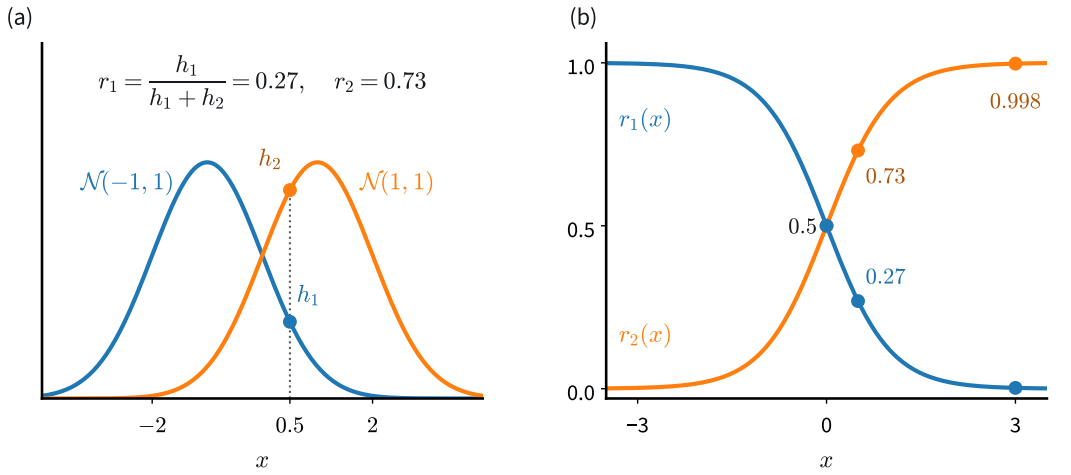


Figure 36.34: Soft assignment as a ratio of density heights, for two equally weighted components $\mathcal{N}(-1, 1)$ and $\mathcal{N}(1, 1)$. (a) At $x = 0.5$ the densities have heights h_1 and h_2, and the left component receives the share $r_1 = h_1/(h_1 + h_2) = 0.27$. (b) The two posterior probabilities $r_1(x)$ and $r_2(x)$: an even split at $x = 0$, 0.27 to 0.73 at $x = 0.5$, and 0.998 for the right component at $x = 3$.

36.7.2 The Evidence Lower Bound

To turn this intuitive strategy into an algorithm with a mathematical guarantee, we construct a lower bound on the log-likelihood in which the logarithm moves back inside the sum. Let $q(z)$ be *any* probability distribution over the latent variable, representing our soft guess. We multiply and divide inside the marginal likelihood by $q(z)$ and apply Jensen's inequality: because the logarithm is concave, $\log \mathbb{E}[\cdot] \geq \mathbb{E}[\log(\cdot)]$ (Section 35.3.3.1). This yields

$$\log p(x \mid \theta) = \log \sum_z q(z) \, \frac{p(x, z \mid \theta)}{q(z)} \;\geq\; \sum_z q(z) \log \frac{p(x, z \mid \theta)}{q(z)} \;=:\; \mathcal{L}(q, \theta). \quad (36.92)$$

Here the sums run over all z with $q(z) > 0$. If q happens to put zero probability on some z where the joint distribution is positive, the first equality simply becomes $\geq$, so the bound still holds. Notice that the marginal $p(x \mid \theta)$ plays the same role for the latent variable z that the evidence $P(X)$ (the denominator of Bayes' rule in Section 36.6.4) plays for model parameters: it is the probability of the observed data after summing out everything unobserved. For that reason it is also called the *evidence*, and the right-hand side $\mathcal{L}(q, \theta)$ is called the **evidence lower bound** (**ELBO**). Splitting the logarithm of the ratio separates the ELBO into two transparent pieces:

$$\mathcal{L}(q, \theta) = \mathbb{E}_q\big[\log p(x, z \mid \theta)\big] + H(q).$$

The first term is the expected *complete-data* log-likelihood, our easy objective averaged over the soft guess q. The second term is the entropy of the guess, which does not depend on θ at all. Even better, we can measure the exact gap between the ELBO and the true log-likelihood: it is a KL divergence, and both EM and variational inference rest on this single identity.

Proposition (the ELBO gap is a KL divergence). *For every θ and every distribution q that puts no mass outside the support of the posterior $p(z \mid x, \theta)$,*

$$\log p(x \mid \theta) = \mathcal{L}(q, \theta) + D_{\mathrm{KL}}\big(q(z) \,\|\, p(z \mid x, \theta)\big). \tag{36.93}$$

Hence $\mathcal{L}(q, \theta) \leq \log p(x \mid \theta)$, with equality if and only if q equals the posterior $p(z \mid x, \theta)$.

Proof. Substitute Bayes' rule, $p(z \mid x, \theta) = p(x, z \mid \theta)/p(x \mid \theta)$, directly into the definition of the KL divergence:

$$D_{\mathrm{KL}}\big(q \,\|\, p(z \mid x, \theta)\big) = \mathbb{E}_q\left[\log \frac{q(z)\, p(x \mid \theta)}{p(x, z \mid \theta)}\right] = \log p(x \mid \theta) - \mathcal{L}(q, \theta),$$

where $\log p(x \mid \theta)$ pulls straight out of the expectation because it does not depend on z. Rearranging gives Equation 36.93. By Gibbs' inequality (Section 36.4.2), the KL divergence is always nonnegative and vanishes if and only if its two arguments match. ■

Let us check this identity on our concrete example $x = 0.5$ from the previous subsection. There the true log-evidence is $\log p(x) = -1.42$. If we use the undecided guess $q = (\frac{1}{2}, \frac{1}{2})$, the ELBO evaluates to -1.54, and the missing 0.12 nats are the exact KL divergence from $(\frac{1}{2}, \frac{1}{2})$ to the posterior $(0.27, 0.73)$. In other words, the bound falls short of the log-evidence by the mismatch between our guess and the true posterior, and by nothing else. Figure 36.35 illustrates how this lower bound behaves as θ varies.

36.7.3 Expectation–Maximization

The ELBO identity turns our difficult optimization problem into **coordinate ascent** on $\mathcal{L}(q, \theta)$: maximize with respect to one argument while holding the other fixed, swap roles, and repeat. Given a dataset $X = \{x_1, \ldots, x_n\}$ of independent observations, the bound applies to each observation separately and sums across the dataset, with one guess q_i per data point. The **expectation–maximization** (EM) algorithm alternates between these two coordinates (Dempster et al. 1977):

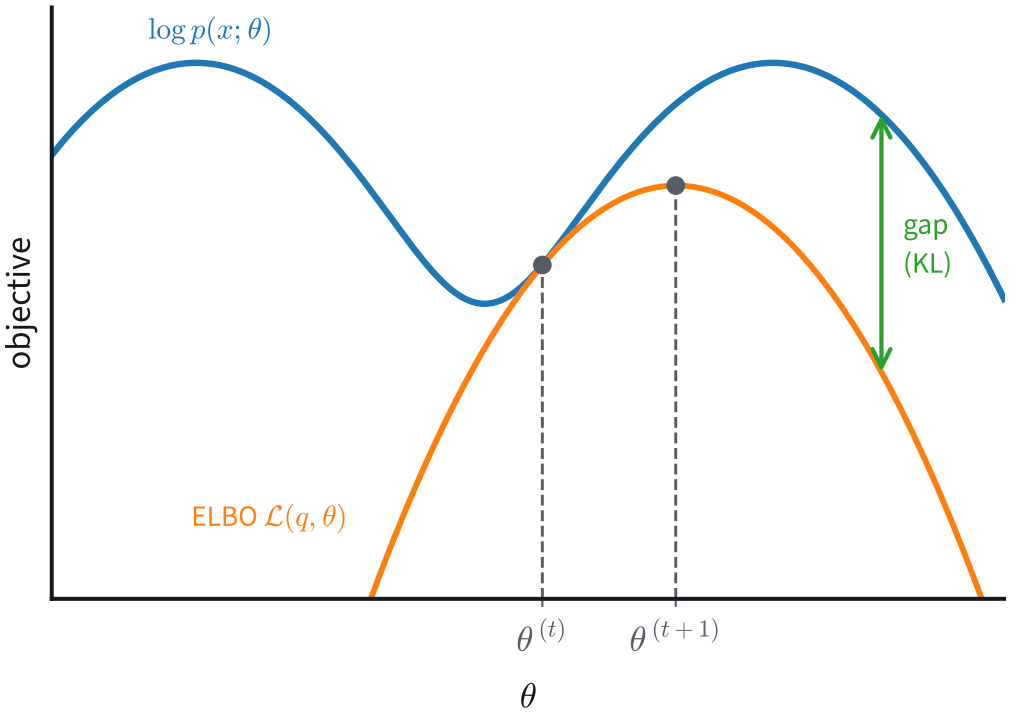


Figure 36.35: The evidence and its lower bound. For a fixed guess q, the ELBO $\mathcal{L}(q, \theta)$ lies below the evidence $\log p(x \mid \theta)$ at every θ, and the vertical gap between the curves is the KL divergence from q to the posterior $p(z \mid x, \theta)$. In EM, the E-step sets q to the posterior at the current iterate $\theta^{(t)}$, closing the gap so the bound touches the evidence there; the M-step then climbs the bound to its peak $\theta^{(t+1)}$. Because the bound touches the evidence at $\theta^{(t)}$ and never rises above it, the evidence at $\theta^{(t+1)}$ is guaranteed to be at least as high as at $\theta^{(t)}$.

- **E-step.** Hold $\theta^{(t)}$ fixed and maximize over the guesses q_i. By the equality condition in Equation 36.93, the optimal choice for every point is its exact posterior, $q_i^{(t)}(z) = p(z \mid x_i, \theta^{(t)})$. This drives every KL gap to zero and makes the bound *tight* at the current parameters: $\sum_i \mathcal{L}(q_i^{(t)}, \theta^{(t)}) = \log P(X \mid \theta^{(t)})$.
- **M-step.** Hold the guesses $q_i^{(t)}$ fixed and maximize over the parameters θ:

$$\theta^{(t+1)} = \operatorname*{argmax}_{\theta} \sum_i \mathbb{E}_{q_i^{(t)}}\big[\log p(x_i, z \mid \theta)\big],$$

  where the entropy terms $H(q_i^{(t)})$ drop out because they do not depend on θ. We are back to the easy complete-data objective, with soft probabilities standing in for the missing oracle labels.

Why can we trust this alternating procedure? Because the geometry in Figure 36.35 guarantees that every single step improves (or at worst preserves) the true log-likelihood.

Proposition (EM never decreases the likelihood). *Every EM iteration satisfies* $\log P(X \mid \theta^{(t+1)}) \geq \log P(X \mid \theta^{(t)})$.

Proof. Chain together three one-line facts:

$$\log P(X \mid \theta^{(t+1)}) \;\geq\; \sum_i \mathcal{L}(q_i^{(t)}, \theta^{(t+1)}) \;\geq\; \sum_i \mathcal{L}(q_i^{(t)}, \theta^{(t)}) \;=\; \log P(X \mid \theta^{(t)}).$$

The first inequality is the ELBO bound Equation 36.92 evaluated at the new parameters, the second holds because the M-step maximizes the bound over θ, and the final equality is the tightness secured by the E-step. ■

For our Gaussian mixture model, both steps come out in closed form. The E-step computes the **responsibilities**, the posterior probabilities that point i was generated by component k, just as we did by hand for $x = 0.5$:

$$r_{ik} = \frac{\pi_k \, \mathcal{N}(x_i;\, \mu_k, \sigma_k^2)}{\sum_j \pi_j \, \mathcal{N}(x_i;\, \mu_j, \sigma_j^2)}.$$

The M-step then refits each component using responsibility-weighted counting and averaging. Writing $n_k = \sum_i r_{ik}$ for the effective number of points assigned to component k, we obtain

$$\pi_k = \frac{n_k}{n}, \qquad \mu_k = \frac{1}{n_k} \sum_i r_{ik} \, x_i, \qquad \sigma_k^2 = \frac{1}{n_k} \sum_i r_{ik} \, (x_i - \mu_k)^2. \tag{36.94}$$

Notice how familiar these formulas are: they are the ordinary complete-data estimators (mixing fraction, sample mean, and sample variance), simply softened. Instead of casting one hard vote for a single component, each data point divides its vote across the components according to its responsibilities. In fact, if we force all components to share a single fixed

variance σ^2 and shrink $\sigma^2 \to 0$, the responsibilities harden into $0/1$ assignments to the nearest mean, and EM reduces directly to the K-means algorithm.

The entire algorithm takes only a dozen lines of code. Let us generate 300 points from $\mathcal{N}(-2, 0.8^2)$ and 200 from $\mathcal{N}(1.5, 0.6^2)$ and write a short function `em` for a one-dimensional mixture of K Gaussians that records the log-likelihood at every step alongside the final fitted parameters.

```
rng = np.random.default_rng()
x = np.concatenate([rng.normal(-2.0, 0.8, 300), rng.normal(1.5, 0.6, 200)])

def gauss(x, mu, sigma):
    return np.exp(-0.5 * ((x - mu) / sigma)**2) / (sigma * np.sqrt(2 * np.pi))

def em(x, pi, mu, sigma, steps):
    """EM for a 1-D Gaussian mixture: log-likelihood trace and final fit."""
    pi, mu, sigma = (np.array(v, dtype=float) for v in (pi, mu, sigma))
    lls = []
    for _ in range(steps):
        joint = pi * gauss(x[:, None], mu, sigma)  # (n, K)
        lls.append(np.log(joint.sum(axis=1)).sum())
        r = joint / joint.sum(axis=1, keepdims=True)  # E-step
        n_k = r.sum(axis=0)  # M-step
        pi, mu = n_k / len(x), (r * x[:, None]).sum(axis=0) / n_k
        sigma = np.sqrt((r * (x[:, None] - mu)**2).sum(axis=0) / n_k)
    return np.array(lls), pi, mu, sigma
```

Our proposition guarantees that the log-likelihood will never decrease, but it says nothing about how quickly EM climbs. Let us test it from a rather poor initial guess, two heavily overlapping unit-variance components centered at ± 0.5, run 30 iterations, and inspect the first six log-likelihood values alongside the final step.

```
lls, pi, mu, sigma = em(x, [0.5, 0.5], [-0.5, 0.5], [1, 1], 30)
print('steps 0-5:', lls[:6].round(1), ' step 29:', lls[-1].round(1))
print(f'pi = {pi.round(2)}, mu = {mu.round(2)}, sigma = {sigma.round(2)}')
```

```
steps 0-5: [-1298.1  -989.9  -942.   -871.   -848.6  -847. ]  step 29: -846.8
pi = [0.59 0.41], mu = [-2.04  1.54], sigma = [0.74 0.6 ]
```

The log-likelihood climbs by roughly 400 nats, rising from around $-1{,}280$ to around -860 (both figures vary by a few tens of nats with the random sample). Remarkably, almost the entire gain happens immediately: after just six steps the log-likelihood is already within about a nat of its final value, and the recovered weights, means, and scales land within about 0.2 of the true generating parameters $(0.6, 0.4)$, $(-2, 1.5)$, and $(0.8, 0.6)$.

Despite this clean convergence, EM comes with four limitations worth knowing. The first is harmless: permuting the labels of the K components produces $K!$ identical copies of every maximum. This *label switching* is the same non-identifiability we encounter when permuting the hidden units of a neural network, and it does not make any solution worse than another.

The second limitation is more consequential: EM stops at the first stationary point it hits, whether or not that point is a global maximum. A symmetric initialization illustrates this

sharply. Before running the next cell, pause and predict what EM will do if both components start with the exact same mean and scale.

```
lls2, pi2, mu2, sigma2 = em(x, [0.5, 0.5], [0, 0], [1, 1], 30)
print(f'log-likelihood {lls2[0]:.1f} -> {lls2[-1]:.1f}, '
      f'{lls[-1] - lls2[-1]:.1f} below the previous fit')
print(f'pi = {pi2.round(2)}, mu = {mu2.round(2)}, sigma = {sigma2.round(2)}')
print(f'mean of x = {x.mean():.2f}, standard deviation of x = {x.std():.2f}')
```

```
log-likelihood -1438.7 -> -1027.5, 180.7 below the previous fit
pi = [0.5 0.5], mu = [-0.59 -0.59], sigma = [1.89 1.89]
mean of x = -0.59, standard deviation of x = 1.89
```

The two components never break apart. Because they begin identically, they receive identical responsibilities on every single data point, so the M-step keeps them locked together. EM settles into the single-Gaussian fit, just the overall mean and standard deviation of the entire dataset, more than 100 nats below the two-cluster solution. Every fixed point of EM is a stationary point of the likelihood, not necessarily a maximum, which is why practitioners restart EM from several random initializations. Random initialization breaks this exact same symmetry among hidden units in a neural network layer (Section 4.4): units that start equal receive equal gradients and remain equal forever.

The third limitation is a quirk of continuous mixtures: the likelihood itself is unbounded. Place the second component directly on top of the first data point, $\mu_2 = x_1$, with weight $\pi_2 = 1/n$ and standard deviation ϵ. Its density at x_1 is $\frac{1}{n}(2\pi)^{-1/2}\epsilon^{-1}$, while the first component continues to cover the remaining $n-1$ points, giving

$$\ell(\theta) \le \log\left(n\sqrt{2\pi}\right) + \log\epsilon + \text{const},$$

where the constant gathers the contribution of the other $n-1$ points. As $\epsilon \to 0$, the negative log-likelihood plunges to $-\infty$: the global maximum-likelihood "solution" is simply an infinitely sharp spike perched on a single data point (Figure 36.36). Imposing a lower floor on the variances, or placing a prior over them, eliminates these degenerate spikes. Once the set of parameters at least as good as the starting point is compact, every limit point of EM is guaranteed to be a stationary point of the regularized problem (Wu 1983).

The fourth limitation is what leads beyond EM altogether. Our monotonicity proof depended on plugging the *exact* posterior into the E-step; whenever we must settle for an approximate E-step, that guarantee disappears. A *variational autoencoder* (Kingma and Welling 2014), for example, keeps the very same ELBO objective but replaces the exact per-point posterior with an encoder neural network $q_\phi(z \mid x)$ trained jointly with the decoder, because a deep nonlinear generator has no closed-form posterior. Instead of exact EM steps, it climbs the ELBO using stochastic gradients.

Here the bound outgrows the algorithm that introduced it. Nothing in the ELBO requires the hidden variable z to be a cluster label: if we put the model parameters themselves in place of z, the posterior that the E-step asks for is the posterior over parameters, the very object we were left wanting at the end of Section 36.6. That posterior is rarely available in closed form either, so Section 36.8 turns to the practical methods we use to approximate and sample from it.

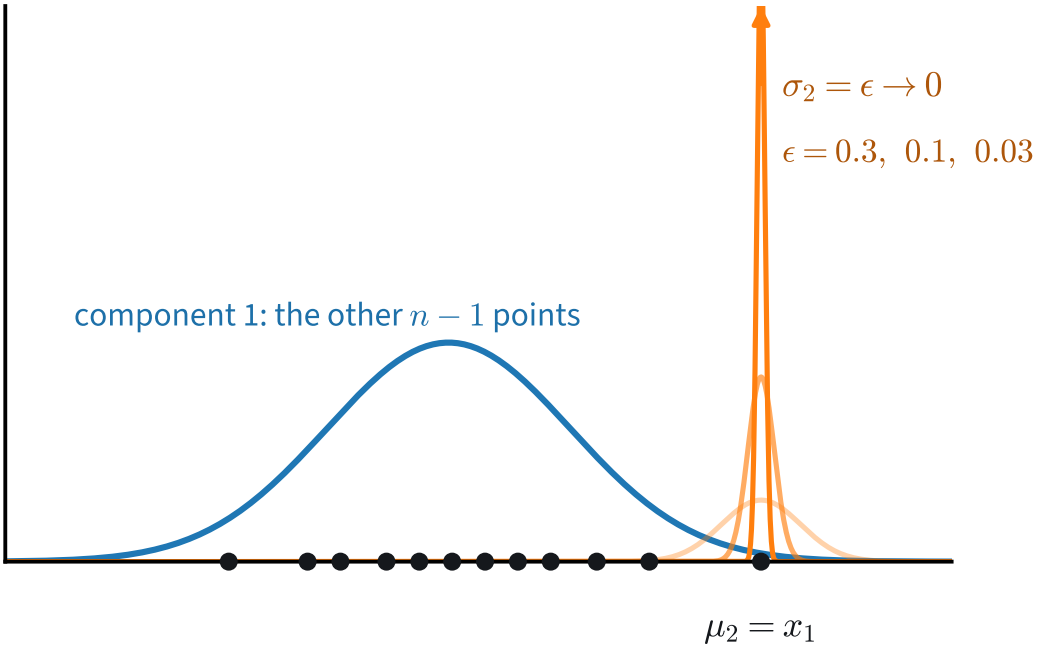


Figure 36.36: The degenerate maximum of a Gaussian mixture likelihood. The first component (blue) covers $n-1$ of the data points (dots). The second (orange) sits on the remaining point, $\mu_2 = x_1$, with standard deviation $\sigma_2 = \epsilon$. As ϵ shrinks from 0.3 to 0.1 to 0.03, its density at x_1 grows like $1/\epsilon$, and the likelihood grows with it without bound.

36.7.4 Summary

When a model includes hidden variables, its marginal likelihood traps a sum inside a logarithm, turning what would have been simple counting and averaging into an optimization problem with no closed form. Introducing a probabilistic guess q over the hidden variables restores the easy complete-data objective as a lower bound: the ELBO falls below the log-evidence by exactly the KL divergence from q to the true latent posterior. EM alternates between closing that gap (the E-step) and maximizing the resulting bound over the parameters (the M-step). Although every iteration is guaranteed never to decrease the likelihood, EM can stop at non-optimal stationary points, requires a variance floor or prior to prevent degenerate spikes in continuous mixtures, and needs an exact posterior to keep its monotonicity guarantee.

36.7.5 Exercises

1. code **EM speed and missing information.** ● Consider the mixture $p(x \mid \pi) = \pi f_0(x) + (1-\pi) f_1(x)$ with known components $f_0 = \mathcal{N}(\cdot\,; 0, 1)$ and $f_1 = \mathcal{N}(\cdot\,; \Delta, 1)$, in which only the mixing weight π is unknown.
   1. Show that EM updates the weight by $\pi^{(t+1)} = M(\pi^{(t)})$ with $M(\pi) = \frac{1}{n}\sum_i r_i(\pi)$ and responsibilities $r_i(\pi) = \pi f_0(x_i)/p(x_i \mid \pi)$.
   2. Near a fixed point $\hat\pi = M(\hat\pi)$, each iteration multiplies the error by approximately $M'(\hat\pi)$. Define the observed information $I_{\text{obs}} = -\frac{d^2}{d\pi^2}\sum_i \log p(x_i \mid \pi)$ at $\hat\pi$, and the complete-data information $I_{\text{com}} = n/(\hat\pi(1-\hat\pi))$ that the labels would carry if they were observed (compare Section 36.6). Show that
   $$M'(\hat\pi) = 1 - \frac{I_{\text{obs}}}{I_{\text{com}}}. \tag{36.95}$$
   Express both sides in terms of the responsibilities and use $\sum_i r_i(\hat\pi) = n\hat\pi$. Dempster et al. (1977) prove the general form of this result: the convergence rate of EM is governed by the fraction of information hidden by the latent variables.
   3. Draw $n = 10{,}000$ points with $\pi = 0.3$ for each $\Delta \in \{0.5, 1, 2, 4\}$, run EM from $\pi^{(0)} = 0.5$, and compare the measured ratio of successive errors against the prediction from (b). Taking $\hat\pi$ from a long run, how many iterations does each Δ need to bring the error below 10^{-6}?
   4. What does the ratio $I_{\text{obs}}/I_{\text{com}}$ imply about the standard error of $\hat\pi$, compared with a dataset of the same size whose labels were observed?
2. **The bound by hand.** Take the single observation $x = 0.5$ and the two equally weighted components $\mathcal{N}(-1, 1)$ and $\mathcal{N}(1, 1)$ from the text.
   1. Compute $\log p(x)$, along with the ELBO Equation 36.92 for the uniform guess $q = (\frac{1}{2}, \frac{1}{2})$ and for the posterior $q = (0.27, 0.73)$, verifying the values quoted in the text.
   2. Verify Equation 36.93 for the uniform guess by computing the KL divergence directly.
   3. A hard guess puts all of q on the right component. Compute its ELBO and its gap. An algorithm that alternates hard guesses with M-steps still never decreases the ELBO; why is its log-likelihood no longer guaranteed to increase?
3. **K-means as a limit.** Let all K components of a Gaussian mixture share equal weights and a single fixed variance σ^2, so that only the means are fitted.

1. Write the responsibilities r_{ik} as a softmax of $-(x_i - \mu_k)^2/(2\sigma^2)$ over k. What quantity plays the role of the temperature? Show that as $\sigma^2 \to 0$, the responsibility of the nearest mean tends to 1 whenever that nearest mean is unique.
2. Show that the M-step Equation 36.94 then sets each mean to the average of the points assigned to it, and that the two steps together never increase $\sum_i (x_i - \mu_{z_i})^2$, where z_i is the component assigned to point i.
3. In the symmetric start from the text, both means begin at the same point. What does K-means do from such a start, and what does this imply about how an implementation should break ties?

36.8 Bayesian Computation

Throughout the Basics part of this book, we trained each model to a single parameter vector, either by maximum likelihood or, with a regularizer, by the maximum a posteriori (MAP) estimate at the peak of the posterior. In Section 36.6, we went a step further and placed an error bar on that estimate. Even then, our predictions still came from that single vector, despite the fact that many other parameter vectors explain the training data almost as well. How much do those plausible models disagree, and what should we predict when they do?

Consider a concrete example: logistic regression with an intercept and a slope, fitted to twenty labeled points whose inputs lie between 0 and 4. At a test input of -3, well below the training data, the fitted MAP line assigns the label 1 a probability of about four in a thousand. Yet other parameter vectors that fit the twenty points nearly as well disagree there by orders of magnitude. If we average their predictions, weighting each vector by how plausible it is given the data, the resulting probability is almost five times larger. At an input of 2, right inside the training data, the single-vector and averaged predictions agree to within 0.01. For a rare event, such as a fraud alert or a medical diagnosis, a factor of five in the predicted probability can change the decision we make.

That weighted average is the Bayesian prediction, and computing it is a problem as old as Bayes' rule itself. The posterior distribution divides by the evidence, an integral over all parameter vectors that rarely has a closed form. In 1774, Laplace bypassed the integral by replacing the posterior of a proportion with the Gaussian that matches its peak. In 1953, Metropolis and his colleagues showed how to let a random walk visit states in proportion to a distribution whose normalizing constant is unknown (Metropolis et al. 1953). By the 1990s, researchers had also turned the integration problem into an optimization problem. One unifying trick runs through all of these methods: we can average over a distribution known only up to a normalizing constant because every method relies solely on ratios of densities, where the unknown constant cancels.

In this section, we develop four approaches by asking four questions in turn. First, can we approximate the posterior by the Gaussian that matches its peak? Second, can we draw from a simpler distribution and correct each draw? Third, can a random walk spend time in each region in proportion to its posterior mass? Fourth, can we optimize a simple distribution to be as close to the posterior as possible? Each of the first three methods remedies a limitation of the one before it, while the fourth replaces sampling with optimization. A low-dimensional posterior is the cleanest setting for a distribution known only up to a constant, though the same ideas reappear across the book; Section 36.8.5 maps out where each returns. All four methods aim to compute the **posterior predictive** distribution of a label $y_\star$ at a new input $\mathbf{x}_\star$,

$$p(y_\star \mid \mathbf{x}_\star, \mathcal{D}) = \int p(y_\star \mid \mathbf{x}_\star, \theta)\, p(\theta \mid \mathcal{D})\, d\theta, \tag{36.96}$$

which averages the predictions of every parameter vector, weighted by its posterior density given the data $\mathcal{D}$.

36.8.1 The Posterior of Bayesian Logistic Regression

To compare approximation methods fairly, we need a posterior that has no closed form yet can still be checked against the exact answer. Recall that a prior is *conjugate* to a likelihood when the posterior stays in the same family as the prior, just as a Beta prior on a coin stays Beta after any number of flips (Section 36.4). A Gaussian prior is not conjugate to the logistic likelihood: we can evaluate the unnormalized posterior at any parameter vector, but we cannot integrate it analytically. Because our running example has only two parameters, however, a fine two-dimensional grid still gives an exact reference against which every approximation can be judged.

36.8.1.1 The Model and Its Unnormalized Posterior

In one-dimensional logistic regression, each binary label $y_i \in \{0, 1\}$ has a probability determined by a linear function of the input x_i,

$$P(y_i = 1 \mid x_i, \theta) = \text{sigmoid}(\theta_0 + \theta_1 x_i) = \text{sigmoid}(\mathbf{z}_i^\top \theta), \qquad \text{sigmoid}(z) = \frac{1}{1 + e^{-z}},$$

where $\mathbf{z}_i = (1, x_i)^\top$. The predicted probability crosses $\frac{1}{2}$ where the linear function crosses zero, at $x = -\theta_0/\theta_1$, while the slope θ_1 controls how sharply the curve rises there. We place independent Gaussian priors $\theta_j \sim \mathcal{N}(0, \tau^2)$ on the intercept and slope, with prior standard deviation τ. By Bayes' rule, the posterior is proportional to the likelihood of the n labels times the prior:

$$p(\theta \mid \mathcal{D}) \propto \left[\prod_{i=1}^{n} \text{sigmoid}(\mathbf{z}_i^\top \theta)^{y_i} \big(1 - \text{sigmoid}(\mathbf{z}_i^\top \theta)\big)^{1 - y_i}\right] \exp\!\left(-\frac{\|\theta\|^2}{2\tau^2}\right). \tag{36.97}$$

The bracketed product is the likelihood and the exponential factor is the prior. The missing normalizing constant is the **evidence** $p(\mathcal{D}) = \int p(\mathcal{D}, \theta)\, d\theta$, an integral over all parameter vectors that has no closed form. We denote the right-hand side by $\tilde{p}(\theta) = p(\mathcal{D}, \theta)$, the *unnormalized posterior*, so that the normalized posterior is simply $\tilde{p}/p(\mathcal{D})$.

For our running dataset, we take $n = 20$ inputs drawn uniformly from $[0, 4]$ and generate labels from the true parameter vector $\theta^\star = (-3, 1.5)$, whose probability curve crosses $\frac{1}{2}$ at $x = 2$ and rises from $\text{sigmoid}(-3) \approx 0.05$ at $x = 0$ to $\text{sigmoid}(3) \approx 0.95$ at $x = 4$. We set the prior variance to $\tau^2 = 4$. With twenty data points, the posterior remains wide enough for every approximation effect in this section to stand out clearly. Rather than drawing a fresh dataset on each run, we fix one representative draw of twenty points (selected as the most typical among 1,500 simulated datasets on the posterior summaries measured below) so that we can quote concrete reference numbers. Every sampler we run still uses fresh

random numbers, so sampler results are reported as typical ranges. In code, `log_joint` evaluates $\log \tilde{p}$, using `logaddexp(0, z)` for $\log(1 + e^z)$ to prevent overflow at large logits.

```
rng = np.random.default_rng()
n, prior_var = 20, 4.0                          # prior_var is tau^2
theta_star = np.array([-3.0, 1.5])

def sigmoid(z):
    return np.exp(-np.logaddexp(0.0, -z))

x = np.array([0.1, 0.3, 0.3, 0.4, 0.8, 1.4, 1.5, 1.6, 1.8, 2.1,
              2.2, 2.3, 2.6, 2.7, 3.0, 3.1, 3.4, 3.5, 3.6, 3.7])
y = np.array([0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1])
Z = np.column_stack([np.ones(n), x])          # rows z_i = (1, x_i)

def log_joint(theta, Z=Z, y=y, prior_var=prior_var):
    """log p(D, theta) up to a constant; theta has final dimension 2."""
    logits = np.einsum('...d,nd->...n', theta, Z)
    log_lik = (y * logits - np.logaddexp(0.0, logits)).sum(axis=-1)
    return log_lik - 0.5 * (theta**2).sum(axis=-1) / prior_var

print(f'{n} points, {y.sum()} of them labeled 1')
```

```
20 points, 10 of them labeled 1
```

36.8.1.2 The Gradient and the Hessian

Several methods below rely on the slope and curvature of $\log \tilde{p}$, so it pays to derive them once up front. Differentiating each log-likelihood term $y_i \, \mathbf{z}_i^\top \theta - \log(1 + e^{\mathbf{z}_i^\top \theta})$ gives $(y_i - p_i)\, \mathbf{z}_i$, where $p_i = \text{sigmoid}(\mathbf{z}_i^\top \theta)$, while the Gaussian prior contributes $-\theta/\tau^2$. Stacking the input rows $\mathbf{z}_i^\top$ into the $n \times 2$ matrix $\mathbf{Z}$, we obtain the gradient $\nabla \log \tilde{p}(\theta) = \mathbf{Z}^\top(\mathbf{y} - \mathbf{p}) - \theta/\tau^2$. Its geometry is transparent: each training point nudges θ along $\mathbf{z}_i$ in proportion to its prediction residual $y_i - p_i$, and the prior gently pulls the parameters back toward the origin. Differentiating a second time yields the negative Hessian

$$\mathbf{H}(\theta) = -\nabla^2 \log \tilde{p}(\theta) = \mathbf{Z}^\top \operatorname{diag}(p_i(1 - p_i))\mathbf{Z} + \tau^{-2}\mathbf{I}. \tag{36.98}$$

Here the data term is a weighted Gram matrix of the inputs. Because each weight $p_i(1 - p_i) \le \frac{1}{4}$ peaks at $p_i = \frac{1}{2}$, the points the model is most uncertain about contribute the most curvature. Adding the prior term $\tau^{-2}\mathbf{I}$ lifts every eigenvalue by at least τ^{-2}, so $\mathbf{H}(\theta)$ is strictly positive definite everywhere. Thus the log posterior is strictly concave and has a unique maximum. Even when the training data are linearly separable, that maximum stays finite, because the log-likelihood is never positive while the quadratic prior penalty drops without bound in every direction.

```
def grad_log_joint(theta, Z=Z, y=y, prior_var=prior_var):
    """Gradient of log p(D, theta); one row per row of theta."""
    theta = np.atleast_2d(theta)
    return (y - sigmoid(theta @ Z.T)) @ Z - theta / prior_var
```

```
def neg_hessian(theta, Z=Z, y=y, prior_var=prior_var):
    """H(theta) = Z^T diag(p (1 - p)) Z + I / tau^2."""
    p = sigmoid(Z @ theta)
    return Z.T @ (Z * (p * (1 - p))[:, None]) + np.eye(2) / prior_var
```

With only two parameters, writing these derivatives by hand is painless. For larger models, we would implement only $\log \tilde{p}$ and let automatic differentiation supply the gradient and Hessian.

Notice that every method in this section accesses the posterior solely through $\tilde{p}$, whether via its values, its gradient, or its Hessian.

36.8.1.3 The Posterior Mode and the Laplace Approximation

Our first and cheapest approximation builds directly on an optimization we already know how to run. The peak of the posterior is the MAP estimate, the penalized fit from Section 36.6, and Newton's method, $\theta \leftarrow \theta + \mathbf{H}(\theta)^{-1} \nabla \log \tilde{p}(\theta)$, reaches it in just a few iterations. One extra Hessian evaluation then turns that point estimate into a Gaussian distribution.

Specifically, we fit a Gaussian centered at the mode θ_{MAP}. Expanding $\log \tilde{p}$ to second order around θ_{MAP}, with $\mathbf{H} = \mathbf{H}(\theta_{\text{MAP}})$, gives

$$\log \tilde{p}(\theta) \approx \log \tilde{p}(\theta_{\text{MAP}}) - \tfrac{1}{2}(\theta - \theta_{\text{MAP}})^{\top} \mathbf{H} (\theta - \theta_{\text{MAP}}). \tag{36.99}$$

There is no linear term because the gradient vanishes at the mode. Up to an additive constant, the right-hand side is simply the log density of a Gaussian with mean θ_{MAP} and covariance $\mathbf{H}^{-1}$. Exponentiating therefore gives the **Laplace approximation**,

$$p(\theta \mid \mathcal{D}) \approx \mathcal{N}(\theta_{\text{MAP}}, \mathbf{H}^{-1}). \tag{36.100}$$

For this approximation to be well defined, $\log \tilde{p}$ must be twice differentiable at an interior mode with a positive definite negative Hessian $\mathbf{H}$, both of which hold here. Intuitively, a sharp peak has large curvature and therefore produces a narrow Gaussian with small variance. The Gaussian matches the location and local curvature of the posterior's peak, and nothing else.

A one-dimensional example makes the behavior immediately visible. Consider the skewed posterior Gamma$(4, 1)$, whose unnormalized density is $\theta^3 e^{-\theta}$ for $\theta > 0$. Its log density, $3 \log \theta - \theta$, has derivative $3/\theta - 1$, which vanishes at the mode $\theta = 3$, and negative second derivative $3/\theta^2 = \frac{1}{3}$ there. The Laplace approximation is therefore $\mathcal{N}(3, 3)$, with standard deviation $\sqrt{3} \approx 1.73$. Because the exact posterior has mean 4 and variance 4, the Gaussian sits too far to the left, is too narrow, and even places $\Phi(-\sqrt{3}) \approx 4.2\%$ of its mass below $\theta = 0$, where the true density is zero (Figure 36.37).

Applying the same calculation to a coin that landed 9 heads and 4 tails under a uniform prior recovers the maximum likelihood estimate $9/13$ and its standard error 0.128 from Section 36.6.3.2. Under a flat prior, in fact, the Laplace approximation is simply the MLE paired with its asymptotic error bar (Exercise 3).

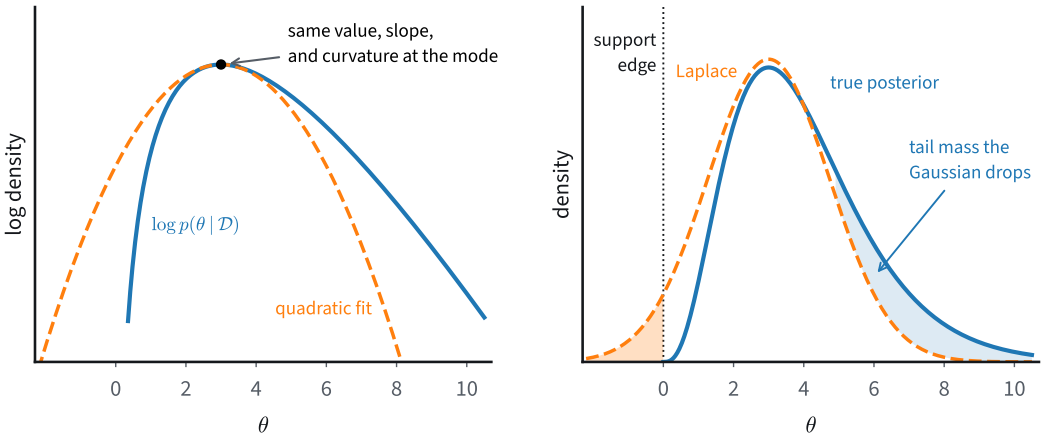


Figure 36.37: The Laplace approximation for a skewed one-dimensional posterior, the Gamma(4,1) density with mode 3. Left: the quadratic fit to the log density matches its value, slope, and curvature at the mode and nothing else. Right: after exponentiating, the Laplace Gaussian $\mathcal{N}(3, 3)$ matches the peak but misses the skew, placing 4.2% of its mass (shaded orange) below the support boundary $\theta = 0$ where the posterior is zero, while underestimating the right tail; the true mean is 4.

Notice that the unknown evidence never enters the calculation: $\log p(\mathcal{D})$ is an additive constant in $\log p(\theta \mid \mathcal{D})$, so it vanishes upon differentiation. Better still, integrating the exponential of the right-hand side of Equation 36.99 even yields an estimate of the evidence itself, $p(\mathcal{D}) \approx \tilde{p}(\theta_{\text{MAP}})(2\pi)^{d/2} \det(\mathbf{H})^{-1/2}$ in d dimensions (Exercise 6).

Because the Laplace approximation uses only local derivatives at a single peak, it is blind to skewness, heavy tails, parameter boundaries, and additional modes, and it centers the distribution at the mode rather than the mean. Scaling is also an issue: for a neural network with d parameters, $\mathbf{H}$ has d^2 entries, so practical applications typically keep only its diagonal or restrict the approximation to the last layer.

In code, ten Newton steps starting from the origin locate the mode, and a single matrix inverse yields the Laplace covariance.

```
theta_map = np.zeros(2)
for _ in range(10):                                  # Newton's method
    theta_map += np.linalg.solve(neg_hessian(theta_map),
                                 grad_log_joint(theta_map)[0])
laplace_cov = np.linalg.inv(neg_hessian(theta_map))
laplace_sd = np.sqrt(np.diag(laplace_cov))
laplace_rho = laplace_cov[0, 1] / (laplace_sd[0] * laplace_sd[1])
print('MAP               :', theta_map.round(3))
print('Laplace sd        :', laplace_sd.round(3))
print('Laplace correlation:', round(laplace_rho, 3))
```

```
MAP               : [-2.138  1.117]
Laplace sd        : [1.004 0.457]
Laplace correlation: -0.861
```

The posterior mode lies at $(-2.14,\ 1.12)$, a shallower line than the generating parameters $(-3, 1.5)$. The Laplace standard deviations are 1.00 for the intercept and 0.46 for the slope, placing the true values a little less than one standard deviation away in each coordinate. Notice also the strong negative correlation, -0.86: a steeper line must start lower to pass through the same cloud of points.

36.8.1.4 A Grid Reference

To see how accurate the Laplace approximation really is, we need the exact posterior as a benchmark. With only two parameters, we can evaluate $\log \tilde{p}$ on a fine $m \times m$ grid, subtract the maximum for numerical stability, exponentiate, and normalize the weights to sum to one. In d dimensions, such a lattice requires m^d evaluations, so grid integration is purely a low-dimensional reference. Even in two dimensions we must know where to place the box: we center the grid at θ_{MAP} and span seven Laplace standard deviations in each direction, leaving only a negligible tail outside.

```
k, m = 7, 241                    # half-width in Laplace sds, points per axis
grid_axes = [np.linspace(c - k * s, c + k * s, m)
             for c, s in zip(theta_map, laplace_sd)]
B0, B1 = np.meshgrid(*grid_axes, indexing='ij')
grid_theta = np.stack([B0.ravel(), B1.ravel()], axis=1)
grid_logp = log_joint(grid_theta)
grid_weight = np.exp(grid_logp - grid_logp.max())
grid_weight /= grid_weight.sum()
post_mean = grid_weight @ grid_theta
centered = grid_theta - post_mean
post_cov = (centered * grid_weight[:, None]).T @ centered
post_sd = np.sqrt(np.diag(post_cov))
post_rho = post_cov[0, 1] / (post_sd[0] * post_sd[1])
print('grid mean       :', post_mean.round(3), '  MAP:', theta_map.round(3))
print('grid sd         :', post_sd.round(3), '  Laplace:', laplace_sd.round(3))
print('grid correlation:', round(post_rho, 3))
print('(mean - MAP)/sd :', ((post_mean - theta_map) / post_sd).round(2))
```

```
grid mean       : [-2.347  1.237]   MAP: [-2.138  1.117]
grid sd         : [1.042 0.48 ]   Laplace: [1.004 0.457]
grid correlation: -0.858
(mean - MAP)/sd : [-0.2   0.25]
```

The grid exposes the skewness that Laplace missed. The posterior mean of the slope, 1.24, sits a quarter of a posterior standard deviation above the MAP estimate of 1.12, while the intercept's mean, -2.35, sits a fifth of a standard deviation below its MAP. The Laplace standard deviations are 4 to 5% too narrow, and the correlation agrees to two digits. Those width errors are modest; the more consequential discrepancy is that the posterior is skewed toward steeper lines, so its mean and its mode do not coincide.

36.8.1.5 The Plug-in and the Posterior Predictive

We can now answer our opening question directly. The *plug-in* prediction $\text{sigmoid}(\mathbf{z}_\star^\top \theta_{\text{MAP}})$ relies on the mode alone, whereas the posterior predictive Equation 36.96 averages $\text{sigmoid}(\mathbf{z}_\star^\top \theta)$ across the grid using the posterior weights. Let us compare both predictions

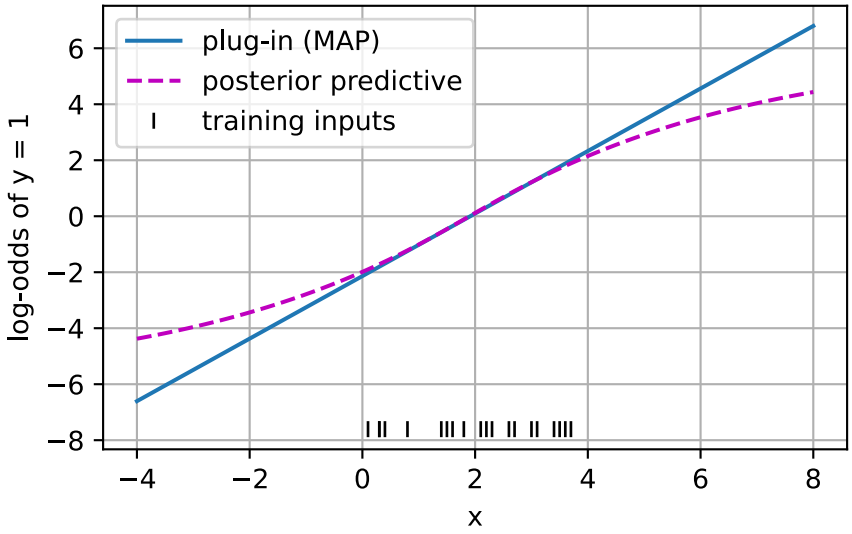
at three test inputs, $x_\star \in \{-3, 2, 7\}$, lying below, inside, and above the training data, alongside the interval containing the middle 90% of predictions across plausible parameter vectors.

```
x_new = np.array([-3.0, 2.0, 7.0])
Z_new = np.column_stack([np.ones(len(x_new)), x_new])
grid_probs = sigmoid(grid_theta @ Z_new.T)     # each grid vector's answer
plug_in = sigmoid(Z_new @ theta_map)
predictive = grid_weight @ grid_probs
for j, x0 in enumerate(x_new):
    order = np.argsort(grid_probs[:, j])
    cdf = np.cumsum(grid_weight[order])
    lo, hi = grid_probs[order[np.searchsorted(cdf, [0.05, 0.95])], j]
    print(f'x={x0:+.0f}: plug-in {plug_in[j]:.4f}, predictive '
          f'{predictive[j]:.4f}, middle 90%: [{lo:.2g}, {hi:.4f}]')
a = grid_theta @ Z_new[0]                      # logits at x = -3
mu = grid_weight @ a
s2 = grid_weight @ (a - mu)**2
print(f'x=-3, log scale: plug-in {np.log(plug_in[0]):.2f}, '
      f'mean logit {mu:.2f}, plus s^2/2 {mu + s2 / 2:.2f},')
print(f'                 log E[e^a] {np.log(grid_weight @ np.exp(a)):.2f}, '
      f'predictive {np.log(predictive[0]):.2f}')
```

```
x=-3: plug-in 0.0041, predictive 0.0187, middle 90%: [3.8e-05, 0.0887]
x=+2: plug-in 0.5238, predictive 0.5292, middle 90%: [0.32, 0.7364]
x=+7: plug-in 0.9966, predictive 0.9826, middle 90%: [0.92, 1.0000]
x=-3, log scale: plug-in -5.49, mean logit -6.06, plus s^2/2 -3.19,
                 log E[e^a] -3.66, predictive -3.98
```

Inside the training range at $x_\star = 2$, the two predictions (0.524 and 0.529) agree to within 0.01, and the plausible parameter vectors span a moderate range from 0.32 to 0.74. Far outside the data at $x_\star = -3$, the middle 90% of plausible predictions stretch across more than three orders of magnitude, from 4×10^{-5} to 0.09, and the posterior predictive (0.019) is 4.6 times the plug-in value (0.0041). At $x_\star = 7$, the same amplification occurs for the probability of label 0 (0.017 versus 0.003). Because probabilities this close to 0 or 1 are easiest to compare on the log-odds scale $\log \frac{P}{1-P}$, we plot both curves across a wider range with the training inputs marked along the bottom.

```
def log_odds(p):
    return np.log(p) - np.log1p(-p)

xs = np.linspace(-4, 8, 121)
Z_xs = np.column_stack([np.ones_like(xs), xs])
plug_xs = Z_xs @ theta_map                      # log-odds of the plug-in
pred_xs = log_odds(grid_weight @ sigmoid(grid_theta @ Z_xs.T))
rug = np.full(n, min(plug_xs.min(), pred_xs.min()) - 1)
d2l.plot([xs, xs, x], [plug_xs, pred_xs, rug],
         xlabel='x', ylabel='log-odds of y = 1', fmts=('-', 'm--', 'k|'),
         legend=['plug-in (MAP)', 'posterior predictive', 'training inputs'],
         figsize=(5, 3))
```



On the log-odds scale, the plug-in prediction is simply the straight line $\theta_0 + \theta_1 x$. Inside $[0, 4]$, the posterior predictive hugs that line; outside the training data, it bends back toward zero. Two competing effects create this gap. First, the posterior mean has a steeper slope than the MAP estimate, which pushes extreme predictions *away* from $\frac{1}{2}$. Second, averaging the sigmoid over a spread of logits pulls predictions *toward* $\frac{1}{2}$: far from the data, plausible parameter vectors with shallower slopes produce much larger probabilities than the typical vector, and those larger probabilities dominate the average.

Which effect wins, and by how much? A quick back-of-the-envelope calculation estimates the size of the averaging effect. Where the probability is small, $\text{sigmoid}(a) \approx e^a$, so the predictive probability is roughly the posterior average of e^a for the logit $a = \mathbf{z}_\star^\top \theta$. If a were Gaussian with mean μ and variance s^2, its moment generating function would give $\mathbb{E}[e^a] = e^{\mu + s^2/2}$, raising the log-probability by $s^2/2$ above the value at the mean logit. At $x_\star = -3$, the grid gives $s^2/2 = 2.87$, which would predict a factor of $e^{2.87} \approx 18$. Yet the actual factor is 4.6, a rise of only 1.51 on the log scale. Where did the missing 1.36 nats go? The last two lines of the code above walk from the plug-in to the posterior predictive one step at a time:

step at $x_\star = -3$	log-probability	change
plug-in: the logit at the mode	-5.49	
the posterior mean of the logit	-6.06	-0.57
add $s^2/2$, as for a Gaussian logit	-3.19	$+2.87$
average e^a over the logit's actual distribution	-3.66	-0.47
average $\text{sigmoid}(a)$ instead of e^a: the predictive	-3.98	-0.32

Three downward corrections account for the entire difference between 2.87 and 1.51. First, the $s^2/2$ boost starts from the mean logit rather than the plug-in logit, and the steeper posterior mean sits 0.57 lower. Second, the logit is skewed rather than Gaussian: it has a long

tail toward very negative values and a shorter tail toward zero, where the average of e^a gets almost all of its mass, trimming another 0.47. Third, the vectors that dominate the average have probabilities at $x_\star = -3$ that are no longer tiny, so the sigmoid bends below e^a and subtracts the final 0.32. Altogether, $2.87 - 0.57 - 0.47 - 0.32 = 1.51$.

The Gaussian calculation is therefore a useful guide rather than an exact law: it gets the direction, the rough scale, and the asymptotic dependence on $x_\star$ right. Specifically, the logit variance $\mathrm{Var}(\theta_0) + 2x_\star \mathrm{Cov}(\theta_0, \theta_1) + x_\star^2 \mathrm{Var}(\theta_1)$ grows quadratically with distance from the data, whereas the mean shift grows only linearly. Far from the training inputs, the variance term inevitably wins, while inside the data both effects remain negligible (Exercise 4).

Notice that the larger predictive probability is not necessarily closer to the data-generating truth. The true line gives $\mathrm{sigmoid}(-3 - 1.5 \cdot 3) = \mathrm{sigmoid}(-7.5) \approx 0.0006$ at $x_\star = -3$, which lies below both the plug-in and the posterior predictive because twenty points cannot pin down the slope and the prior shrinks it toward zero. That true probability does fall comfortably inside the 90% posterior interval, as expected for a typical dataset. The posterior predictive does not claim to know the true parameter; rather, it gives the exact probability of $y_\star = 1$ averaged over all parameter vectors consistent with the prior and the twenty observations. The rest of this section asks how to compute such averages when the parameter space is too large for a grid.

36.8.2 Rejection and Importance Sampling

The Laplace approximation is fast, but it centers on the mode and ignores skewness. If we could draw samples directly from the exact posterior, their empirical average would estimate any posterior expectation cleanly, capturing skewness and tails automatically. The two methods in this section turn draws from a tractable *proposal* distribution q into exact or weighted draws from the posterior, relying only on the ratio $\tilde{p}/q$.

36.8.2.1 Rejection Sampling

The simplest exact sampler draws candidates from q, keeps some, and discards the rest. Choose a proposal q that is easy to sample and evaluate, along with a constant M such that $M q(\theta) \geq \tilde{p}(\theta)$ for every θ, forming an envelope above the unnormalized posterior. For each draw $\theta \sim q$, we keep it with probability

$$a(\theta) = \frac{\tilde{p}(\theta)}{M\, q(\theta)}. \tag{36.101}$$

Geometrically, the acceptance probability is the ratio of the target height to the envelope height at the proposed point. Every accepted draw is an exact sample from $p(\theta \mid \mathcal{D})$, and the overall acceptance rate is $p(\mathcal{D})/M$. The verification takes two lines: a proposal lands in an infinitesimal region $d\theta$ around θ with probability $q(\theta)\, d\theta$ and is accepted with probability $a(\theta)$, so the joint probability of landing there and being kept is $\tilde{p}(\theta)\, d\theta / M$. Integrating over all θ yields the marginal acceptance probability $\int \tilde{p}/M = p(\mathcal{D})/M$, and dividing the joint density by this marginal gives the conditional density of a kept draw, $\tilde{p}(\theta)/p(\mathcal{D}) = p(\theta \mid \mathcal{D})$.

Crucially, the unknown evidence appears only in the acceptance rate, never in the accept-or-reject rule itself. If we choose the prior as the proposal q, then $\tilde{p}/q$ is simply the likelihood

and M is its maximum value: we draw parameter vectors from the prior and keep each with probability equal to its likelihood divided by the maximum likelihood.

To see how quickly rejection sampling degrades as data accumulate, consider the coin with 9 heads and 4 tails under a uniform prior. The likelihood $\theta^9(1-\theta)^4$ peaks at $\theta = 9/13$ with height $M = (9/13)^9(4/13)^4 \approx 3.3 \times 10^{-4}$, while the evidence is $p(\mathcal{D}) = \int_0^1 \theta^9(1-\theta)^4\,d\theta = 9!\,4!/14! \approx 1.0 \times 10^{-4}$, giving an acceptance rate of 0.305. Multiply the data by ten (90 heads and 40 tails) and the acceptance rate falls to 0.10; multiply by a hundred and it drops to 0.03. Because the posterior width shrinks like $1/\sqrt{n}$ while the prior stays fixed, the acceptance rate decays as $1/\sqrt{n}$ in one dimension and as $n^{-d/2}$ in d dimensions. A quick simulation with 100,000 draws from the uniform prior confirms the calculation.

```
N = 100_000
theta_coin = rng.random(N)                    # draws from the uniform prior
lik = theta_coin**9 * (1 - theta_coin)**4     # likelihood of 9 heads, 4 tails
M = (9 / 13)**9 * (4 / 13)**4                 # its maximum, at theta = 9/13
kept = theta_coin[rng.random(N) < lik / M]
print(f'acceptance {len(kept) / N:.3f}, mean {kept.mean():.3f}, '
      f'sd {kept.std():.3f}')
```

```
acceptance 0.304, mean 0.666, sd 0.118
```

Two serious obstacles limit rejection sampling in practice. First, we must know a valid upper bound M, which is rarely available outside simple models. Second, even with a fixed number of observations per parameter, the acceptance rate decays at least exponentially with the dimension d.

36.8.2.2 Importance Weights

Rejection sampling throws away most of the draws it generates. Rather than using the ratio $w = \tilde{p}/q$ to flip a coin and discard candidates, we can keep every draw and weight it by w. To see why this works, express any posterior expectation as a ratio of two integrals and multiply and divide by q inside both:

$$\begin{aligned}\mathbb{E}_{p(\cdot\mid\mathcal{D})}[h(\theta)] &= \frac{\int h\,\tilde{p}\,d\theta}{\int \tilde{p}\,d\theta} = \frac{\int h\,\frac{\tilde{p}}{q}\,q\,d\theta}{\int \frac{\tilde{p}}{q}\,q\,d\theta} \\ &= \frac{\mathbb{E}_q[w(\theta)h(\theta)]}{\mathbb{E}_q[w(\theta)]}, \qquad w(\theta) = \frac{\tilde{p}(\theta)}{q(\theta)}.\end{aligned} \tag{36.102}$$

The denominator $\mathbb{E}_q[w]$ equals the unknown evidence $p(\mathcal{D}) = \int \tilde{p}$, and we never need to compute it explicitly because the same normalizing constant scales both numerator and denominator. Replacing the two expectations with sample averages over draws $\theta_1, \dots, \theta_N \sim q$ gives **self-normalized importance sampling**, $\sum_s w_s h(\theta_s) / \sum_s w_s$. By the law of large numbers, this ratio converges to the exact posterior expectation as $N \to \infty$, though at finite N it carries a small bias because it is a ratio of two random averages. Since products of many likelihood terms quickly underflow in floating point, we always compute log-weights and subtract the largest log-weight before exponentiating, just as in the log-sum-exp trick. When q is the prior, the importance weight reduces to the likelihood: every prior draw is weighted by how well it explains the observed data.

36.8.2.3 The Effective Sample Size

Keeping every draw does not mean every draw helps equally. A weighted sample in which a single draw carries nearly all the weight is effectively worth just one draw, no matter how large N is. To quantify this, we define the **effective sample size** (ESS) from the normalized weights $\bar{w}_s = w_s / \sum_r w_r$:

$$N_{\text{eff}} = \frac{1}{\sum_{s=1}^{N} \bar{w}_s^2}. \tag{36.103}$$

When all weights are equal ($\bar{w}_s = 1/N$), $N_{\text{eff}} = N$; when one draw takes all the weight, $N_{\text{eff}} = 1$. Roughly speaking, the ESS estimates how many independent draws from the exact posterior the weighted sample is worth. As we will soon see, however, it can still mislead in two important ways.

Proposition. As $N \to \infty$,

$$\frac{N_{\text{eff}}}{N} \to \frac{(\mathbb{E}_q[w])^2}{\mathbb{E}_q[w^2]} = \frac{1}{1 + \chi^2(p \,\|\, q)}, \qquad \chi^2(p \,\|\, q) = \int \frac{p^2}{q}\, d\theta - 1, \tag{36.104}$$

where p is the normalized posterior. For the same proposal q, this limiting efficiency is always at least the acceptance rate $p(\mathcal{D})/M$ of rejection sampling.

Proof. Dividing numerator and denominator of Equation 36.103 by N^2 gives $N_{\text{eff}}/N = (\frac{1}{N}\sum_s w_s)^2 / (\frac{1}{N}\sum_s w_s^2)$, and the law of large numbers applies to both averages. Substituting $w = p(\mathcal{D})\, p/q$ yields $\mathbb{E}_q[w^2] = p(\mathcal{D})^2 \int p^2/q$ and $\mathbb{E}_q[w] = p(\mathcal{D})$, giving the χ^2-divergence expression. Finally, since $w \le M$, we have $\mathbb{E}_q[w^2] \le M\, \mathbb{E}_q[w]$, so the limit is at least $\mathbb{E}_q[w]/M = p(\mathcal{D})/M$. ■

The ESS is therefore an empirical measure of how closely q matches p. In the large-N limit, importance sampling retains at least as many effective draws as rejection sampling accepts, without ever needing an envelope bound M. Returning to the coin with a uniform prior proposal, $\mathbb{E}_q[w^2] = \int_0^1 \theta^{18}(1-\theta)^8\, d\theta$, which gives a limiting ESS fraction of 0.42 compared with the rejection acceptance rate of 0.305. We can reuse the exact same draws from the rejection cell and simply reweight them instead of discarding two-thirds of them.

```
w = lik / lik.sum()                          # normalized weights
post_coin = w @ theta_coin
print(f'ESS fraction {1 / (w @ w) / N:.3f}, mean {post_coin:.3f}, '
      f'sd {np.sqrt(w @ (theta_coin - post_coin)**2):.3f}')
```

```
ESS fraction 0.420, mean 0.666, sd 0.118
```

The limiting formula also exposes how sharply a poorly scaled proposal can fail. For a one-dimensional Gaussian target with standard deviation s_p and a centered Gaussian proposal with standard deviation $s_q = r s_p$, the limiting ESS fraction is $\sqrt{2r^2 - 1}/r^2$ (Exercise 5). That curve equals 0.83 at $r = 0.8$, peaks at 1 when $r = 1$, and declines gently to 0.66 at $r = 2$ and 0.46 at $r = 3$. For $r \le 1/\sqrt{2}$, however, it collapses abruptly to zero because

$\mathbb{E}_q[w^2] = \infty$. A proposal that is too narrow fails far more catastrophically than one that is too wide.

Two conditions guarantee that the importance identity Equation 36.102 holds: $q(\theta) > 0$ everywhere $\tilde{p}(\theta) > 0$, and $\mathbb{E}_q|wh| < \infty$. For a central limit theorem, and hence $O(1/\sqrt{N})$ error bars, we additionally need $\mathbb{E}_q[w^2]$ and $\mathbb{E}_q[w^2h^2]$ to be finite. Even when those conditions hold, the finite-sample ESS can be dangerously optimistic, as Figure 36.38 illustrates.

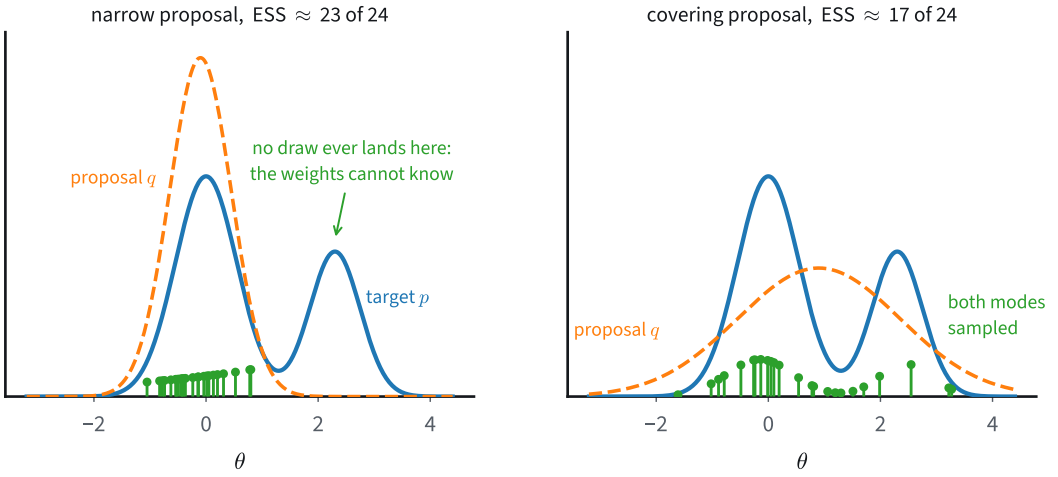


Figure 36.38: Importance sampling of a two-mode target using two different proposals; stems show the normalized weights of 24 draws. Left: a narrow proposal never visits the right-hand mode, so its weights look nearly uniform (ESS 23 of 24) even though its estimate is badly wrong. Right: a broader proposal covers both modes; its weights are less uniform (ESS 17 of 24), yet its estimate is far more accurate. The ESS measures weight inequality, not whether the proposal found all of the posterior mass.

36.8.2.4 Proposals for the Logistic Posterior

How should we choose a proposal for our logistic regression posterior? Let us compare two Gaussians, the prior with 20,000 draws and the Laplace approximation with 5,000 draws, by recording the ESS as a fraction of the draws, the largest normalized weight, and the error in the estimated posterior mean relative to the grid reference, measured in units of posterior standard deviations.

```
def gaussian_logpdf(samples, mean, cov):
    diff = samples - mean
    _, logdet = np.linalg.slogdet(cov)
    maha = np.einsum('ni,ij,nj->n', diff, np.linalg.inv(cov), diff)
    return -0.5 * (len(mean) * np.log(2 * np.pi) + logdet + maha)

def importance(log_target, mean, cov, N):
    """Self-normalized importance sampling from a Gaussian proposal."""
    samples = rng.multivariate_normal(mean, cov, N)
    log_w = log_target(samples) - gaussian_logpdf(samples, mean, cov)
```

```
    w = np.exp(log_w - log_w.max())
    w /= w.sum()
    return w @ samples, 1 / (w @ w), w.max()

for name, mean, cov, N in [('prior', np.zeros(2), prior_var * np.eye(2), 20000),
                           ('Laplace', theta_map, laplace_cov, 5000)]:
    estimate, ess, w_max = importance(log_joint, mean, cov, N)
    error = np.abs(estimate - post_mean) / post_sd
    print(f'{name:8s}: ESS {ess / N:.1%} of {N}, largest weight '
          f'{w_max:.4f}, error in sds {error.round(3)}')
```

```
prior   : ESS 4.2% of 20000, largest weight 0.0025, error in sds [0.013 0.001]
Laplace : ESS 85.5% of 5000, largest weight 0.0030, error in sds [0.006 0.003]
```

Using the prior as the proposal yields an ESS of only about 4% of the draws, because most prior samples land where the likelihood is negligible. Aiming the proposal at the posterior with the Laplace Gaussian raises the ESS dramatically: typically 80% to 90% of the draws are effective, although an occasional run falls far lower. Both proposals usually estimate the posterior mean within a few hundredths of a posterior standard deviation of the grid reference, and rarely err by more than 0.15.

Yet the Laplace proposal hides a subtle defect that a typical ESS printout never reveals. Far from the origin, the posterior's tails are governed by the prior: along any ray $\theta = t\mathbf{v}$, the log-likelihood decays at most linearly in t (Exercise 8), whereas the log prior decays quadratically as $-t^2/(2\tau^2)$. Consequently, for a Gaussian proposal $\mathcal{N}(\mathbf{m}, \mathbf{S})$, the integral $\mathbb{E}_q[w^2] = \int \tilde{p}^2/q$ diverges whenever any eigenvalue of $\mathbf{S}$ falls below $\tau^2/2 = 2$. Here both eigenvalues of the Laplace covariance (roughly 1.17 and 0.05) lie well below 2, so the importance weights have infinite variance in *every* direction, not just along a single thin axis.

Why did our 5,000-draw run look so healthy? Because the region where the weights blow up sits many Laplace standard deviations away, so most runs of 5,000 draws never stumble into it. In a few runs out of a hundred, however, a single distant draw captures a huge share of the total weight and the ESS plummets. A single run cannot expose this instability; only repeated runs or a tail analysis can. Two simple fixes restore finite weight variance: replacing the Gaussian with a heavier-tailed Student-t proposal (Exercise 8), or using a *defensive mixture* such as $q = 0.9\,\mathcal{N}(\theta_{\text{MAP}}, \mathbf{H}^{-1}) + 0.1$ prior. Because the mixture density is always at least one-tenth of the prior density, every weight is bounded above by ten times the likelihood, which never exceeds 1.

Importance sampling cancels the unknown evidence automatically, but it still demands a single global proposal q that covers the entire posterior. In high dimensions, no simple global distribution can do that.

36.8.3 Markov Chain Monte Carlo

Rather than searching for one global proposal that blankets the whole posterior, we can explore the parameter space through local moves, designing a random walk that spends time in each region in proportion to its posterior mass. Such a walk is a *Markov chain*: its next state depends only on its current state through a *transition rule* $T(\theta' \mid \theta)$, the probability of moving from θ to θ'. A distribution π is *stationary* for the chain if drawing the current state from π leaves the next state distributed as π as well:

$$\sum_{\theta} \pi(\theta)\, T(\theta' \mid \theta) = \pi(\theta') \quad \text{for every } \theta', \tag{36.105}$$

where the sum becomes an integral for continuous parameters. Our goal is to construct a simple transition rule whose stationary distribution is the posterior.

36.8.3.1 The Metropolis Rule

The Metropolis algorithm builds such a chain from any symmetric proposal $g(\theta' \mid \theta) = g(\theta \mid \theta')$, such as a local Gaussian step $\theta' = \theta + \mathbf{L}\epsilon$. From the current state θ, we propose a candidate $\theta' \sim g(\cdot \mid \theta)$. If the candidate has higher or equal posterior density, $\tilde{p}(\theta') \geq \tilde{p}(\theta)$, we always move to θ'. If it moves downhill, we accept it with probability $\tilde{p}(\theta')/\tilde{p}(\theta)$ and otherwise remain at θ. Crucially, a rejected proposal repeats the current state, and that repeat counts as the next state in the chain (Figure 36.39).

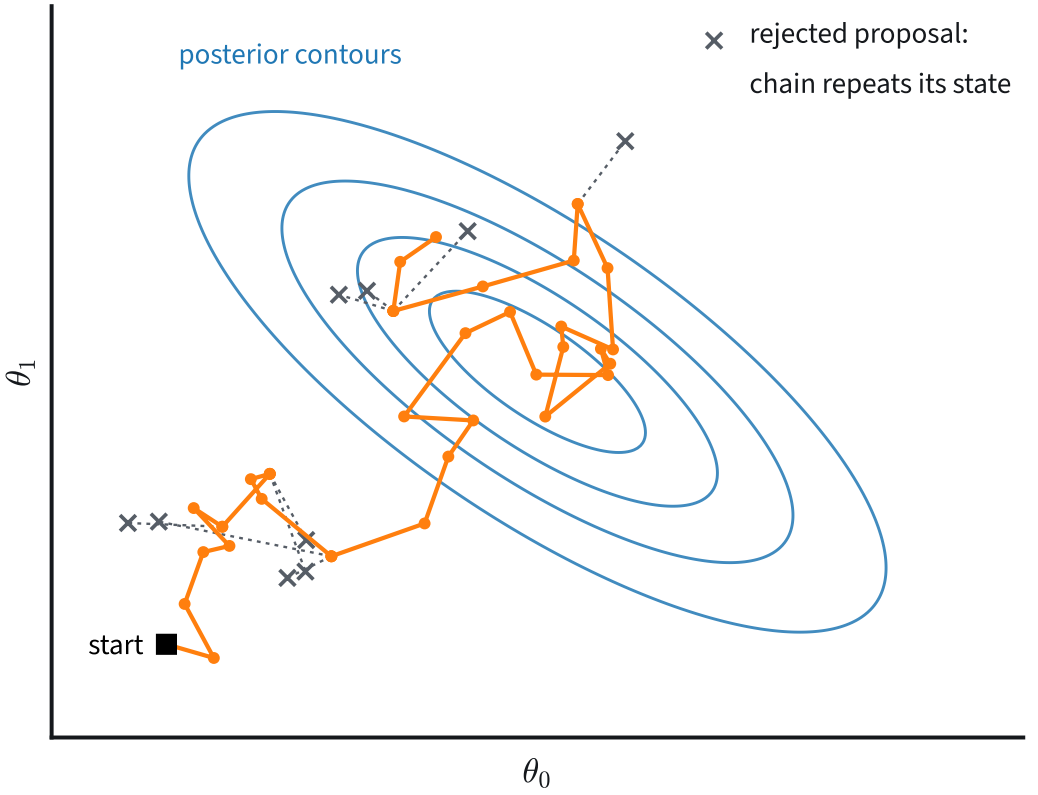


Figure 36.39: A Metropolis random walk on posterior contours. From the starting square, each step proposes a local move: uphill moves are always accepted (path), downhill moves are accepted probabilistically, and rejected proposals (crosses) leave the chain in place for another step. In the long run, the fraction of time spent in any region matches its posterior mass.

Hastings (Hastings 1970) generalized the rule to asymmetric proposals by multiplying the

density ratio by a correction factor, $g(\theta \mid \theta')/g(\theta' \mid \theta)$, that compensates for any directional bias in the proposal. A candidate is then accepted with probability

$$\alpha(\theta, \theta') = \min\left(1, \frac{\tilde{p}(\theta')\, g(\theta \mid \theta')}{\tilde{p}(\theta)\, g(\theta' \mid \theta)}\right). \tag{36.106}$$

Moves toward states that the proposal rarely visits are accepted more readily, and when g is symmetric the correction factor equals 1. Once again, the unknown evidence cancels in the ratio of $\tilde{p}$ values.

The smallest possible chain illustrates why counting rejections is essential. Consider two states with target distribution $\pi = (\frac{1}{4}, \frac{3}{4})$, and a symmetric proposal that always suggests the *other* state. From state 1, the target ratio is 3, so the chain always jumps to state 2. From state 2, the ratio is $\frac{1}{3}$, so it jumps to state 1 with probability $\frac{1}{3}$ and stays at state 2 with probability $\frac{2}{3}$. Writing the transition matrix with rows for the current state, we can check stationarity directly:

$$\mathbf{T} = \begin{pmatrix} 0 & 1 \\ \frac{1}{3} & \frac{2}{3} \end{pmatrix}, \qquad \left(\tfrac{1}{4}, \tfrac{3}{4}\right)\mathbf{T} = \left(\tfrac{3}{4}\cdot\tfrac{1}{3},\ \tfrac{1}{4} + \tfrac{3}{4}\cdot\tfrac{2}{3}\right) = \left(\tfrac{1}{4}, \tfrac{3}{4}\right).$$

If we discarded rejections and recorded only accepted moves, the sequence would simply alternate $2, 1, 2, 1$, yielding $(\frac{1}{2}, \frac{1}{2})$ instead of $(\frac{1}{4}, \frac{3}{4})$. It is precisely the repeated visits to state 2 after rejections that give state 2 its proper weight (Figure 36.40).

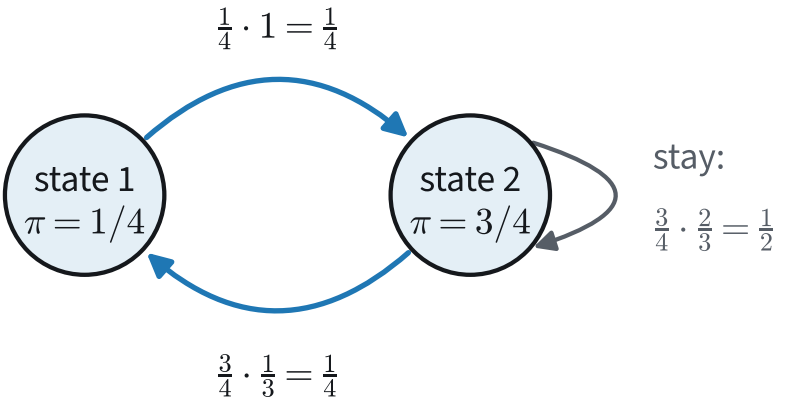


Figure 36.40: The two-state Metropolis chain with target $\pi = (1/4, 3/4)$. The probability flow from state 1 to state 2 is $\frac{1}{4}\cdot 1$ and the return flow is $\frac{3}{4}\cdot\frac{1}{3}$; both equal $1/4$, so the occupation $(1/4, 3/4)$ is preserved at every step. The self-loop represents rejected moves from state 2; recording only accepted moves would give $(1/2, 1/2)$.

36.8.3.2 Detailed Balance and Stationarity

Figure 36.40 captures the core mechanism behind Metropolis–Hastings: in one step, the probability mass flowing from state 1 to state 2 equals the probability mass flowing back,

so neither state gains nor loses probability. When this pairwise equilibrium holds for every pair of states, the target distribution is guaranteed to be stationary. If you remember one theoretical result from this section, make it this one.

Proposition. If a transition rule satisfies *detailed balance*,

$$\pi(\theta)\, T(\theta' \mid \theta) = \pi(\theta')\, T(\theta \mid \theta') \quad \text{for all } \theta, \theta', \tag{36.107}$$

then π is stationary for the chain. Moreover, the Metropolis–Hastings chain satisfies detailed balance with $\pi \propto \tilde{p}$.

Proof. Summing both sides of Equation 36.107 over θ yields the left-hand side of Equation 36.105, while the right-hand side simplifies to $\pi(\theta') \sum_{\theta} T(\theta \mid \theta') = \pi(\theta')$ because the transition probabilities from θ' sum to 1. For the second claim, detailed balance is automatic when $\theta' = \theta$. When $\theta' \neq \theta$, a move occurs if and only if θ' is proposed and accepted, so

$$\begin{aligned} \pi(\theta)\, T(\theta' \mid \theta) &= \pi(\theta)\, g(\theta' \mid \theta)\, \alpha(\theta, \theta') \\ &= \min\{\pi(\theta)\, g(\theta' \mid \theta),\ \pi(\theta')\, g(\theta \mid \theta')\}, \end{aligned}$$

which is visibly symmetric in θ and θ'. Because the normalizing constant of π cancels inside α, we can substitute $\tilde{p}$ for π. $\blacksquare$

The key algebraic step is pulling πg inside the minimum, which reveals the symmetry at a glance. On a continuous state space, the point mass of rejected moves requires slightly more care (Exercise 1).

Stationarity alone does not guarantee that a chain converges to π from an arbitrary starting point. The chain must also be able to reach every region of positive posterior mass, and it must not cycle deterministically between regions. Under these two mild conditions, the long-run fraction of time the chain spends in any region converges to its posterior probability.

Implementing Metropolis takes only a few lines. Below we shape the Gaussian proposal using the Cholesky factor of the Laplace covariance, $\mathbf{L}\mathbf{L}^{\top} = \mathbf{H}^{-1}$.

```
def metropolis(log_target, start, proposal_chol, n_steps):
    current = np.array(start, dtype=float)
    current_logp = log_target(current)
    states, accepted = np.empty((n_steps, len(current))), 0
    for t in range(n_steps):
        proposal = current + proposal_chol @ rng.standard_normal(len(current))
        proposal_logp = log_target(proposal)
        if np.log(rng.random()) < proposal_logp - current_logp:   # accept
            current, current_logp = proposal, proposal_logp
            accepted += 1
        states[t] = current            # a rejection repeats the state
    return states, accepted / n_steps
```

We launch four independent chains of 12,000 steps each from over-dispersed starting points scattered twice as widely as the Laplace Gaussian, $\theta_{\text{MAP}} + 2\mathbf{L}\epsilon$, proposing steps of size $1.1\,\mathbf{L}\epsilon$.

Because early states still remember where each chain began, we discard the first 2,000 iterations of each chain, the *warm-up* period, before computing averages.

```
def run_chains(scale, n_chains=4, n_steps=12000, warmup=2000):
    L = np.linalg.cholesky(laplace_cov)
    chains, rates = [], []
    for _ in range(n_chains):
        start = theta_map + 2 * L @ rng.standard_normal(2)  # over-dispersed
        states, rate = metropolis(log_joint, start, scale * L, n_steps)
        chains.append(states[warmup:])
        rates.append(rate)
    return np.array(chains), np.array(rates)

chains, rates = run_chains(1.1)
print('acceptance rates:', rates.round(3))
```

```
acceptance rates: [0.523 0.529 0.533 0.526]
```

Roughly half of all proposals are accepted across every chain. Remember that the Metropolis acceptance rule, not the Gaussian proposal, determines the stationary distribution, so a badly shaped proposal would slow the chain down without introducing bias.

36.8.3.3 Diagnostics for Markov Chain Mixing

Unlike independent samples, consecutive states in a Markov chain are correlated, and every chain begins wherever we initialize it. Before trusting any average computed from a chain, we ask three diagnostic questions and answer each with a concrete metric.

Have the chains forgotten their starting points and converged to the same distribution? Visually, a trace plot of each chain exposes sticking or drift, while $\widehat{R}$ quantifies disagreement across chains. Given multiple chains of N states each, let W denote the average within-chain variance and B denote N times the variance of the chain means. Then

$$\widehat{R}^2 = \frac{\frac{N-1}{N} W + \frac{1}{N} B}{W}. \tag{36.108}$$

The numerator estimates the total posterior variance and tends to overestimate it while the chains remain separated, whereas W underestimates it while each chain has explored only a piece of the posterior. Thus $\widehat{R}$ compares two variance estimates and approaches 1 as the chains mix together. In practice, we compute **split** $\widehat{R}$ by splitting every chain into two halves first, so that even a single drifting chain registers as two disagreeing chains. Staying within 0.01 of 1 is necessary before trusting the output, though never sufficient on its own: chains that all miss the same distant mode will happily agree with one another.

How many independent draws are these correlated states actually worth? Let ρ_k be the autocorrelation between states k steps apart. The integrated autocorrelation time $\kappa = 1 + 2\sum_{k \geq 1} \rho_k$ counts how many chain steps carry the same information as a single independent draw. Consequently, c chains of N states each yield an effective sample size of $N_{\text{eff}} = cN/\kappa$, carrying the same meaning and symbol as in importance sampling.

The simplest model of serial correlation is the first-order autoregressive process, AR(1), given by $x_t = \phi x_{t-1} + \varepsilon_t$ with independent noise ε_t and $|\phi| < 1$: at each step the chain

retains a fraction ϕ of its previous value. Its lag-k autocorrelation is $\rho_k = \phi^k$, which gives

$$\kappa = 1 + 2\sum_{k \geq 1} \phi^k = \frac{1+\phi}{1-\phi}.$$

At $\phi = 0.9$, we get $\kappa = 19$, meaning 10,000 states are worth only about 530 independent draws; at $\phi = 0.99$, $\kappa = 199$, leaving barely 50 independent draws.

Because estimated autocorrelations at very large lags are dominated by noise, our implementation groups them into consecutive pairs $\rho_{2k} + \rho_{2k+1}$ and truncates the sum at the first negative pair. Production inference packages refine both diagnostics further, using rank-normalized $\widehat{R}$ and separate effective sample sizes for the bulk and tails of the distribution.

How accurate is our estimated posterior mean? By the Markov chain central limit theorem, the **Monte Carlo standard error** (MCSE) of an estimated posterior mean is the posterior standard deviation divided by $\sqrt{N_{\text{eff}}}$.

```
def split_rhat(chains):
    """Split R-hat per coordinate; chains has shape (c, N, d)."""
    half = chains.shape[1] // 2
    halves = np.concatenate([chains[:, :half], chains[:, half:2 * half]])
    W = halves.var(axis=1, ddof=1).mean(axis=0)
    B = half * halves.mean(axis=1).var(axis=0, ddof=1)
    return np.sqrt(((half - 1) / half * W + B / half) / W)

def autocorrelation(x):
    """rho_k of one sequence for k = 0, ..., N - 1, by FFT."""
    x = x - x.mean()
    f = np.fft.rfft(x, n=2 * len(x))
    acov = np.fft.irfft(f * np.conj(f))[:len(x)]
    return acov / acov[0]

def chain_ess(chains):
    """c N / kappa per coordinate, summing rho in pairs (Geyer)."""
    c, N, d = chains.shape
    ess = np.empty(d)
    for j in range(d):
        rho = np.mean([autocorrelation(chain[:, j]) for chain in chains],
                      axis=0)
        pairs = rho[:N - 1:2] + rho[1:N:2]        # rho_2k + rho_2k+1
        negative = np.flatnonzero(pairs < 0)
        n_pairs = negative[0] if len(negative) else len(pairs)
        ess[j] = c * N / (-1 + 2 * pairs[:n_pairs].sum())
    return ess
```

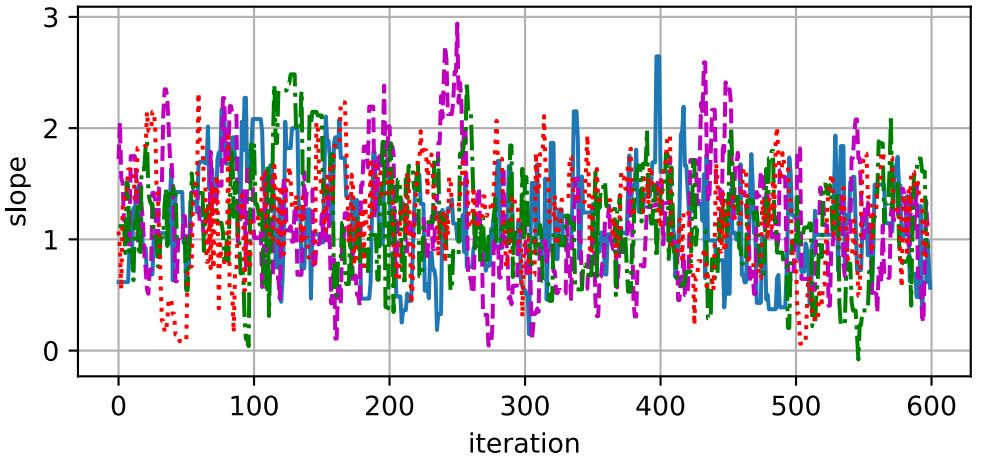
Let us put these diagnostics to the test. If our error bar is calibrated, the MCMC mean from our four chains should sit within roughly one MCSE of the exact grid mean. We also check how closely the measured autocorrelation time κ of the slope matches the AR(1) formula evaluated at the lag-1 autocorrelation ρ_1.

```
flat = chains.reshape(-1, 2)
ess = chain_ess(chains)
mcmc_mean, mcmc_sd = flat.mean(axis=0), flat.std(axis=0, ddof=1)
mcse = mcmc_sd / np.sqrt(ess)
rho1 = np.mean([autocorrelation(chain[:, 1])[1] for chain in chains])
print('split R-hat       :', split_rhat(chains).round(3))
print('ESS               :', ess.round().astype(int), 'of', len(flat))
print('(MCMC - grid)/MCSE:', ((mcmc_mean - post_mean) / mcse).round(1))
print(f'slope: rho_1 {rho1:.2f}, AR(1) kappa {(1 + rho1) / (1 - rho1):.1f},'
      f' measured kappa {len(flat) / ess[1]:.1f}')
```

```
split R-hat       : [1.001 1.001]
ESS               : [3798 3697] of 40000
(MCMC - grid)/MCSE: [ 0.6 -0.4]
slope: rho_1 0.81, AR(1) kappa 9.4, measured kappa 10.8
```

A trace plot of the first 600 post-warm-up iterations for the slope rounds out our diagnostic checks.

```
d2l.plot(np.arange(600), [chain[:600, 1] for chain in chains],
         xlabel='iteration', ylabel='slope', figsize=(6, 2.5))
```



The diagnostics look clean: split $\widehat{R}$ sits within 0.01 of 1, and the four trace plots interleave closely. The ESS is roughly one-tenth of the 40,000 kept states, giving $\kappa \approx 10$, right in line with the AR(1) prediction from the lag-1 autocorrelation $\rho_1 \approx 0.8$. Most reassuringly, the MCMC mean differs from the exact grid mean by roughly one MCSE, verifying that our uncertainty estimate actually matches the true estimation error.

36.8.3.4 Proposal Scale and Mixing Failures

The acceptance rate is the easiest number to print from a Metropolis run, yet by itself it tells us very little. Let us rerun the four chains with the proposal scale set to 0.05 and to 10 instead of 1.1. Before looking at the numbers, pause and guess which of the two extremes produces fewer effective draws.

```
for scale in [0.05, 10]:
    bad_chains, bad_rates = run_chains(scale)
    print(f'scale {scale:5}: acceptance {bad_rates.mean():.3f}, max R-hat '
          f'{split_rhat(bad_chains).max():.2f}, '
          f'min ESS {chain_ess(bad_chains).min():.0f}')
```

```
scale  0.05: acceptance 0.975, max R-hat 1.13, min ESS 43
scale    10: acceptance 0.020, max R-hat 1.00, min ESS 689
```

With tiny steps (scale 0.05), roughly 97% of proposals are accepted, yet $\widehat{R}$ rises well above 1.01 and 40,000 iterations yield only a few dozen effective draws because each step barely moves. With huge steps (scale 10), only about 2% of proposals are accepted and the chain gets stuck repeating states for long stretches, though it still achieves a few hundred effective draws. The optimal step size lies comfortably between these two extremes (Exercise 2). More fundamentally, a blind random walk moves uphill only by luck, whereas a proposal guided by the gradient of $\log \tilde{p}$ drifts uphill by design. Such gradient-informed moves arise from Langevin dynamics (Section 38.6.2), and their Metropolis-adjusted version is developed in Exercise 9.

36.8.4 Variational Inference

Sampling methods estimate each posterior expectation anew and always carry Monte Carlo noise. *Variational inference* takes a fundamentally different route, turning integration into optimization. We choose a tractable family of distributions q_ψ indexed by variational parameters ψ, optimize ψ so that q_ψ approximates the posterior as closely as possible, and then compute expectations under q_ψ either in closed form or by fast independent sampling.

36.8.4.1 The Evidence Lower Bound

To optimize q_ψ, we need an objective that measures closeness to the posterior without requiring the unknown normalizing constant $p(\mathcal{D})$. Fortunately, we already derived such an objective in Section 36.7, where an approximating distribution q over a latent variable gave a lower bound on the log evidence whose gap was a KL divergence. Treating the parameter vector θ as the latent variable and the dataset $\mathcal{D}$ as the observation yields the exact same identity:

$$\log p(\mathcal{D}) = \underbrace{\mathbb{E}_{q_\psi}\big[\log p(\mathcal{D},\theta) - \log q_\psi(\theta)\big]}_{\mathcal{L}(\psi)} + D_{\mathrm{KL}}\big(q_\psi \,\|\, p(\cdot \mid \mathcal{D})\big). \tag{36.109}$$

Since the KL divergence is always non-negative, $\mathcal{L}(\psi)$ is the **evidence lower bound** (ELBO) of q_ψ,

$$\mathcal{L}(\psi) = \mathbb{E}_{q_\psi}[\log p(\mathcal{D},\theta) - \log q_\psi(\theta)] \le \log p(\mathcal{D}). \tag{36.110}$$

Because the two terms on the right-hand side of Equation 36.109 sum to the fixed constant $\log p(\mathcal{D})$, every unit increase in $\mathcal{L}(\psi)$ decreases $D_{\mathrm{KL}}(q_\psi \,\|\, p(\cdot \mid \mathcal{D}))$ by exactly one unit. Maximizing the ELBO over ψ is therefore equivalent to minimizing the KL divergence, with

the unknown evidence dropping out as an additive constant. Every term inside the expectation can be evaluated pointwise, and the expectation itself can be estimated easily using draws from q_ψ.

36.8.4.2 The Direction of the KL Divergence

Notice the order of arguments in Equation 36.109: variational inference minimizes $D_{\mathrm{KL}}(q\|p) = \int q \log(q/p)$, the *reverse* KL divergence, with q in the first slot. Because its integrand blows up wherever $q > 0$ while $p \approx 0$, reverse KL penalizes q heavily for placing mass where the posterior has none. As a result, q shrinks away from low-density regions and tends to lock onto a single mode: reverse KL is *mode-seeking*. By contrast, the *forward* divergence $D_{\mathrm{KL}}(p\|q) = \int p \log(p/q)$ penalizes regions where $p > 0$ and $q \approx 0$, forcing q to stretch across all of the posterior's mass: forward KL is *mass-covering*. For a Gaussian family, minimizing forward KL simply matches the posterior's exact mean and covariance, but computing it requires expectations under the intractable posterior p, whereas reverse KL needs only expectations under q. Figure 36.41 contrasts the two behaviors on a bimodal target.

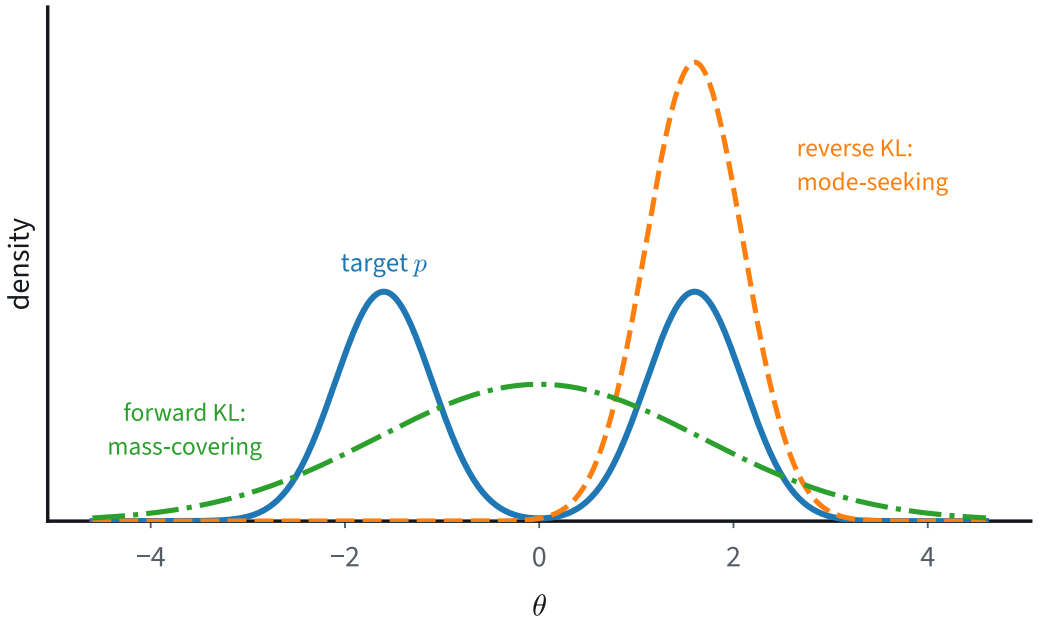


Figure 36.41: The direction of the KL divergence determines how an overly simple approximation fails. On a two-mode target, the reverse-KL Gaussian found by variational inference locks onto a single mode and underestimates uncertainty, whereas the forward-KL Gaussian covers both modes and places its peak in the low-density valley between them.

36.8.4.3 A Mean-Field Gaussian Fit

The simplest variational family is the *mean-field* Gaussian $q = \mathcal{N}(\mathbf{m}, \mathrm{diag}(\mathbf{s}^2))$, which models the parameters as independent. To compute gradients of the ELBO with respect to $\mathbf{m}$ and $\log \mathbf{s}$, we reparameterize a draw as $\theta = \mathbf{m} + \mathbf{s} \odot \epsilon$ with $\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$. Because the noise distribution does not depend on the variational parameters, we can move the gradient inside

the expectation (Section 34.7.3). The entropy $-\mathbb{E}_q[\log q]$ equals $\sum_j \log s_j$ up to an additive constant, giving the unbiased gradient formulas

$$\nabla_{\mathbf{m}}\mathcal{L} = \mathbb{E}[\nabla_\theta \log p(\mathcal{D},\theta)], \qquad \nabla_{\log \mathbf{s}}\mathcal{L} = \mathbb{E}[\nabla_\theta \log p(\mathcal{D},\theta) \odot \mathbf{s} \odot \epsilon] + \mathbf{1}, \tag{36.111}$$

where $\mathbf{1}$ comes from differentiating the entropy $\sum_j \log s_j$ with respect to $\log \mathbf{s}$. The first gradient shifts the mean uphill on average; the second expands q until the entropy's outward push balances the posterior's downward curvature. Before running the optimizer on our logistic posterior, we can predict what will happen by solving the Gaussian target case exactly.

Proposition. For a Gaussian target $p = \mathcal{N}(\mu, \Lambda^{-1})$ with precision matrix Λ, the mean-field Gaussian minimizing $D_{\mathrm{KL}}(q\|p)$ has mean μ and variances $s_j^2 = 1/\Lambda_{jj}$, the *conditional* variances of the target. In two dimensions with correlation ρ, each s_j is $\sqrt{1-\rho^2}$ times the true marginal standard deviation.

Proof. Dropping constants independent of q, $D_{\mathrm{KL}}(q\|p) = -\sum_j \log s_j + \frac{1}{2}\mathbb{E}_q[(\theta-\mu)^\top \Lambda(\theta-\mu)]$. Because the coordinates are independent under q with means m_j and variances s_j^2, all cross terms in the fluctuations vanish in expectation, leaving $\sum_j \Lambda_{jj} s_j^2 + (\mathbf{m}-\mu)^\top \Lambda (\mathbf{m}-\mu)$. Since Λ is positive definite, the quadratic in $\mathbf{m}$ is minimized at $\mathbf{m} = \mu$. Setting the derivative with respect to s_j to zero, $-1/s_j + \Lambda_{jj} s_j = 0$, gives $s_j^2 = 1/\Lambda_{jj}$. For $d = 2$, inverting the 2×2 covariance matrix yields $1/\Lambda_{jj} = \sigma_j^2(1-\rho^2)$, where σ_j is the marginal standard deviation. ■

A diagonal Gaussian cannot rotate its axes to align with a correlated posterior, and under reverse KL it pays a steep penalty for spilling into low-density corners. It therefore contracts to the width of a conditional slice through the posterior rather than the full marginal width (Figure 36.42). Notice that this contraction is specifically a consequence of combining a factorized family with reverse KL: a full-covariance Gaussian fitted by the exact same ELBO objective to a Gaussian target recovers it without error. In our logistic regression example, $\rho = -0.86$, so the proposition predicts that mean-field standard deviations will be roughly $\sqrt{1-0.86^2} = 0.51$ times the true marginal standard deviations, about half as wide.

Let us check this prediction numerically. We initialize at the Laplace approximation and run 3,000 iterations of stochastic gradient ascent using Equation 36.111, with 256 reparameterized draws per step and a learning rate decaying as $1/\sqrt{t}$. Averaging the final 1,000 iterates smooths out optimization noise, letting us compare the fitted standard deviations directly against the proposition's prediction using the grid correlation ρ.

```
m_vi, log_s_vi = theta_map.copy(), np.log(laplace_sd)
m_avg, log_s_avg = np.zeros(2), np.zeros(2)
n_steps, n_avg = 3000, 1000
for t in range(1, n_steps + 1):
    eps = rng.standard_normal((256, 2))
    s_vi = np.exp(log_s_vi)
    g = grad_log_joint(m_vi + s_vi * eps)        # at theta = m + s * eps
    lr = 0.05 / np.sqrt(t)
    m_vi += lr * g.mean(axis=0)
    log_s_vi += lr * ((g * eps * s_vi).mean(axis=0) + 1)
```

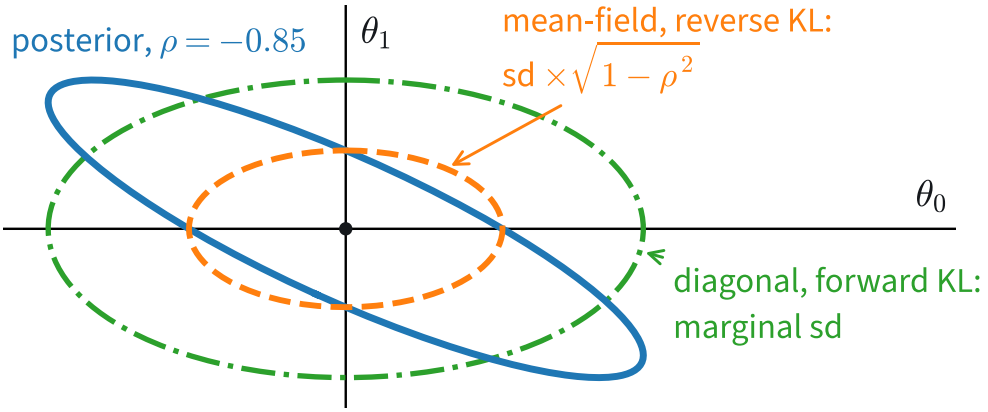


Figure 36.42: Diagonal Gaussians fitted to a correlated Gaussian posterior with standard deviations $(1, 0.5)$ and correlation $\rho = -0.85$, shown as 2-sd ellipses. Because a factorized family cannot tilt, reverse KL (orange) contracts to the conditional width, $\sqrt{1-\rho^2} \approx 0.53$ times the marginal standard deviation, whereas forward KL (green) preserves the marginal width while spilling into low-density regions.

```
    if t > n_steps - n_avg:                    # average the last iterates
        m_avg += m_vi / n_avg
        log_s_avg += log_s_vi / n_avg
s_avg = np.exp(log_s_avg)
print('             mean             sd')
print('grid      ', post_mean.round(3), post_sd.round(3))
print('Laplace   ', theta_map.round(3), laplace_sd.round(3))
print('mean-field', m_avg.round(3), s_avg.round(3))
print('predicted sd, grid sd * sqrt(1 - rho^2):',
      (post_sd * np.sqrt(1 - post_rho**2)).round(3))
```

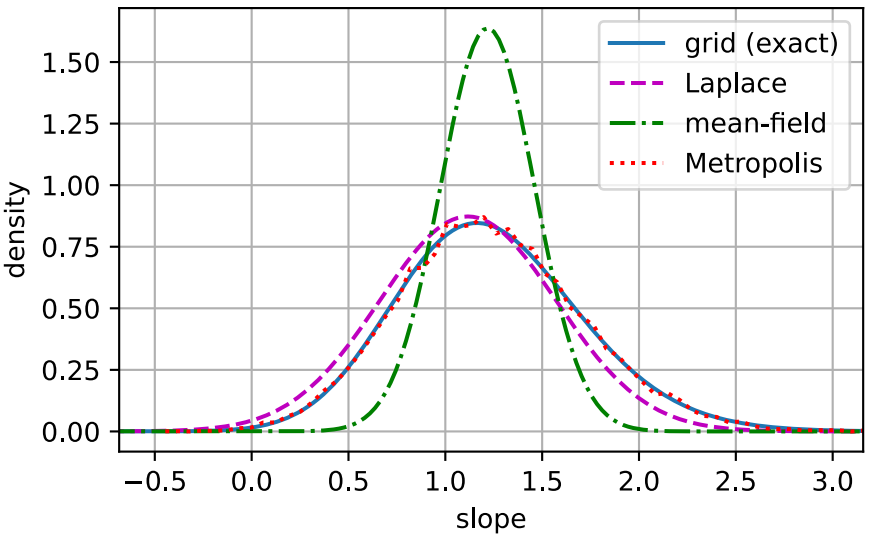
```
             mean             sd
grid       [-2.347  1.237] [1.042 0.48 ]
Laplace    [-2.138  1.117] [1.004 0.457]
mean-field [-2.276  1.218] [0.533 0.244]
predicted sd, grid sd * sqrt(1 - rho^2): [0.536 0.247]
```

The mean-field mean lands within roughly a tenth of a posterior standard deviation of the grid mean, while its standard deviations shrink to about half the true widths, matching our analytical predictions (0.54 and 0.25) to within 2%. The optimizer has converged cleanly, so the remaining discrepancy comes entirely from the restrictive variational family. Richer families, such as full-covariance Gaussians, mixtures, or normalizing flows, close that gap at higher computational cost. Keep in mind that a converged ELBO only confirms that optimization has finished *within the chosen family*; by itself, it never certifies that the family actually fits the posterior.

36.8.5 Comparison of the Methods

We now have four distinct approximations to the exact same posterior. A single plot brings them side by side on the marginal density of the slope θ_1: the exact grid curve, the Laplace and mean-field Gaussians, and a histogram of the Metropolis samples.

```
def gauss_pdf(t, mean, sd):
    return np.exp(-0.5 * ((t - mean) / sd)**2) / (sd * np.sqrt(2 * np.pi))

b1 = grid_axes[1]
exact = grid_weight.reshape(m, m).sum(axis=0) / (b1[1] - b1[0])
hist, edges = np.histogram(flat[:, 1], bins=60, density=True)
d2l.plot([b1, b1, b1, (edges[:-1] + edges[1:]) / 2],
         [exact, gauss_pdf(b1, theta_map[1], laplace_sd[1]),
          gauss_pdf(b1, m_avg[1], s_avg[1]), hist],
         xlabel='slope', ylabel='density', figsize=(5, 3),
         xlim=(post_mean[1] - 4 * post_sd[1], post_mean[1] + 4 * post_sd[1]),
         legend=['grid (exact)', 'Laplace', 'mean-field', 'Metropolis'])
```



The Metropolis histogram tracks the exact posterior faithfully, skewness and all. The Laplace curve sits centered at the mode to the left of the true mean and is slightly too narrow. The mean-field Gaussian is roughly half as wide as the exact marginal, and by construction it reports a correlation of zero against the true $\rho = -0.86$.

Method	Representation	Main diagnostic	Characteristic failure
Grid	weighted points on a lattice	mass at the edge of the grid	cost m^d in dimension d
Laplace	one Gaussian at the mode	comparison with samples	skew, tails, boundaries, other modes
Rejection sampling	independent exact draws	acceptance rate	rate falls like $n^{-d/2}$; needs a bound M

Method	Representation	Main diagnostic	Characteristic failure
Importance sampling	independent weighted draws	weight ESS, largest weight, reruns	light-tailed or misplaced proposal
MCMC	correlated posterior draws	traces, split $\widehat{R}$, ESS, MCSE	slow mixing, unvisited modes
Variational inference	optimized member of a family	comparison with samples	family too narrow; factorized q too confident

Which method should you reach for in practice? With only a handful of parameters, use a grid. For a smooth, unimodal posterior, the Laplace approximation gives a fast first answer and a strong starting proposal. When an expectation requires a trustworthy error bar, turn to MCMC paired with its three diagnostics. Variational inference shines when posterior approximation must sit inside a larger optimization loop, provided you remember that factorized families underestimate uncertainty. Because neither Gaussian approximation provides a self-contained accuracy check, both should be validated against a sampler whenever feasible.

It is worth being candid about scale: none of these four methods scales cleanly to the full parameter posterior of a large neural network. Importance weights collapse as dimension grows, random-walk MCMC needs progressively more steps per independent draw, the full Hessian has d^2 entries, and mean-field q understates uncertainty by the correlation factor we just derived. Worse still, a neural network's architectural symmetries duplicate every posterior mode for every permutation and sign flip of its hidden units (Exercise 7).

For large networks, the most widely used answer to the question that opened this section is simpler than any of the four methods. We train the same network several times from different random initializations and average the predicted probabilities. Such a *deep ensemble* (Lakshminarayanan et al. 2017) is a crude Monte Carlo estimate of the posterior predictive Equation 36.96, in which a handful of independently trained networks stand in for posterior draws. Each run settles in a different mode, so the average spans several modes, which a single Laplace or mean-field fit cannot do. Where the members disagree, typically far from the training data, the averaged prediction becomes less confident. The members are not posterior samples, however, and nothing guarantees that they should be weighted equally. Their errors are also correlated, so adding members helps only up to a point: with pairwise correlation ρ, the variance of the average never falls below the fraction ρ of a single member's variance (Exercise 2 in Section 36.2).

The core ideas of this section nevertheless reappear throughout deep learning wherever a distribution is known only up to a normalizing constant. The evidence lower bound trains variational autoencoders (Kingma and Welling 2014) (Section 17.7), where an encoder network outputs q, as well as discrete diffusion models (Section 17.9). Importance weights and effective sample size allow policy-gradient methods to reuse past trajectories (Section 15.2). And gradient-guided proposals, stripped of their accept–reject step, become the Langevin samplers powering score-based generative models (Section 17.1). Finally, every sampling method returns an empirical average over finitely many draws; the next section asks how tightly such finite-sample averages concentrate around their expectations when the draws are independent.

36.8.6 Summary

Training a model to a single parameter vector treats that point estimate as if it were known with certainty. Far from the training data, plausible parameter vectors can disagree by orders of magnitude, and averaging over them in the posterior predictive can shift rare-event probabilities by a factor of several. Computing that average requires integrating against a posterior known only up to its normalizing constant. All four methods in this section sidestep the unknown constant by working only with ratios or differences where it cancels: local curvature at the mode (Laplace), density ratios $\tilde{p}/q$ (rejection and importance sampling), acceptance ratios $\tilde{p}(\theta')/\tilde{p}(\theta)$ (Metropolis–Hastings), and an optimization objective that differs from reverse KL divergence by a constant (variational inference). The samplers bring built-in diagnostics (weight ESS for importance sampling, and split $\widehat{R}$, autocorrelation time κ, and MCSE for MCMC), while Gaussian approximations must be checked against samples. Although full Bayesian inference remains intractable for large neural networks, the evidence lower bound, importance weights, and Langevin dynamics return across the book whenever we work with unnormalized distributions.

36.8.7 Exercises

1. **Metropolis–Hastings acceptance.** The acceptance probability Equation 36.106 turns any proposal into a chain that targets the posterior.
   1. On a continuous state space the transition kernel has a density part $g(\theta' \mid \theta)\alpha(\theta, \theta')$ and an atom at the current state θ of mass $r(\theta) = 1 - \int g(\theta' \mid \theta)\alpha(\theta, \theta')\, d\theta'$, the probability of a rejection. Show that the posterior is stationary, accounting for the atom. *Hint:* the atom contributes $\pi(\theta')r(\theta')$ to the density at θ' after one step.
   2. An independence sampler proposes $\theta' \sim q$ regardless of the current state. Show that its acceptance probability is $\min(1, w(\theta')/w(\theta))$ with the importance weight w of Equation 36.102. What does the chain do after it reaches a state whose weight is far larger than the weights of typical proposals?
   3. Barker's rule accepts a symmetric proposal with probability $\tilde{p}(\theta')/(\tilde{p}(\theta)+\tilde{p}(\theta'))$. Show that it satisfies detailed balance. Write its transition matrix for the two-state chain of the text, and show that the Metropolis chain moves more often from both states. For a two-state chain the autocorrelation of any function of the state is $\rho_k = \lambda^k$, where λ is the second eigenvalue of the transition matrix. Compute λ and $\kappa = (1+\lambda)/(1-\lambda)$ for both rules. Which gives the smaller Monte Carlo error for the same number of steps? *Hint:* Barker gives the off-diagonal entries $3/4$ and $1/4$, against Metropolis' 1 and $1/3$.
2. code **Proposal scale.** The function `run_chains` multiplies the Cholesky factor of `laplace_cov` by a scale, 1.1 in the text.
   1. Sweep the scale over $\{0.05, 0.2, 0.5, 1.1, 1.7, 2.4, 3, 10\}$, keeping the 12,000 steps per chain and the warm-up. For each value, record the acceptance rate, the split $\widehat{R}$, and the smallest ESS of `chain_ess` divided by the $48{,}000$ log-joint evaluations of the four chains. Plot the ESS per evaluation against the acceptance rate. Over which range of acceptance rates does the ESS per evaluation stay within a factor of two of its maximum?
   2. For a d-dimensional standard Gaussian target and the proposal $\mathcal{N}(\theta, (\ell^2/d)\mathbf{I})$, Roberts et al. (1997) show that as $d \to \infty$ the efficiency is largest at $\ell \approx 2.38$, where the acceptance rate is about 0.234. The logistic posterior is close to its

Laplace approximation, so the Cholesky factor makes it roughly standard. Which scale does this result suggest for $d = 2$? How do the best scale of your sweep and its acceptance rate compare?

3. Which of the diagnostics of the text flag the run with scale 0.05? Is the acceptance rate one of them? How far is that run's posterior mean from the grid mean, in units of its own MCSE?

3. **Laplace and parameterization.** A coin shows 9 heads and 4 tails. Under a uniform prior, the posterior of its heads probability θ is $\text{Beta}(10, 5)$, with mean $2/3$, standard deviation 0.118, and central 95% interval $[0.419, 0.872]$.
   1. The Laplace approximation Equation 36.100 in the parameter θ has mean $9/13$ and standard deviation 0.128, as computed in the text. Give its central 95% interval and the probability it places outside $[0, 1]$.
   2. In the log-odds $\eta = \log(\theta/(1-\theta))$ the posterior density is proportional to $\text{sigmoid}(\eta)^{10}(1 - \text{sigmoid}(\eta))^{5}$, because the Jacobian $d\theta/d\eta = \theta(1-\theta)$ adds one count to each side. Show that its mode is $\eta = \log 2$, compute the curvature of the log density there, and map the central 95% interval of the Laplace approximation back to θ.
   3. Which of the two approximations is closer to the exact posterior? Why does the Laplace approximation depend on the parameterization, whereas the exact posterior probability of an event such as $\theta > 1/2$ does not?

4. code **Moderated predictions.** ● Under the Laplace approximation Equation 36.100, the logit $a = \mathbf{z}_\star^\top \theta$ at a test input with $\mathbf{z}_\star = (1, x_\star)^\top$ is Gaussian with mean $\mu_a = \mathbf{z}_\star^\top \theta_{\text{MAP}}$ and variance $s_a^2 = \mathbf{z}_\star^\top \mathbf{H}^{-1} \mathbf{z}_\star$. The probit approximation $\text{sigmoid}(a) \approx \Phi(\lambda a)$ with $\lambda^2 = \pi/8$, where Φ is the standard normal c.d.f., turns the predictive integral Equation 36.96 into a closed form (MacKay 1992).
   1. Show that $\mathbb{E}[\Phi(\lambda a)] = \Phi(\lambda \mu_a / \sqrt{1 + \lambda^2 s_a^2})$ for $a \sim \mathcal{N}(\mu_a, s_a^2)$. *Hint:* write $\Phi(\lambda a)$ as $P(U \le \lambda a)$ for an independent standard normal U. Conclude that the moderated prediction $\text{sigmoid}(\mu_a / \sqrt{1 + \pi s_a^2/8})$ lies between $\text{sigmoid}(\mu_a)$ and $1/2$.
   2. Compute the moderated predictions at $x_\star \in \{-3, 2, 7\}$ from `theta_map` and `laplace_cov`, and compare them with the plug-in and posterior predictive values that your run prints.
   3. Compute three quantities at the same inputs: the plug-in $\text{sigmoid}(\mathbf{z}_\star^\top \theta_{\text{MAP}})$, the prediction at the posterior mean $\text{sigmoid}(\mathbf{z}_\star^\top \mathbb{E}[\theta \mid \mathcal{D}])$, and the posterior predictive $\mathbb{E}[\text{sigmoid}(\mathbf{z}_\star^\top \theta) \mid \mathcal{D}]$. Which of the two steps, from mode to mean and from mean to average, moves the prediction away from $1/2$, and which toward it? Why are both steps small at $x_\star = 2$ but not at $x_\star = -3$ or $x_\star = 7$? What does this imply for using the Laplace approximation to moderate overconfident predictions?

5. code **Predicting the weight ESS.** ● By the proposition of the text, N_{eff}/N approaches $(\mathbb{E}_q[w])^2 / \mathbb{E}_q[w^2]$ as the number N of draws grows.
   1. Let the normalized target be $\mathcal{N}(\mu, s_p^2)$ and the proposal $\mathcal{N}(\nu, s_q^2)$ in one dimension, and let $r = s_q / s_p$. Show that for $s_q^2 > s_p^2/2$

$$\frac{\mathbb{E}_q[w^2]}{(\mathbb{E}_q[w])^2} = \frac{r^2}{\sqrt{2r^2 - 1}} \exp\left(\frac{(\mu - \nu)^2}{2 s_q^2 - s_p^2} \right),$$

   and that the ratio is infinite for $s_q^2 \le s_p^2/2$. Which r minimizes the ratio when $\mu = \nu$?

2. Approximate the posterior by independent Gaussians with the grid means and standard deviations, ignoring the correlation, so that the ratio factorizes over the coordinates. Predict the ESS of the 20,000 prior-proposal draws and compare it with the printed value. Repeat for the prior proposals with $\tau = 0.5$ and $\tau = 10$: set `prior_var` $= \tau^2$ in the model cell, rerun the cells through the importance cell, and compare prediction and measurement. Which factor of the formula makes a narrow prior a poor proposal, and which makes a broad prior a poor one?
3. Under the same approximation, for which coordinates of your run does the proposal $\mathcal{N}(\theta_{\text{MAP}}, 0.3\,\mathbf{I})$ have infinite weight variance? Run it ten times with 5,000 draws and record the ESS, the largest normalized weight, and the estimated posterior mean. Why does a run whose estimate looks plausible not show that the proposal is adequate?
4. Rejection sampling from the prior keeps a draw with probability $L(\theta)/\max L$, where L is the likelihood and $\max L$ its value at the maximum-likelihood estimate (for separable data, its supremum 1). Measure the acceptance rate on your data and on data with ten times as many points. Explain the decay with n from the volume of the posterior, and compare the rate with the ESS fraction of importance sampling from the same prior.

6. code **Estimating the evidence.** ● Let q be a normalized proposal, let $w = p(\mathcal{D}, \theta)/q(\theta)$, and let $\hat{Z}_N = \frac{1}{N}\sum_{s=1}^{N} w(\theta_s)$ for independent draws $\theta_s \sim q$.
   1. Show that $\hat{Z}_N$ is an unbiased estimate of $p(\mathcal{D})$ and that $\mathbb{E}[\log \hat{Z}_N] \le \log p(\mathcal{D})$. Show that $\mathbb{E}[\log \hat{Z}_1]$ is the ELBO Equation 36.110 of q and that $\mathbb{E}[\log \hat{Z}_N]$ does not decrease with N (Burda et al. 2016).
   2. The function `log_joint` omits the normalizing constant $-\log(2\pi\tau^2)$ of the two-dimensional prior. Add it, and compute $\log p(\mathcal{D})$ in three ways: by summing the density over the grid and multiplying by the area of a grid cell, by $\hat{Z}_N$ with 5,000 Laplace-proposal draws, and by the Laplace evidence $\log \tilde{p}(\theta_{\text{MAP}}) + \log 2\pi - \frac{1}{2}\log\det \mathbf{H}$. The Laplace estimate of $p(\mathcal{D})$ comes out a few percent low, about 0.04 in $\log p(\mathcal{D})$. Which feature of the posterior, seen in the grid cell, explains the sign?
   3. Estimate the ELBO of the Laplace approximation and the mean-field fit using 10^5 draws from each and report the MCSE of each estimate. By Equation 36.109, the gap between $\log p(\mathcal{D})$ and the ELBO of q is $D_{\text{KL}}(q \| p(\cdot \mid \mathcal{D}))$. Compare the two Gaussians by this divergence.

7. code **A posterior with two modes.** ● Fifty observations are drawn as $y_i \sim \mathcal{N}(\theta^2, 1)$ with the true value $\theta = 1$, and θ has the prior $\mathcal{N}(0, 4)$. The likelihood depends on θ only through θ^2, just as the loss of a neural network does not change when the incoming weights, the bias, and the outgoing weights of a tanh hidden unit flip sign together. The posterior is therefore symmetric under $\theta \mapsto -\theta$.
   1. Compute the posterior on a one-dimensional grid. Report $\mathbb{E}[\theta \mid \mathcal{D}]$, $\mathbb{E}[\theta^2 \mid \mathcal{D}]$, and the ratio of the posterior density at $\theta = 0$ to its value at a mode.
   2. Show that the Gaussian that minimizes the forward divergence $D_{\text{KL}}(p\|q)$ matches the mean and the variance of p. Where does this Gaussian put its mode for the present posterior, and how plausible is that point under p?
   3. Fit three local approximations: the Laplace approximation at the positive mode, a one-dimensional Gaussian fitted by the ELBO ascent of the variational cell started at $m = 0.5$, and four Metropolis chains with proposal standard deviation 0.2 started at $\theta \in \{0.5, 1, 1.5, 2\}$. Then rerun the chains from $\theta \in \{-2, -0.5, 0.5, 2\}$. Which methods report a single mode? For each run, which

of its diagnostics (split $\widehat{R}$ and the ESS for the chains, the ELBO for the variational fit) reveal the problem?

4. Which posterior expectations does a single-mode approximation still estimate correctly? A network with h hidden tanh units has the sign flip of this problem for every unit and, in addition, the $h!$ permutations of its hidden units (Exercise 6). How many equivalent copies of a mode does its posterior have under an isotropic Gaussian prior? What does this suggest about approximating the posterior of a neural network by one of the methods of this section?

8. code **Tails of the logistic posterior.** ●● A central limit theorem for importance sampling needs a finite second moment of the weights. The Laplace proposal of the importance cell usually reaches a high ESS. Throughout, $\tau^2 = \texttt{prior_var} = 4$ is the prior variance in the posterior Equation 36.97.
   1. Fix a unit vector $\mathbf{v}$ and let $\theta = t\mathbf{v}$. Show that the log-likelihood in Equation 36.97 equals $-c(\mathbf{v})t + O(1)$ as $t \to \infty$, where $c(\mathbf{v})$ is the sum of $|\mathbf{z}_i^\top \mathbf{v}|$ over the observations with $y_i = 1$ and $\mathbf{z}_i^\top \mathbf{v} < 0$ or with $y_i = 0$ and $\mathbf{z}_i^\top \mathbf{v} > 0$. For which datasets is $c(\mathbf{v}) > 0$ for every $\mathbf{v}$?
   2. Conclude that a Gaussian proposal $\mathcal{N}(\mathbf{m}, \mathbf{S})$ has $\mathbb{E}_q[w^2] = \infty$ when some eigenvalue of $\mathbf{S}$ is smaller than $\tau^2/2$, and $\mathbb{E}_q[w^2] < \infty$ when every eigenvalue of $\mathbf{S}$ exceeds $\tau^2/2$. Which of the two proposals of the importance cell has weights with finite variance? Compare this answer with the one that the formula of Exercise 5a gives for the Laplace proposal when the posterior is approximated by a Gaussian.
   3. Along each eigenvector of `laplace_cov`, in both directions, find the Mahalanobis distance from θ_{MAP} beyond which $\tilde{p}^2/q$ exceeds its value at θ_{MAP}. How likely is a single Laplace-proposal draw to land that far out? Reconcile your answer with the ESS that your run prints.
   4. Geweke (1989) builds importance densities from multivariate Student-t approximations at the posterior mode. Use a Student-t proposal with 3 degrees of freedom, location θ_{MAP}, and scale matrix `laplace_cov`, and show that its weights have finite variance. Compare it with the Laplace proposal over 20 runs on the ESS and on the estimate of the tail probability $P(\theta_1 > \bar{\theta}_1 + 3s_1 \mid \mathcal{D})$, where $\bar{\theta}_1$ and s_1 are the grid mean and standard deviation of the slope; for a reference value, evaluate the posterior on a grid twice as wide. Also try the defensive mixture of the text, and explain why its weights are bounded.

9. code **Langevin proposals.** ●● The Metropolis-adjusted Langevin algorithm proposes $\theta' = \theta + \frac{\epsilon^2}{2}\mathbf{C}\nabla \log \tilde{p}(\theta) + \epsilon\, \mathbf{C}^{1/2}\xi$ with $\xi \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ and a preconditioner $\mathbf{C}$; take $\mathbf{C}$ to be the Laplace covariance $\mathbf{H}^{-1}$ and $\mathbf{C}^{1/2}$ its Cholesky factor.
   1. Write down the proposal density $g(\theta' \mid \theta)$ and the Hastings factor of the acceptance rule Equation 36.106. Why is the proposal not symmetric?
   2. Implement the sampler with `grad_log_joint` and run four chains for $\epsilon \in \{0.5, 1, 1.5\}$. Report the acceptance rate, the split $\widehat{R}$, and the ESS per gradient evaluation, and compare with the random walk's ESS per log-joint evaluation.
   3. Without the accept step the chain is the unadjusted Langevin algorithm. Run it at $\epsilon = 1$ and compare its mean and standard deviations with the grid. Which of the two errors, the bias of the discretization or the Monte Carlo error, does the Metropolis step remove?

36.9 Concentration Inequalities

Whenever we report a model's test accuracy, we are reporting a measurement made on a finite sample. In Section 36.5, we attached an error bar to that measurement, $\pm 1.96\,\hat{\sigma}/\sqrt{n}$, by invoking the central limit theorem. That Gaussian approximation is convenient and often accurate, but it is exact only in the limit $n \to \infty$, and it relies on plugging in an estimated standard deviation $\hat{\sigma}$. Earlier, in Section 36.1, we saw a bound that holds with no approximations at every finite n, Chebyshev's inequality, yet it turned out to be painfully weak. This leaves us with a natural question: how far can a test-set error stray from the truth if we want a guarantee that holds at the sample size we actually have, and remains useful even when we demand that it almost never fail?

To make the gap concrete, consider a classifier that makes 830 mistakes on 10,000 test images, giving an observed error rate of 8.3%. Chebyshev's inequality bounds the probability that the average of n independent draws misses its mean by t or more by $\sigma^2/(nt^2)$. Because a 0/1 outcome has variance at most $\sigma^2 \le \frac{1}{4}$, Chebyshev guarantees that the true error lies within $\pm 2.2\%$ of 8.3% with 95% confidence. That is already wider than the Gaussian interval, and worse still, it collapses when we demand high reliability: ask for a guarantee that fails only once in a million, and Chebyshev answers $\pm 500\%$, which tells us nothing at all. Yet the true probability that an average of 0/1 outcomes strays far from its mean decays *exponentially* in n, not like $1/n$. A bound that captures this exponential decay answers $\pm 1.4\%$ at 95% confidence and $\pm 2.7\%$ at one in a million.

The key trick behind these sharper bounds is surprisingly simple: apply Markov's inequality not to the average itself, but to its exponential, which turns a sum of independent terms into a product. Sergei Bernstein developed bounds of this kind in the 1920s, Herman Chernoff used the method in 1952 to measure how rapidly statistical tests become reliable, and Wassily Hoeffding gave in 1963 the clean form used throughout machine learning (Hoeffding 1963).

Why would a practitioner ever ask for a guarantee that fails only once in a million? In practice, we rarely evaluate just one model: a learner compares many candidates and keeps the best. Guaranteeing that *none* of them looks good by chance requires a tiny failure probability for each one, and only an exponentially shrinking tail makes that affordable at a realistic sample size, a theme we will develop fully in Section 36.10. In this section, we first use exponential tail bounds to size a test set for a desired precision. We then apply the exact same machinery to the coordinates of a random vector in $\mathbb{R}^d$, revealing two striking geometric facts: a standard Gaussian vector in $\mathbb{R}^{784}$ almost always has length within a few percent of 28, and two independent random directions are nearly orthogonal. Both facts underpin neural network initialization and high-dimensional embeddings.

Three results from this section are worth committing to memory: the Chernoff method Equation 36.112, Hoeffding's confidence interval Equation 36.116, and the concentration of the Euclidean norm Equation 36.121.

```
rng = np.random.default_rng()
```

36.9.1 Exponential Tail Bounds

The Gaussian confidence interval is narrow because Gaussian tails decay like $e^{-z^2/2}$, a rate inherited from the central limit theorem and therefore guaranteed only in the limit of large

n. We want that same rapid decay combined with a guarantee that holds at every finite sample size. Before building such a bound, let us see just how far Chebyshev's inequality falls short of the truth.

36.9.1.1 Chebyshev's Rate

Chebyshev's bound is simply Markov's inequality (Section 36.1) applied to a squared deviation. Consider the sample mean $\bar{X}$ of n independent draws, each with mean μ and finite variance σ^2. Because the expected squared deviation is $\mathbb{E}[(\bar{X} - \mu)^2] = \mathrm{Var}(\bar{X}) = \sigma^2/n$, Markov's inequality yields

$$P(|\bar{X} - \mu| \geq t) \leq \frac{\sigma^2}{nt^2}.$$

Here the miss probability shrinks only like $1/n$. To push the failure probability from 1 down to 10^{-6} at a fixed deviation t, we must grow the sample size a million-fold.

How small is the actual miss probability? The simplest setting where we can compute it exactly is a fair coin. Let $\hat{p}$ denote the fraction of heads in n independent flips, and ask for the probability that $\hat{p}$ misses $\frac{1}{2}$ by $t = 0.1$ or more. Each flip has variance $\frac{1}{4}$, so Chebyshev's bound is $0.25/(0.01\,n) = 25/n$. The exact probability is a sum of binomial probabilities, which we evaluate in log space so that $n = 5{,}000$ does not underflow to zero.

```
def exact_tail(n, t):
    """P(|p_hat - 1/2| >= t) for n fair flips, a binomial sum in log space."""
    k = np.arange(n + 1)
    log_pmf = n * np.log(0.5) + np.concatenate(
        ([0.0], np.cumsum(np.log((n - k[1:] + 1) / k[1:]))))
    in_tail = log_pmf[np.abs(k / n - 0.5) >= t - 1e-12]  # fp-safe boundary
    m = in_tail.max()
    return np.exp(m) * np.exp(in_tail - m).sum()  # stable log-sum-exp
```

At $n = 1{,}000$ Chebyshev allows a miss in one experiment out of forty. Before reading the table, guess how often such a miss really happens.

```
t = 0.1
print(f'{"n":>6} {"exact":>10} {"Chebyshev":>10}')
for n in (10, 100, 1000, 5000):
    print(f'{n:6d} {exact_tail(n, t):10.2e} {0.25 / (n * t**2):10.2e}')
```

```
     n      exact  Chebyshev
    10   7.54e-01   2.50e+00
   100   5.69e-02   2.50e-01
  1000   2.73e-10   2.50e-02
  5000   1.30e-45   5.00e-03
```

At $n = 10$ the exact tail is 0.75 while the bound, 2.5, is trivial. At $n = 100$ the bound drops to 0.25, roughly four times the exact 0.057. Beyond that point the two curves part ways dramatically. At $n = 1{,}000$ the exact tail is 2.7×10^{-10} whereas Chebyshev still allows 0.025, eight orders of magnitude larger; by $n = 5{,}000$ the gap has widened to forty-two orders of magnitude. Every tenfold increase in n merely divides Chebyshev's bound by ten,

while the exact tail sheds a fixed number of decimal places for every hundred flips: on a logarithmic scale it descends along a straight line.

Unfortunately, Chebyshev's bound cannot be tightened without assuming more about the distribution. As we saw in Section 36.1, it holds with equality for a distribution supported on three points, so no bound relying solely on the mean and variance can improve upon it in general. A coin flip, however, has far more structure than two finite moments: its values are bounded, which keeps all of its higher moments small as well. To harness all moments at once, we turn to the moment generating function.

36.9.1.2 The Chernoff Method

To see how to use higher moments, look at what Markov's and Chebyshev's inequalities have in common. Suppose a function g lies everywhere above the indicator $\mathbf{1}\{x \geq t\}$, a *majorant* of the indicator. Taking expectations immediately gives $P(X \geq t) = \mathbb{E}[\mathbf{1}\{X \geq t\}] \leq \mathbb{E}[g(X)]$. Markov's inequality chooses the linear majorant $g(x) = x/t$ for a nonnegative X, while Chebyshev's chooses the quadratic $g(x) = (x/t)^2$ (applied to the centered deviation $X - \mu$). Now notice that for any $\lambda > 0$, the exponential $e^{\lambda(x-t)}$ also lies above the indicator across the entire real line. Panel (a) of Figure 36.43 compares all three majorants for a nonnegative X. Since the bound is simply the average of $g(X)$, what determines its tightness is how small g stays where the probability mass of X actually sits.

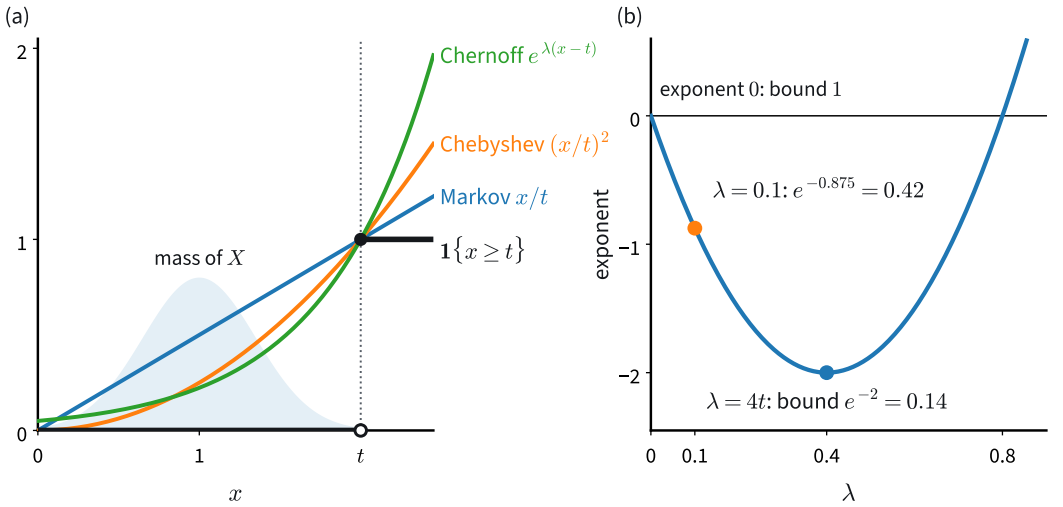


Figure 36.43: A tail bound is the expectation of a function lying above the indicator. (a) For a threshold $t = 2$, the indicator $\mathbf{1}\{x \geq t\}$ (black) and three functions above it: Markov's x/t, Chebyshev's $(x/t)^2$, and Chernoff's $e^{\lambda(x-t)}$ with $\lambda = 1.5$. The shaded bump marks where the mass of X sits; each bound is the average of its function under that mass, and raising λ makes the exponential small there. (b) The Chernoff exponent $-\lambda nt + n\lambda^2/8$ for the mean of $n = 100$ fair coins at $t = 0.1$. Every $\lambda > 0$ gives a valid bound; the best, at the minimum $\lambda = 4t$ of the parabola, is e^{-2}.

The expectation of this exponential has a standard name. The *moment generating function* (MGF) of X is $\mathbb{E}[e^{\lambda X}]$, viewed as a function of λ wherever the expectation is finite. For a fair coin centered at zero, $Z \in \{-\frac{1}{2}, \frac{1}{2}\}$, it is simply $\frac{1}{2}(e^{\lambda/2} + e^{-\lambda/2}) = \cosh(\lambda/2)$. The MGF is

finite for every λ whenever X is bounded, but it is infinite for every $\lambda > 0$ when the right tail decays only polynomially, as with a Student-t variable; for such heavy-tailed variables, the exponential method below does not apply.

Applying Markov's inequality to the nonnegative random variable $e^{\lambda X}$ gives the bound in one line. Because $x \mapsto e^{\lambda x}$ is strictly increasing for $\lambda > 0$, the event $\{X \geq t\}$ is identical to $\{e^{\lambda X} \geq e^{\lambda t}\}$, so

$$P(X \geq t) = P(e^{\lambda X} \geq e^{\lambda t}) \leq e^{-\lambda t}\,\mathbb{E}\big[e^{\lambda X}\big], \qquad \text{hence} \qquad P(X \geq t) \leq \inf_{\lambda > 0} e^{-\lambda t}\,\mathbb{E}\big[e^{\lambda X}\big]. \tag{36.112}$$

Every choice of $\lambda > 0$ yields a valid upper bound; taking the infimum simply keeps the tightest one. This two-step recipe is the *Chernoff method*.

There is a second, equally decisive reason why the exponential is the right majorant. When $S = X_1 + \cdots + X_n$ is a sum of independent random variables, $e^{\lambda S} = \prod_{i=1}^{n} e^{\lambda X_i}$ becomes a product of independent factors, and the expectation of a product of independent variables is the product of their expectations:

$$\mathbb{E}\big[e^{\lambda S}\big] = \prod_{i=1}^{n} \mathbb{E}\big[e^{\lambda X_i}\big].$$

Thus the moment generating function of a sum factorizes into the product of the MGFs of its terms, so the logarithm of the Chernoff bound splits into a sum of n pieces and scales linearly with n. No polynomial majorant enjoys this property: $\mathbb{E}[S^4]$ does not factor into a product of fourth moments. All that remains is to bound the MGF of a single summand.

36.9.1.3 The Fair Coin

Let us work through the fair coin all the way to the end, since this simplest case already contains every step of the general proof. A centered flip $Z = X - \frac{1}{2}$ takes the values $\pm\frac{1}{2}$ with equal probability, so its MGF is $\mathbb{E}[e^{\lambda Z}] = \cosh(\lambda/2)$. Comparing the Taylor series of $\cosh u$ and $e^{u^2/2}$ term by term gives

$$\cosh u = \sum_{k \geq 0} \frac{u^{2k}}{(2k)!} \leq \sum_{k \geq 0} \frac{(u^2/2)^k}{k!} = e^{u^2/2},$$

because $(2k)! \geq 2 \cdot 4 \cdots (2k) = 2^k k!$: the even factors of $(2k)!$ alone already produce $2^k k!$, and every odd factor is at least 1. Setting $u = \lambda/2$, we see that a single centered flip satisfies $\mathbb{E}[e^{\lambda Z}] \leq e^{\lambda^2/8}$ for every λ. By comparison, a Gaussian $\mathcal{N}(0, \sigma^2)$ has $\mathbb{E}[e^{\lambda X}] = e^{\lambda^2 \sigma^2/2}$ (found by completing the square in the integral), so a centered coin's MGF is never larger than that of a Gaussian with standard deviation $\frac{1}{2}$.

Now consider n independent flips. The event $\hat{p} - \frac{1}{2} \geq t$ is the same as $\sum_{i=1}^{n} Z_i \geq nt$, so the Chernoff method and the product rule give

$$P(\hat{p} - \tfrac{1}{2} \geq t) \leq e^{-\lambda n t} \prod_{i=1}^{n} \mathbb{E}\big[e^{\lambda Z_i}\big] \leq \exp\Big(-\lambda n t + \frac{n\lambda^2}{8}\Big).$$

The exponent is a simple parabola in λ. Panel (b) of Figure 36.43 plots it for $n = 100$ and $t = 0.1$, where it equals $-10\lambda + 12.5\lambda^2$. Any $\lambda > 0$ gives a valid bound, for example $e^{-0.875} \approx 0.42$ at $\lambda = 0.1$. The tightest bound sits at the vertex of the parabola, $\lambda = 4t$, where the exponent simplifies to $-4nt^2 + 2nt^2 = -2nt^2$. By symmetry the lower tail obeys the same bound, and adding the two tails yields

$$P(|\hat{p} - \tfrac{1}{2}| \geq t) \leq 2e^{-2nt^2}.$$

The probability of straying by t now decays exponentially in nt^2. At $n = 100$ and $t = 0.1$ this gives $2e^{-2} \approx 0.27$, and at $n = 1{,}000$ it drops to $2e^{-20} \approx 4.1 \times 10^{-9}$, a dramatic improvement over Chebyshev's 0.025.

Why does the exponential majorant work so well? At the threshold $S = nt$, the function $e^{\lambda(S-nt)}$ equals 1, while at the mean $S = 0$, where almost all the probability mass sits, it shrinks to $e^{-\lambda nt}$, exponentially small in n. What prevents us from pushing λ to infinity is that the function grows steeply for $S > 0$, so upward fluctuations carry disproportionate weight. The product of the MGFs, bounded by $e^{n\lambda^2/8}$, measures that penalty, and the optimal $\lambda = 4t$ strikes the exact balance between the two.

Notice that the entire derivation relied on just one property of the coin: the MGF of a centered flip is bounded by $e^{\lambda^2/8}$, the MGF of a Gaussian with standard deviation $\frac{1}{2}$, which is half the coin's range. Hoeffding's lemma generalizes this single step, showing that *every* random variable confined to an interval $[a, b]$ obeys the same bound with $(b-a)/2$ in place of $\frac{1}{2}$.

36.9.1.4 Hoeffding's Lemma and Hoeffding's Inequality

A test-set loss is rarely a fair coin. An error rate of 8.3% behaves like a heavily biased coin, and a clipped loss takes a continuum of values rather than just 0 and 1. We would like the fair coin's exponential bound to hold for *any* bounded random variable. The lemma that delivers this is the main technical step of the section, and its proof rests on one elementary fact about variances.

Proposition (variance of a bounded variable). *If Y takes values in $[a, b]$, then $\mathrm{Var}(Y) \leq (b-a)^2/4$, with equality when Y takes the values a and b with probability $\frac{1}{2}$ each.*

Proof. The mean minimizes the expected squared deviation, since $\mathbb{E}[(Y-c)^2] = \mathrm{Var}(Y) + (\mathbb{E}[Y] - c)^2$ for every choice of center c. Taking $c = (a+b)/2$, every point of $[a, b]$ lies within $(b-a)/2$ of the midpoint, so

$$\mathrm{Var}(Y) \leq \mathbb{E}\Big[\Big(Y - \frac{a+b}{2}\Big)^2\Big] \leq \Big(\frac{b-a}{2}\Big)^2.$$

Both inequalities become equalities when Y places equal mass at the two endpoints a and b. ■

The fair coin on $[0, 1]$ is precisely this extreme case, with variance $\frac{1}{4}$. With this variance bound in hand, we can prove Hoeffding's lemma.

Proposition (Hoeffding's lemma). *Let X be a random variable with $\mathbb{E}[X] = 0$ taking values in $[a, b]$. Then for every $\lambda \in \mathbb{R}$,*

$$\mathbb{E}[e^{\lambda X}] \le \exp\!\left(\frac{\lambda^2 (b-a)^2}{8}\right). \tag{36.113}$$

Proof. We want to show that the log-MGF $\psi(\lambda) = \log \mathbb{E}[e^{\lambda X}]$ grows no faster than the parabola $\lambda^2 (b-a)^2/8$. Because X is bounded, ψ is finite and smooth everywhere, allowing us to differentiate under the expectation. At $\lambda = 0$, we have $\psi(0) = \log 1 = 0$ and

$$\psi'(\lambda) = \frac{\mathbb{E}[X e^{\lambda X}]}{\mathbb{E}[e^{\lambda X}]}, \qquad \text{so} \qquad \psi'(0) = \mathbb{E}[X] = 0.$$

Just as for the log-partition function of an exponential family (Section 36.4), the second derivative of ψ is a variance. Define the *tilted* distribution q_λ by reweighting the distribution of X by $e^{\lambda x}$ and normalizing. Under q_λ, $\psi'(\lambda)$ is the mean of X, and differentiating once more gives

$$\psi''(\lambda) = \frac{\mathbb{E}[X^2 e^{\lambda X}]}{\mathbb{E}[e^{\lambda X}]} - \left(\frac{\mathbb{E}[X e^{\lambda X}]}{\mathbb{E}[e^{\lambda X}]}\right)^2 = \mathrm{Var}_{q_\lambda}(X).$$

Crucially, exponential tilting only reweights points; it never moves mass outside $[a, b]$. Our variance bound therefore holds for q_λ at every λ, giving $\psi''(\lambda) \le (b-a)^2/4$. Taylor's theorem with the Lagrange remainder now finishes the proof: for some ξ between 0 and λ,

$$\psi(\lambda) = \psi(0) + \lambda\,\psi'(0) + \frac{\lambda^2}{2}\,\psi''(\xi) \le \frac{\lambda^2 (b-a)^2}{8},$$

and exponentiating both sides yields Equation 36.113. ■

In words, any bounded, centered random variable has a moment generating function no larger than that of a Gaussian with standard deviation $(b-a)/2$. For a centered coin, $b-a = 1$, so the lemma immediately recovers the bound $\cosh(\lambda/2) \le e^{\lambda^2/8}$ that we proved by Taylor series. With that single ingredient in place, our coin proof carries over verbatim to any average of independent bounded variables.

Proposition (Hoeffding's inequality). *Let $X_1, \dots, X_n$ be independent random variables with $X_i \in [a, b]$, and let $\bar{X} = \frac{1}{n}\sum_i X_i$. Then for every $t > 0$,*

$$P\big(|\bar{X} - \mathbb{E}[\bar{X}]| \ge t\big) \le 2\exp\!\left(-\frac{2nt^2}{(b-a)^2}\right). \tag{36.114}$$

Proof. Let $Z_i = X_i - \mathbb{E}[X_i]$. Each Z_i has mean zero and lies in an interval of length $b - a$, so Hoeffding's lemma applies to every term. For any $\lambda > 0$, the Chernoff method Equation 36.112 and the product rule give

$$\begin{aligned} P(\bar{X} - \mathbb{E}[\bar{X}] \ge t) &= P\Big(\sum_i Z_i \ge nt\Big) \le e^{-\lambda n t} \prod_{i=1}^{n} \mathbb{E}[e^{\lambda Z_i}] \\ &\le \exp\!\left(-\lambda n t + \frac{n\lambda^2 (b-a)^2}{8}\right). \end{aligned}$$

This is the same parabola we minimized for the coin, with $(b-a)^2$ in place of 1. Its minimum occurs at $\lambda = 4t/(b-a)^2$, where the exponent equals $-2nt^2/(b-a)^2$. Applying the same argument to $-Z_i$ bounds the lower tail by the same quantity, and adding the two tails produces the factor of 2. $\blacksquare$

Thus the sample mean of n independent bounded variables strays from its expectation by t with a probability that decays exponentially in nt^2, where t is measured in units of the range $b-a$. Notice what the bound does *not* require: no Gaussianity, no variance estimate, and no asymptotic limit. It asks only for boundedness and independence, and it holds at every finite n. Independence, however, is a genuine requirement: the bound does not apply to the correlated samples of a Markov chain.

Look back at the proof and you will notice we never actually needed all the intervals to have the same length. If the independent centered variables satisfy $Y_i \in [a_i, b_i]$, the product of Hoeffding's lemma bounds replaces $n(b-a)^2$ by $\sum_i (b_i - a_i)^2$, and the same quadratic minimization gives

$$P\Big(\Big|\sum_{i} Y_i\Big| \ge s\Big) \le 2\exp\Big(-\frac{2s^2}{\sum_i (b_i-a_i)^2}\Big). \tag{36.115}$$

We will put this unequal-range version to work shortly when we analyze random directions in high dimensions.

Let us translate Hoeffding's inequality into a practical test-set size. For losses in $[0,1]$, we have $b-a=1$, so the failure probability is at most δ once $n \ge \log(2/\delta)/(2t^2)$. To estimate an error rate within $t = 0.01$ with failure probability $\delta = 0.05$, we need $n \ge \log 40/(2\cdot 10^{-4}) = 18{,}444.4$, or $18{,}445$ test examples. This is the figure quoted in Chapter 3.5, roughly twice the 9,604 examples suggested by the asymptotic Gaussian approximation.

Equivalently, setting the right-hand side of Equation 36.114 equal to δ and solving for t gives the confidence-interval form used in practice: with probability at least $1-\delta$,

$$\big|\bar X - \mathbb{E}[\bar X]\big| \le (b-a)\sqrt{\frac{\log(2/\delta)}{2n}}. \tag{36.116}$$

Just like the Gaussian interval $\hat\mu \pm 1.96\,\hat\sigma/\sqrt{n}$, the half-width shrinks at the $1/\sqrt{n}$ rate, but it comes with a strict finite-sample guarantee.

Returning to our classifier with 830 errors on 10,000 images at $\delta = 0.05$, Hoeffding gives $\pm 1.36\%$, compared with the Gaussian interval $\pm 1.96\sqrt{0.083\cdot 0.917/10^4} = \pm 0.54\%$. The two differ only in their constants: Hoeffding replaces the empirical standard deviation $\hat\sigma = 0.28$ by its worst-case value $(b-a)/2 = 0.5$, and replaces the Gaussian quantile 1.96 by $\sqrt{2\log(2/\delta)} = 2.72$, making the interval 2.5 times wider. Whenever the actual spread of the data is much smaller than their range, Hoeffding is inevitably conservative, a gap that Bernstein's inequality will close below.

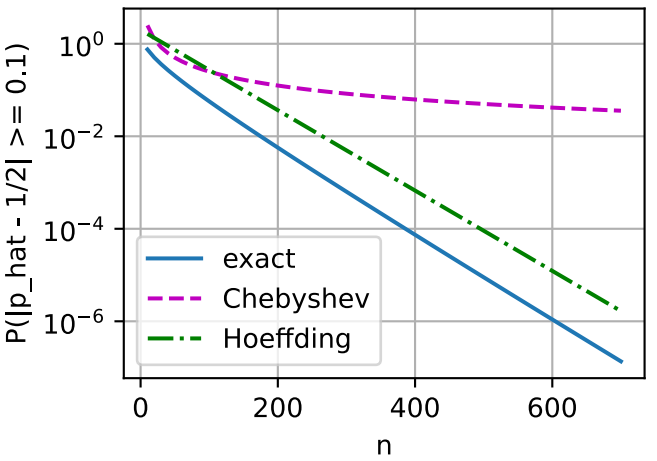
One might wonder whether the constant $\tfrac{1}{8}$ in Hoeffding's lemma can be sharpened. It cannot: a variable placing probability $\tfrac{1}{2}$ at each endpoint of $[a,b]$ attains $\tfrac{1}{8}$ to second order in λ (Exercise 1).

36.9.1.5 The Fair Coin in Numbers

Let us see how close Hoeffding's bound $2e^{-2nt^2}$ comes to the exact tail of the fair coin at $t = 0.1$. We print both alongside their ratio, and plot them together with Chebyshev's bound against n on a logarithmic scale.

```
print(f'{"n":>6} {"exact":>10} {"Hoeffding":>10} {"ratio":>8}')
for n in (10, 100, 1000, 5000):
    hoeffding = 2 * np.exp(-2 * n * t**2)
    print(f'{n:6d} {exact_tail(n, t):10.2e} {hoeffding:10.2e} '
          f'{hoeffding / exact_tail(n, t):8.1f}')

ns = np.arange(10, 701, 10)
d2l.plot(ns, [np.array([exact_tail(n, t) for n in ns]),
              0.25 / (ns * t**2), 2 * np.exp(-2 * ns * t**2)],
         'n', 'P(|p_hat - 1/2| >= 0.1)',
         legend=['exact', 'Chebyshev', 'Hoeffding'], yscale='log')
```

```
     n      exact  Hoeffding    ratio
    10   7.54e-01   1.64e+00      2.2
   100   5.69e-02   2.71e-01      4.8
  1000   2.73e-10   4.12e-09     15.1
  5000   1.30e-45   7.44e-44     57.2
```



At $n = 10$ both bounds exceed 1 and are uninformative. At $n = 100$, Chebyshev's 0.25 is actually a touch tighter than Hoeffding's 0.27: at small sample sizes the leading constants decide, and Chebyshev benefits from knowing the exact variance $\frac{1}{4}$. Once n grows, however, the exponential rate takes over completely. Hoeffding stays within a factor of 15 of the exact tail at $n = 1{,}000$ and within a factor of 57 at $n = 5{,}000$.

On the logarithmic plot, the exact tail and Hoeffding's bound trace nearly parallel straight lines. The logarithm of Hoeffding's bound drops by $2t^2 = 0.02$ per flip, while the exact tail drops by roughly 0.0201 per flip. That sharper slope is the Kullback–Leibler divergence $D_{\mathrm{KL}}(0.6 \| 0.5) = 0.6 \log 1.2 + 0.4 \log 0.8$ between a biased coin with probability 0.6 and a fair coin (Section 37.3). Applying the Chernoff method with the exact MGF of a Bernoulli variable recovers this exact rate (Exercise 3).

Of course, when n is tiny, no inequality can create evidence that is not there. Recall the coin from the chapter introduction that landed heads 9 times in 13 flips, a deviation of $t = 2.5/13$ from $\frac{1}{2}$. Hoeffding's bound on such a deviation is $2e^{-2 \cdot 13 \cdot (2.5/13)^2} = 2e^{-0.96} \approx 0.77$, which rules out nothing. Even the exact two-sided binomial probability is 0.27: thirteen flips simply cannot rule out a fair coin.

36.9.1.6 Sub-Gaussian and Sub-Exponential Variables

Hoeffding's interval for our 8.3% classifier was 2.5 times wider than the Gaussian interval because it knows only the range of a 0/1 loss and ignores its small variance. *Bernstein's inequality* incorporates both.

Proposition (Bernstein's inequality). *Let $X_1, \dots, X_n$ be independent with $\mathrm{Var}(X_i) \le \sigma^2$ and $|X_i - \mathbb{E}[X_i]| \le M$. Then for every $t > 0$,*

$$P(|\bar{X} - \mathbb{E}[\bar{X}]| \ge t) \le 2 \exp\left(-\frac{nt^2}{2(\sigma^2 + Mt/3)} \right). \tag{36.117}$$

Once again, the proof is the Chernoff method (Boucheron et al. 2013), paired with an MGF bound that tracks both the variance and the range (worked out in Exercise 7). Look at the two regimes in the denominator. When t is small, σ^2 dominates $Mt/3$ and the bound behaves like a Gaussian tail with the *true* variance σ^2. When t is large, the linear term $Mt/3$ takes over and the tail decays only exponentially in t rather than as a Gaussian in t^2.

How much does accounting for the variance help in practice? Consider a classifier with a 2% error rate on $n = 10{,}000$ test examples: each 0/1 loss has variance $\sigma^2 = 0.02 \cdot 0.98$, while its centered deviation satisfies $M = 1$. Setting the right-hand side of Equation 36.117 equal to δ produces a quadratic equation in t, namely $nt^2/(2(\sigma^2 + t/3)) = \log(2/\delta)$. Below, we solve this quadratic at $\delta = 0.05$ for both our 8.3% and 2% error rates alongside Hoeffding and the Gaussian interval, computing both the plug-in estimate (using the observed rate in the variance) and a fully guaranteed version that we explain right after the output.

```
def half_widths(p, n, delta=0.05, z=1.96):
    """Hoeffding, Bernstein (plug-in, guaranteed), Gaussian half-widths."""
    L = np.log(2 / delta)
    hoeffding = np.sqrt(L / (2 * n))
    c = 2 * L / 3  # Bernstein with M = 1: n t^2 - c t - 2 L var = 0
    bernstein = lambda var: (c + np.sqrt(c**2 + 8 * n * L * var)) / (2 * n)
    plug_in = t = bernstein(p * (1 - p))
    for _ in range(20):  # true rate at most p + t: a fixed point in t
        t = bernstein((p + t) * (1 - p - t))
    return hoeffding, plug_in, t, z * np.sqrt(p * (1 - p) / n)
```

Evaluating `half_widths(p, 10000)` at both the 8.3% and 2.0% error rates compares the resulting 95% confidence half-widths across all four methods:

```
for p in (0.083, 0.02):
    hoeffding, plug_in, bernstein, gauss = half_widths(p, 10000)
    print(f'error {p:.1%}: Hoeffding ±{hoeffding:.2%}, Bernstein '
          f'±{plug_in:.2%} (plug-in) ±{bernstein:.2%} (guaranteed), '
```

```
            f'Gaussian ±{gauss:.2%}')
```

```
error 8.3%: Hoeffding ±1.36%, Bernstein ±0.76% (plug-in) ±0.79% (guaranteed), Gaussian ±0.54%
error 2.0%: Hoeffding ±1.36%, Bernstein ±0.39% (plug-in) ±0.43% (guaranteed), Gaussian ±0.27%
```

At a 2% error rate, Bernstein's plug-in half-width shrinks to $\pm 0.39\%$, compared with Hoeffding's $\pm 1.36\%$: roughly three and a half times tighter, and within a factor of 1.4 of the asymptotic Gaussian $\pm 0.27\%$. At 8.3% error the improvement is smaller but still substantial, narrowing $\pm 1.36\%$ down to $\pm 0.76\%$.

Strictly speaking, those two plug-in numbers ($\pm 0.39\%$ and $\pm 0.76\%$) are not yet true finite-sample guarantees. The code substituted the observed rate $\hat{p}$ into the variance $p(1-p)$, whereas Bernstein's inequality requires an upper bound on the *true* variance, the exact same plug-in objection we raised against the Gaussian interval. Fortunately, fixing this is inexpensive. Whenever the bound holds, the true error rate is at most $\hat{p} + t$, so we simply solve the same equation with the worst-case variance $(\hat{p}+t)(1-\hat{p}-t)$. Iterating this fixed-point equation (as the cell does) nudges the half-widths only slightly, to the fully guaranteed $\pm 0.43\%$ and $\pm 0.79\%$. This gives a sharp answer to our opening question: for the classifier with 830 errors on 10,000 images, its true error rate lies within about 0.8% of 8.3% with 95% confidence. Because practical models typically operate at error rates of a few percent, variance-aware bounds make a real difference.

Stepping back, Hoeffding's proof relied on just one property of the summands: their moment generating functions were bounded by that of a Gaussian. Naming this property lets us extend the same proof to unbounded random variables, and clarifies exactly where it stops working. A random variable X is **sub-Gaussian** with *variance proxy* σ^2 if its centered MGF never exceeds that of $\mathcal{N}(0, \sigma^2)$:

$$\mathbb{E}\big[e^{\lambda(X-\mathbb{E}[X])}\big] \le \exp\left(\frac{\lambda^2\sigma^2}{2}\right) \qquad \text{for all } \lambda \in \mathbb{R}. \tag{36.118}$$

Any sub-Gaussian variable has tails at least as light as a Gaussian's: applying the Chernoff method with $\lambda = t/\sigma^2$ yields $P(|X - \mathbb{E}[X]| \ge t) \le 2e^{-t^2/(2\sigma^2)}$. Keep in mind that σ^2 here is a *proxy*, not necessarily the true variance. Comparing both sides of Equation 36.118 to second order in λ shows that the proxy is never smaller than $\mathrm{Var}(X)$, and it can be far larger: a 0/1 outcome with 2% error has variance 0.0196, yet Hoeffding's lemma supplies the proxy $\frac{1}{4}$, which is precisely why Hoeffding's interval was so wide.

Three standard families are sub-Gaussian. A Gaussian $\mathcal{N}(\mu, \sigma^2)$ has proxy σ^2 with equality; any variable on $[a, b]$ has proxy $(b-a)^2/4$ by Hoeffding's lemma; and a uniform random sign $\xi \in \{-1, +1\}$ has proxy 1, since $\mathbb{E}[e^{\lambda\xi}] = \cosh\lambda \le e^{\lambda^2/2}$ by our earlier series comparison. Best of all, sub-Gaussianity is preserved by averaging: if $X_1, \dots, X_n$ are independent with proxy σ^2, the product rule gives $\bar{X}$ the proxy σ^2/n, and therefore $P(|\bar{X} - \mathbb{E}\bar{X}| \ge t) \le 2e^{-nt^2/(2\sigma^2)}$.

Not every well-behaved random variable is sub-Gaussian, however, and the very next quantity we need is not. The squared length of a Gaussian vector is a sum of squares of standard Gaussians, and for a single standard Gaussian x, completing the square gives

$$\mathbb{E}\big[e^{\lambda(x^2-1)}\big] = \frac{e^{-\lambda}}{\sqrt{1-2\lambda}} \qquad \text{for } \lambda < \tfrac{1}{2},$$

while the expectation diverges to infinity for every $\lambda \geq \frac{1}{2}$. Because the MGF blows up at $\lambda = \frac{1}{2}$, Equation 36.118 cannot hold for x^2 at any variance proxy: its right tail decays like $e^{-t/2}$, too slowly for a Gaussian bound.

Even so, a Gaussian-shaped MGF bound *does* hold on a restricted window around $\lambda = 0$. We say that a random variable X is **sub-exponential** with parameters (ν, α) if

$$\mathbb{E}\big[e^{\lambda(X-\mathbb{E}[X])}\big] \leq \exp\left(\frac{\lambda^2\nu^2}{2}\right) \qquad \text{for } |\lambda| \leq \frac{1}{\alpha}. \tag{36.119}$$

Here ν^2 acts as a local variance proxy near $\lambda = 0$, while $1/\alpha$ sets the half-width of the admissible window, capping how large a λ the Chernoff method may choose. In Section 36.9.2.1 we will see that x^2 is sub-exponential with $\nu = 2$ and $\alpha = 4$. When we apply the Chernoff method to an average of sub-exponential variables, the optimization naturally splits at the boundary of this window. You may skip the proof on a first reading; the next subsection uses only the statement.

Proposition (sub-exponential tail bound). *Let $X_1, \dots, X_n$ be independent and (ν, α)-sub-exponential. Then for every $t > 0$,*

$$P\big(|\bar{X} - \mathbb{E}[\bar{X}]| \geq t\big) \leq 2\exp\left(-\frac{n}{2}\min\Big(\frac{t^2}{\nu^2}, \frac{t}{\alpha}\Big)\right). \tag{36.120}$$

Proof. For any $0 < \lambda \leq 1/\alpha$, the Chernoff method and the product rule bound $P(\bar{X} - \mathbb{E}[\bar{X}] \geq t)$ by $\exp(-\lambda nt + n\lambda^2\nu^2/2)$. When $t \leq \nu^2/\alpha$, the unconstrained minimizer $\lambda = t/\nu^2$ falls inside the window and yields $e^{-nt^2/(2\nu^2)}$. When $t > \nu^2/\alpha$, that minimizer lies outside the window, so we clamp λ to the largest admissible value, $\lambda = 1/\alpha$. The exponent becomes $-nt/\alpha + n\nu^2/(2\alpha^2)$, which is at most $-nt/\alpha + nt/(2\alpha) = -nt/(2\alpha)$ because $\nu^2/\alpha < t$. Combining both regimes gives an exponent of at most $-\frac{n}{2}\min(t^2/\nu^2, t/\alpha)$. The lower tail follows identically using $\lambda \in [-1/\alpha, 0)$. ■

The contrast with sub-Gaussian averages is now crystal clear. Averages of bounded or sub-Gaussian variables enjoy Gaussian-type tails e^{-ct^2} at every scale. A sub-exponential average has that same Gaussian tail for moderate deviations below ν^2/α, and then transitions to a slower exponential tail $e^{-nt/(2\alpha)}$ farther out.

36.9.2 Probability in High Dimension

Up to now, our bounds have protected a single sample average. Now consider a random vector in $\mathbb{R}^d$ with d independent coordinates: both its squared length and its inner products with other vectors are sums over those d coordinates. Concentration inequalities therefore govern the geometry of high-dimensional space. If you are jumping in here, you only need two results from the first half of the section: the sub-exponential tail bound Equation 36.120 for squared lengths, and the unequal-range Hoeffding bound Equation 36.115 for inner products.

36.9.2.1 The Thin Shell

Draw a standard Gaussian vector $\mathbf{x} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}_{784})$, which has as many coordinates as a 28×28 image. Its probability density is highest at the origin. Surprisingly, however, the probability that $\mathbf{x}$ lands within distance 14 of the origin (half its typical length) is roughly 10^{-110}, the chance that a sum of 784 squared standard Gaussians is at most 196. No sample you could ever draw on a computer will contain a point anywhere near the mode. Instead, a typical draw has norm 28.0 ± 0.7.

The explanation is that $\|\mathbf{x}\|^2 = \sum_{i=1}^{d} x_i^2$ is a sum of d independent terms, each with mean 1, so $\|\mathbf{x}\|^2/d$ is simply a sample mean. Because x_i^2 is unbounded, Hoeffding's inequality does not apply, but our sub-exponential bound handles it cleanly.

Proposition (norm concentration). *Let $\mathbf{x} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}_d)$. For every $0 < \varepsilon \le 1$,*

$$P\left(\left| \frac{\|\mathbf{x}\|}{\sqrt{d}} - 1 \right| \ge \varepsilon \right) \le 2 \exp\left(-\frac{d\,\varepsilon^2}{8} \right). \tag{36.121}$$

Proof. The argument proceeds in three short steps. First, a power-series comparison (Exercise 4) shows that the MGF of a centered square, $e^{-\lambda}/\sqrt{1-2\lambda}$, satisfies

$$\mathbb{E}\big[e^{\lambda(x_i^2 - 1)}\big] \le e^{2\lambda^2} \qquad \text{for } |\lambda| \le \tfrac{1}{4}. \tag{36.122}$$

In the terminology of Equation 36.119, x_i^2 is sub-exponential with $\nu = 2$ and $\alpha = 4$. Second, applying the sub-exponential tail bound Equation 36.120 with $n = d$ and $t = \varepsilon$ gives

$$P\Big(\Big| \frac{\|\mathbf{x}\|^2}{d} - 1 \Big| \ge \varepsilon \Big) \le 2 \exp\Big(-\frac{d}{2} \min\Big(\frac{\varepsilon^2}{4}, \frac{\varepsilon}{4} \Big) \Big) = 2e^{-d\varepsilon^2/8} \qquad \text{for } \varepsilon \le 1.$$

Third, we pass from the squared norm to the norm itself. Let $z = \|\mathbf{x}\|/\sqrt{d} \ge 0$. Whenever $|z - 1| \ge \varepsilon$, we also have $|z^2 - 1| = |z-1|\,(z+1) \ge \varepsilon$, since $z + 1 \ge 1$. Thus the event for the norm is contained in the event for the squared norm and inherits its upper bound. ∎

In short, the probability that the length of a standard Gaussian vector misses $\sqrt{d}$ by a fixed fraction decays exponentially in the dimension d. Note that the restriction $\varepsilon \le 1$ is an artifact of bounding the *squared* norm, whose upper tail beyond $\varepsilon = 1$ decays only exponentially as $e^{-d\varepsilon/8}$ (Exercise 4). The norm itself keeps a Gaussian tail at every scale: exceeding $(1+\varepsilon)\sqrt{d}$ forces the squared norm above $(1 + 2\varepsilon + \varepsilon^2)d$, whose probability is at most $e^{-d(2\varepsilon+\varepsilon^2)/8} \le e^{-d\varepsilon^2/8}$.

How thick is this shell of typical norms? Since $\mathbb{E}[x^4] = 3$, each coordinate square has $\mathrm{Var}(x^2) = 3 - 1 = 2$, giving $\mathrm{Var}(\|\mathbf{x}\|^2) = 2d$. Linearizing the square root around d gives $\|\mathbf{x}\| \approx \sqrt{d} + (\|\mathbf{x}\|^2 - d)/(2\sqrt{d})$, so the standard deviation of $\|\mathbf{x}\|$ is approximately $\sqrt{2d}/(2\sqrt{d}) = 1/\sqrt{2} \approx 0.71$, independent of the dimension. Remarkably, a standard Gaussian in $\mathbb{R}^d$ concentrates on a thin shell of radius $\sqrt{d}$ whose thickness stays of order one no matter how large d becomes.

Why does all the probability mass sit so far from the mode? The joint density $(2\pi)^{-d/2} e^{-\|\mathbf{x}\|^2/2}$ depends on $\mathbf{x}$ solely through its radius $r = \|\mathbf{x}\|$, while the surface area

of a sphere of radius r grows like r^{d-1}. Multiplying density by surface area, the distribution of the radius $\|\mathbf{x}\|$ is proportional to $r^{d-1}e^{-r^2/2}$. As r grows, the volume factor r^{d-1} explodes while the density $e^{-r^2/2}$ plummets; their product peaks where the derivative of $(d-1)\log r - r^2/2$ vanishes, namely at $r = \sqrt{d-1}$. Panel (a) of Figure 36.44 illustrates this competition between volume and density at $d = 20$.

At a typical point of norm $\sqrt{d}$, the pointwise density is $e^{-d/2}$ times its value at the origin (a tiny 10^{-170} when $d = 784$), yet there is so much more surface area at radius $\sqrt{d}$ that virtually all the mass lives there. Panel (b) displays the exact density of $\|\mathbf{x}\|$ from $d = 2$ to $d = 512$: the peaks march outward like $\sqrt{d}$ while their widths stay pinned near $1/\sqrt{2}$.

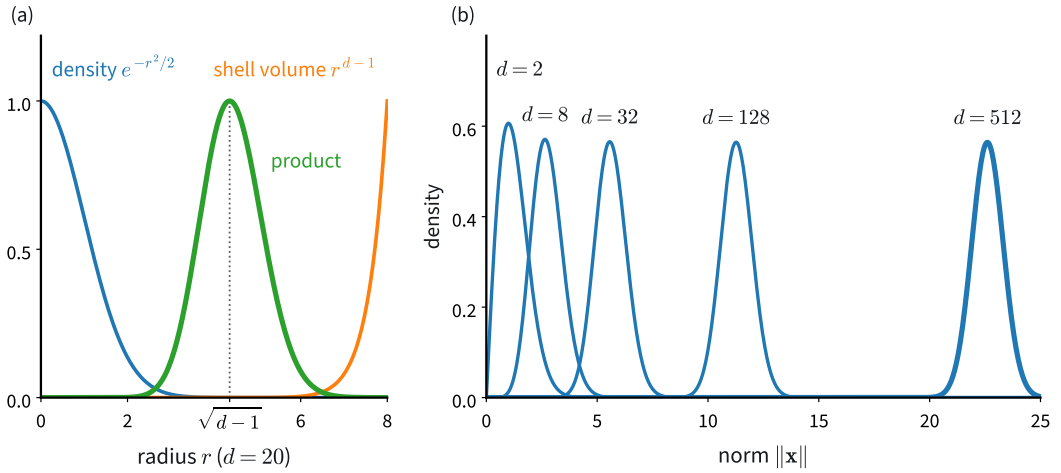


Figure 36.44: The thin shell. (a) At $d = 20$ the Gaussian density $e^{-r^2/2}$ falls with the radius r, the volume r^{d-1} of the shell at radius r grows, and their product, the density of $\|\mathbf{x}\|$, peaks at $\sqrt{d-1}$, far from the origin where the density is largest; each curve is scaled to maximum 1. (b) The exact density of $\|\mathbf{x}\|$ for $\mathbf{x} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}_d)$ and $d = 2, 8, 32, 128, 512$. The centers move out like $\sqrt{d}$; the widths stay near $1/\sqrt{2}$.

Notice that our bound captures the exponential scaling in d, though not the exact numerical value. A norm below 14 at $d = 784$ corresponds to $\varepsilon = \frac{1}{2}$, for which Equation 36.121 gives $2e^{-784/32} = 2e^{-24.5} \approx 5 \times 10^{-11}$, compared with the exact 10^{-110}. The constant 8 in the denominator is loose, but the qualitative conclusion, exponential decay in d, is right on target.

36.9.2.2 Near-Orthogonality of Random Directions

In Section 33.1, we saw that the cosine of the angle between a fixed unit vector and a random direction in $\mathbb{R}^d$ has variance $1/d$. Paired with Chebyshev's inequality, that gave a weak tail of order $1/(d\varepsilon^2)$. Concentration sharpens this to an exponential tail, which in turn lets us count how many random directions can be simultaneously nearly orthogonal.

Random sign vectors provide the cleanest calculation. Fix any unit vector $\mathbf{u}$ and draw independent uniform signs $x_i \in \{-1, +1\}$. Each term $u_i x_i$ of the inner product $\langle \mathbf{x}, \mathbf{u} \rangle$ has mean zero and lies in $[-|u_i|, |u_i|]$, an interval of length $2|u_i|$. Substituting these ranges into our

unequal-range Hoeffding bound Equation 36.115, the denominator becomes $4\sum_i u_i^2 = 4$, leaving

$$P(|\langle \mathbf{x}, \mathbf{u}\rangle| \geq s) \leq 2e^{-s^2/2}. \tag{36.123}$$

The inner product of a random sign vector with any fixed unit vector stays of order one with a Gaussian tail, because the $\pm|u_i|$ contributions largely cancel. Meanwhile, the norm of $\mathbf{x}$ is $\sqrt{d}$. Setting $s = \varepsilon\sqrt{d}$ immediately gives $P(|\cos(\mathbf{x}, \mathbf{u})| \geq \varepsilon) \leq 2e^{-d\varepsilon^2/2}$. For a standard Gaussian vector $\mathbf{x}$, the same behavior holds: the inner product $\langle \mathbf{x}, \mathbf{u}\rangle$ is itself a standard Gaussian for every unit vector $\mathbf{u}$, so it obeys the exact same tail $2e^{-s^2/2}$, and the thin shell keeps $\|\mathbf{x}\|$ within a few percent of $\sqrt{d}$.

With an exponential tail in hand, we can control *many* pairs at once using one extra tool: the *union bound*, which states that the probability that at least one of several events occurs is at most the sum of their individual probabilities, regardless of whether those events are independent. Take m independent random sign vectors. They form $m(m-1)/2 < m^2/2$ distinct pairs, and any single pair has $|\cos| \geq \varepsilon$ with probability at most $2e^{-d\varepsilon^2/2}$. By the union bound, the probability that *any* pair exceeds ε is at most $m^2 e^{-d\varepsilon^2/2}$. Consequently, with probability at least $1-\delta$, all pairwise cosines stay below ε whenever $m \leq \sqrt{\delta}\, e^{d\varepsilon^2/4}$. In high dimensions, you can pack *exponentially* many nearly orthogonal vectors into $\mathbb{R}^d$.

Let us plug in $d = 784$ and $\delta = 0.05$. Even this loose bound guarantees roughly 570 directions with all pairwise $|\cos| < 0.2$, and roughly 10^7 directions with all pairwise $|\cos| < 0.3$. And those are worst-case guarantees over all pairs; a single typical pair has $|\cos|$ much smaller, around $\sqrt{2/(\pi d)} \approx 0.028$, the mean absolute value of a Gaussian with standard deviation $1/\sqrt{d}$.

Of course, learned neural network embeddings are not independent random vectors, and that is precisely why this calculation is useful. At $d = 784$, a cosine similarity of 0.3 sits more than eight standard deviations ($1/\sqrt{784} = 0.036$) above what independent random directions produce. Random directions thus serve as the null model against which genuine geometric structure in trained embeddings is measured.

36.9.2.3 The Thin Shell in Numbers

Because our theoretical bounds have loose constants, let us measure the true numbers directly. Our tangent-line argument predicts that the standard deviation of $\|\mathbf{x}\|$ stays near $1/\sqrt{2} \approx 0.71$ at every dimension d, while the Gaussian approximation predicts a mean $|\cos|$ of $\sqrt{2/(\pi d)}$. Below, we draw 10,000 pairs of standard Gaussian vectors at $d = 2, 20, 200, 2000$ to test both predictions; notice that one row will not match.

```
for d in (2, 20, 200, 2000):
    x, y = rng.standard_normal((2, 10000, d))
    nx = np.linalg.norm(x, axis=1)
    cos = (x * y).sum(1) / (nx * np.linalg.norm(y, axis=1))
    print(f'd = {d:4d}: mean ||x||/sqrt(d) = {(nx / np.sqrt(d)).mean():.3f}, '
          f'sd ||x|| = {nx.std():.2f}, mean |cos| = {np.abs(cos).mean():.4f}'
          f' vs {np.sqrt(2 / (np.pi * d)):.4f}')
```

```
d =    2: mean ||x||/sqrt(d) = 0.886, sd ||x|| = 0.65, mean |cos| = 0.6377 vs 0.5642
d =   20: mean ||x||/sqrt(d) = 0.986, sd ||x|| = 0.71, mean |cos| = 0.1827 vs 0.1784
d =  200: mean ||x||/sqrt(d) = 0.999, sd ||x|| = 0.70, mean |cos| = 0.0560 vs 0.0564
d = 2000: mean ||x||/sqrt(d) = 1.000, sd ||x|| = 0.71, mean |cos| = 0.0179 vs 0.0178
```

As predicted, the standard deviation of the norm sits near 0.65 at $d = 2$ and settles right around 0.7 thereafter, even as the norm itself climbs from roughly 1.3 to 45. The one row that does not match the asymptotic formula is $d = 2$, where the mean $|\cos|$ is about 0.64: in the plane, the angle between two isotropic vectors is uniform, giving $\mathbb{E}|\cos\theta| = 2/\pi$, whereas $\sqrt{2/(\pi d)}$ is a high-dimensional approximation. From $d = 20$ onward, the measured values (about 0.18, 0.057, and 0.018) agree with the formula to within a few percent.

Two follow-up questions make the geometry even more vivid. First, what fraction of the probability mass lies within 10% of the typical norm, inside the shell $(1 \pm 0.1)\sqrt{d}$? Second, do Euclidean distances still separate nearby points from distant ones? To measure separation, we compute the *contrast* of a query point, $(d_{\text{max}} - d_{\text{min}})/d_{\text{min}}$ across its distances to 200 random points, and report the median contrast over 100 queries.

```
eps = 0.1
for d in (2, 20, 200, 2000):
    x = rng.standard_normal((10000, d))
    ratio = np.linalg.norm(x, axis=1) / np.sqrt(d)
    shell = np.mean(np.abs(ratio - 1) <= eps)          # mass in the 10% shell
    query, points = x[:100], x[100:300]                # 100 queries, 200 points
    dist = np.linalg.norm(query[:, None] - points[None], axis=2)  # (100, 200)
    contrast = (dist.max(1) - dist.min(1)) / dist.min(1)
    print(f'd = {d:4d}: mass in (1 ± 0.1) sqrt(d) = {shell:6.1%}, '
          f'median contrast = {np.median(contrast):.3f}')
```

```
d =    2: mass in (1 ± 0.1) sqrt(d) =  14.5%, median contrast = 30.417
d =   20: mass in (1 ± 0.1) sqrt(d) =  46.7%, median contrast = 1.179
d =  200: mass in (1 ± 0.1) sqrt(d) =  95.6%, median contrast = 0.270
d = 2000: mass in (1 ± 0.1) sqrt(d) = 100.0%, median contrast = 0.079
```

At $d = 2$, the 10% shell captures only about 15% of the mass; by $d = 20$ it captures roughly 47%, and by $d = 200$ roughly 95%. That 95% matches our tangent-line estimate: at $d = 200$, the ratio $\|\mathbf{x}\|/\sqrt{d}$ has standard deviation $1/\sqrt{2d} = 0.05$, so ± 0.1 spans two standard deviations on either side of the mean. By $d = 2000$, all 10,000 draws land inside the shell. Meanwhile, the median distance contrast collapses from around 30 at $d = 2$ (fluctuating between roughly 25 and 40 from run to run) down to about 1.2, 0.27, and finally 0.08. At $d = 2000$, for a typical query, the farthest of 200 random points is less than 10% farther away than the nearest.

36.9.2.4 Consequences for Deep Learning

Three standard practices in deep learning rest directly on this high-dimensional geometry.

Initialization. LeCun initialization (LeCun, Leon Bottou, et al. 1998) samples the entries of a $d \times d$ weight matrix independently with variance $1/d$, and He initialization (He et al. 2015) uses variance $2/d$ for ReLU networks (Section 4.4). The classical derivation of these scales works in expectation over all random draws of the weights: at LeCun's scale, each pre-activation $\langle \mathbf{w}, \mathbf{x} \rangle$ has variance $\|\mathbf{x}\|^2/d$, which norm concentration keeps close to 1 for inputs with unit-variance coordinates. In practice, however, we initialize and train a *single*

network, not an average over infinitely many. Concentration tells us when that expectation actually describes our one draw.

For a single He-initialized ReLU layer of width d, the ratio $\rho = \|\mathbf{h}_k\|^2/\|\mathbf{h}_{k-1}\|^2$ of squared activation norms has mean 1 and variance $5/d$ (Exercise 5), so one layer perturbs the squared norm by roughly $\sqrt{5/d}$, or about 10% at $d = 512$. Across K layers, these multiplicative fluctuations add on a logarithmic scale, giving $\log(\|\mathbf{h}_K\|^2/\|\mathbf{h}_0\|^2)$ a variance of roughly $5K/d$. The expected-variance heuristic therefore describes a typical network only while $K \ll d$. At $K = 50$ and $d = 512$, the log-variance reaches about 0.5, meaning the squared norm of a typical initialized network is already off by a factor of two in either direction.

Distances and nearest neighbors. For two independent standard Gaussian draws, the difference satisfies $\mathbf{x} - \mathbf{y} \sim \mathcal{N}(\mathbf{0}, 2\mathbf{I})$, so norm concentration applies directly to $(\mathbf{x} - \mathbf{y})/\sqrt{2}$: each pairwise distance falls in $(1 \pm \varepsilon)\sqrt{2d}$ except with probability $2e^{-d\varepsilon^2/8}$. Applying the same union bound we used for directions places *all* pairwise distances among n points inside that narrow band simultaneously once $d \geq (8/\varepsilon^2)\log(n^2/\delta)$, a threshold that grows only logarithmically with n. When every distance is nearly the same, nearest and farthest neighbors differ by only a small relative margin, just as our contrast experiment showed. Real learned embeddings retain meaningful distance contrast only because real data lie near a much lower-dimensional structure. On the flip side, this same calculation yields a powerful tool: a random projection down to k dimensions preserves all pairwise distances among n points once k is of order $\log n$, regardless of the ambient dimension (Exercise 6).

Latent codes. A generative decoder trained on latent codes $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}_d)$ almost exclusively sees inputs whose norm is close to $\sqrt{d}$. Both the prior mean (norm 0) and the midpoint of the straight chord connecting two independent codes (norm close to $\sqrt{d/2}$) lie deep in the empty interior where the decoder was never trained. For this reason, practitioners typically interpolate between latent codes along a spherical arc rather than a straight line, as Figure 36.45 illustrates (Exercise 2).

One crucial limitation remains. Every bound in this section protects a single quantity fixed *before* the data are drawn. A learning algorithm, by contrast, inspects the data and picks the best model among many candidates *after* the fact, and the best of many fair coins will always look biased. In Section 36.10, we show how to extend concentration inequalities to account for that selection.

36.9.3 Summary

How far can an average of n independent terms stray from its expectation at a finite sample size n? Chebyshev's inequality gives a tail probability that decays only like $1/n$, far too slowly when we need a guarantee that almost never fails. The Chernoff method applies Markov's inequality to $e^{\lambda X}$, turning the moment generating function of an independent sum into a product, and then optimizes over λ. Combined with Hoeffding's lemma, this yields the exponential tail $2e^{-2nt^2}$ for losses in $[0, 1]$ and a finite-sample confidence interval of half-width $\sqrt{\log(2/\delta)/(2n)}$; Bernstein's inequality tightens this interval substantially whenever the variance is small relative to the range. Applied to the d coordinates of a random vector, the same ideas show that a standard Gaussian concentrates on a thin shell of radius $\sqrt{d}$ and thickness about $1/\sqrt{2}$, and that exponentially many random directions are pairwise nearly orthogonal. Together, these results tell us how to size test sets, when initialization heuristics hold for a single trained network, and how to judge cosine similarities and distances in high-dimensional embeddings.

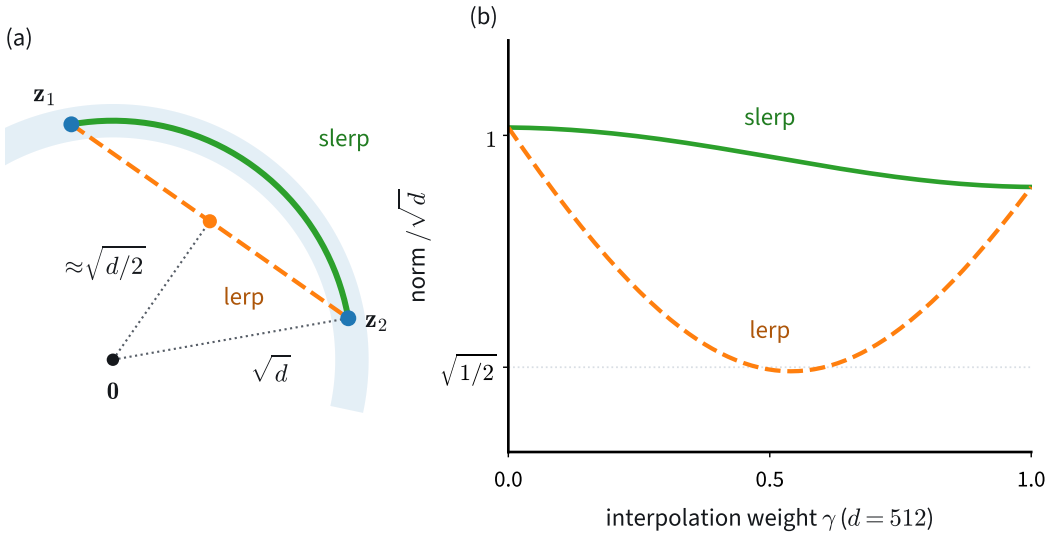


Figure 36.45: Two ways to interpolate between latent codes $\mathbf{z}_1, \mathbf{z}_2 \sim \mathcal{N}(\mathbf{0}, \mathbf{I}_d)$. (a) Both codes lie in the thin shell of radius $\sqrt{d}$ (shaded) and are nearly orthogonal. The straight chord (lerp) cuts through the interior, and its midpoint has norm about $\sqrt{d/2}$; the spherical arc (slerp) stays in the shell. (b) The norm along each path, divided by $\sqrt{d}$, for one random pair at $d = 512$.

36.9.4 Exercises

1. **Sub-Gaussian random signs.** A random sign ξ takes the values -1 and $+1$ with probability $\frac{1}{2}$ each, so that $\mathbb{E}[e^{\lambda\xi}] = \cosh\lambda \le e^{\lambda^2/2}$.
   1. Conclude with the Chernoff method Equation 36.112 that the average $\bar{\xi}$ of n independent random signs satisfies $P(|\bar{\xi}| \ge t) \le 2e^{-nt^2/2}$.
   2. Hoeffding's lemma Equation 36.113 with $[a, b] = [-1, 1]$ also gives a variance proxy for ξ. Compare it with the bound $\cosh\lambda \le e^{\lambda^2/2}$. Expand $\log\cosh\lambda = \lambda^2/2 - \lambda^4/12 + \cdots$ around $\lambda = 0$, and explain why no constant smaller than $\frac{1}{8}$ can replace the $\frac{1}{8}$ in Equation 36.113 for all bounded variables.
2. **Latent interpolation.** A generative model decodes latent codes $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}_d)$ with $d = 512$. Let $\mathbf{z}_1, \mathbf{z}_2$ be two independent codes and $\mathbf{z}(\gamma) = (1-\gamma)\mathbf{z}_1 + \gamma\mathbf{z}_2$ for $\gamma \in [0, 1]$ their linear interpolation.
   1. Find the distribution of $\mathbf{z}(\gamma)$ and the approximate value of $\|\mathbf{z}(\gamma)\|$. At which γ is the norm smallest?
   2. Use Equation 36.121 to bound the probability that the midpoint $\mathbf{z}(\frac{1}{2})$ has norm at least $0.9\sqrt{d}$, that is, reaches the shell $(1 \pm 0.1)\sqrt{d}$ in which typical codes lie. Then compute the exact probability, which is $P(\chi^2_{512} \ge 2 \cdot 0.81 \cdot 512)$ for a chi-square variable with 512 degrees of freedom (`scipy.stats.chi2.sf`). By how many orders of magnitude is the bound loose?
   3. Spherical interpolation (White 2016) replaces the chord by the arc

$$\tilde{\mathbf{z}}(\gamma) = \frac{\sin((1-\gamma)\theta)}{\sin\theta}\,\mathbf{z}_1 + \frac{\sin(\gamma\theta)}{\sin\theta}\,\mathbf{z}_2,$$

where θ is the angle between $\mathbf{z}_1$ and $\mathbf{z}_2$. Show that $\|\tilde{\mathbf{z}}(\gamma)\| = r$ for all γ when $\|\mathbf{z}_1\| = \|\mathbf{z}_2\| = r$. Which value does θ take approximately for $d = 512$, and why?

3. code **Chernoff bound for coins.** ● Let $\hat{p}$ be the fraction of heads in n independent flips of a coin with bias p, and let $a > p$.
   1. Apply the Chernoff method Equation 36.112 to $n\hat{p}$, whose moment generating function is $(1 - p + pe^{\lambda})^n$. Optimize over λ and show that $P(\hat{p} \geq a) \leq e^{-n\, D_{\mathrm{KL}}(a\|p)}$, where $D_{\mathrm{KL}}(a\|p) = a \log \frac{a}{p} + (1-a) \log \frac{1-a}{1-p}$.
   2. Show that $D_{\mathrm{KL}}(a\|p) \geq 2(a-p)^2$ by examining the second derivative in a, so that the bound of (a) is never weaker than one side of Hoeffding's bound. Evaluate both exponents for $p = \frac{1}{2}$ and $a = 0.6$.
   3. Add the two-sided bound $2e^{-n\, D_{\mathrm{KL}}(0.6\|0.5)}$ as a column to the table of Section 36.9.1.5, for $n \in \{10, 100, 1000, 5000\}$. The ratio of the exact tail to this bound still decreases with n. How does it scale with n, and what does the scaling say about the factor that the Chernoff method loses?
4. **Chi-square tail.** ● The proof of Equation 36.121 rests on the bound Equation 36.122 for $x^2 - 1$ with $x \sim \mathcal{N}(0,1)$, which the proof states without derivation.
   1. Show that $\mathbb{E}[e^{\lambda(x^2-1)}] = e^{-\lambda}/\sqrt{1-2\lambda}$ for $\lambda < \frac{1}{2}$ by completing the square in the Gaussian integral.
   2. Expand $\log \mathbb{E}[e^{\lambda(x^2-1)}]$ into a power series in 2λ and prove Equation 36.122. The series starts at the quadratic term.
   3. The proof of Equation 36.121 takes $\lambda = \varepsilon/4$, which is admissible only for $\varepsilon \leq 1$. For $\varepsilon > 1$, set $\lambda = \frac{1}{4}$ and bound $P(\|\mathbf{x}\|^2 \geq (1+\varepsilon)d)$. Does the result decay like $e^{-c\varepsilon^2}$ or like $e^{-c\varepsilon}$ in ε, and which regime of Equation 36.120 does it belong to?
5. code **Activation norms through depth.** ● Consider a ReLU network of width d and depth K without biases, $\mathbf{h}_k = \mathrm{ReLU}(\mathbf{W}_k \mathbf{h}_{k-1})$, whose weights are independent draws from $\mathcal{N}(0, 2/d)$ (He initialization, Section 4.4). Let $\rho_k = \|\mathbf{h}_k\|^2 / \|\mathbf{h}_{k-1}\|^2$. How such norms fluctuate with depth and width was analyzed by Hanin and Nica (2020) .
   1. Show that, given $\mathbf{h}_{k-1} \neq \mathbf{0}$, the ratio ρ_k is distributed as $\frac{2}{d} \sum_{j=1}^{d} \mathrm{ReLU}(g_j)^2$ with independent $g_j \sim \mathcal{N}(0,1)$, whatever the direction of $\mathbf{h}_{k-1}$. Conclude that $\mathbb{E}[\rho_k] = 1$ and $\mathrm{Var}(\rho_k) = 5/d$, using $\mathbb{E}[g^2] = 1$ and $\mathbb{E}[g^4] = 3$.
   2. Write $\log(\|\mathbf{h}_K\|^2 / \|\mathbf{h}_0\|^2)$ as a sum of K independent terms ($\rho_k = 0$ has probability 2^{-d} per layer; ignore it). Expand $\log \rho$ to second order around 1 and predict the mean and variance of this sum in terms of K/d.
   3. Simulate $1{,}000$ networks for each width $d \in \{32, 128\}$ and depth $K \in \{1, 10, 50\}$, each with a fresh standard Gaussian input. Compare the mean and variance of $\log(\|\mathbf{h}_K\|^2 / \|\mathbf{h}_0\|^2)$ with (b), and report the mean and the median of the ratio $\|\mathbf{h}_K\|^2 / \|\mathbf{h}_0\|^2$ itself. The variance calculation of Section 4.4 gives $\mathbb{E}[\|\mathbf{h}_K\|^2 / \|\mathbf{h}_0\|^2] = 1$ at every depth. Repeat the run at $K = 50$ and $d = 32$ several times. Why does the sample mean over $1{,}000$ networks scatter so widely around 1? Why does the expectation not describe a typical network there?
6. code **Random projections preserve distances.** ● Let $\mathbf{A} \in \mathbb{R}^{k \times d}$ have independent $\mathcal{N}(0, 1/k)$ entries.
   1. Show that for a fixed $\mathbf{v} \neq \mathbf{0}$ the ratio $\|\mathbf{A}\mathbf{v}\|^2 / \|\mathbf{v}\|^2$ has the distribution of $\|\mathbf{g}\|^2/k$ with $\mathbf{g} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}_k)$, so that by Equation 36.121 $P(|\|\mathbf{A}\mathbf{v}\|/\|\mathbf{v}\| - 1| \geq \varepsilon) \leq 2e^{-k\varepsilon^2/8}$. Hint: rotate so that $\mathbf{v}$ lies along the first axis.
   2. For n points, apply (a) to the $n(n-1)/2$ differences and a union bound. Show that all pairwise distances are preserved within a factor $1 \pm \varepsilon$ with probability at

least $1-\delta$ once $k \geq (8/\varepsilon^2)\log(n^2/\delta)$, a bound that does not depend on d. This is the Johnson–Lindenstrauss lemma. Evaluate it for $n = 1{,}000$, $\varepsilon = 0.3$ and $\delta = 0.05$.

3. Verify it on $1{,}000$ standard Gaussian points in $d = 2{,}000$. Project to $k \in \{50, 200, 800\}$ and report the largest distortion $|\|\mathbf{A}(\mathbf{x}_i - \mathbf{x}_j)\|/\|\mathbf{x}_i - \mathbf{x}_j\| - 1|$ over all pairs, together with the typical fluctuation $1/\sqrt{2k}$. Compute the pairwise distances from the Gram matrix, $\|\mathbf{x}_i - \mathbf{x}_j\|^2 = \|\mathbf{x}_i\|^2 + \|\mathbf{x}_j\|^2 - 2\,\mathbf{x}_i^\top \mathbf{x}_j$, not by broadcasting a $1{,}000 \times 1{,}000 \times 2{,}000$ array. Explain the ratio between the two numbers from the number of pairs. Already at $k = 200$ the largest distortion is below $\varepsilon = 0.3$, while (b) asked for a much larger k. Which part of the bound is loose: the constant 8 or the $\log n$?

7. **Bernstein's inequality.** ● Let X be centered with $|X| \leq M$ and $\mathrm{Var}(X) = \sigma^2$.
   1. Show that $\mathbb{E}[X^k] \leq \sigma^2 M^{k-2}$ for $k \geq 2$, and hence that $\mathbb{E}[e^{\lambda X}] \leq 1 + \sigma^2(e^{\lambda M} - 1 - \lambda M)/M^2$ for $\lambda > 0$.
   2. Show that $e^u - 1 - u \leq u^2/(2(1-u/3))$ for $0 \leq u < 3$ by comparing $u^k/k!$ with $(u^2/2)(u/3)^{k-2}$, that is, by using $k! \geq 2 \cdot 3^{k-2}$. Conclude with $1 + y \leq e^y$ that $\mathbb{E}[e^{\lambda X}] \leq \exp(\lambda^2\sigma^2/(2(1-\lambda M/3)))$ for $0 < \lambda < 3/M$.
   3. Apply the Chernoff method to an average of n such variables with $\lambda = t/c$, where $c = \sigma^2 + Mt/3$. Check that $\lambda < 3/M$ and that $1 - \lambda M/3 = \sigma^2/c$, and conclude that the exponent is exactly $-nt^2/(2c)$, which proves Equation 36.117.
   4. Show that (b) makes X sub-exponential in the sense of Equation 36.119 with $\nu^2 = 2\sigma^2$ and $\alpha = 2M/3$.
   5. With $n = 10{,}000$ and $\delta = 0.05$, show that Bernstein's bound is weaker than Hoeffding's at the worst-case variance $\sigma^2 = \frac{1}{4}$ of a 0/1 outcome. Below which variance does it win, and which error rates does that correspond to?

36.10 Uniform Convergence and Generalization

The previous section showed how to put an error bar on an average computed from data. A classifier evaluated on 10,000 held-out images has a true error within about 1.4 percentage points of its test error, with 95% confidence. That guarantee rests on a condition that is easy to overlook: the classifier must be fixed *before* the test images are drawn. In practice, we rarely work that way. We train many configurations, inspect their validation scores, and keep the best one. The score we report belongs to the winner of a contest, and the winner is partly the configuration that got luckiest on the sample.

Chapter 3.5 introduced this problem together with the remedy of Vapnik and Chervonenkis (Vapnik and Chervonenkis 1971): demand a guarantee that holds for every candidate function simultaneously, so that it automatically covers whichever one the learner selects. When the list of candidates is finite, the bound widens only with the logarithm of the number of candidates. We briefly recall that finite-class result here and pair it with a concrete experiment that shows just how much luck there is to pick from: flip each of 1,000 fair coins 100 times and keep the coin with the most heads. How many heads does it show?

The heart of this section, however, is the infinite function classes we actually train. For them, a later idea measures the richness of a class by how well it can fit pure coin-flip labels, its *Rademacher complexity*. This measure explains why the capacity of a linear model depends on its weight norm rather than its parameter count, and it also reveals where uniform convergence reaches its limit: at models flexible enough to fit random labels, such as the deep networks of the main text. In the final part, we revisit the double-descent exper-

iment of Section 4.5.2 in a model simple enough that we can compute every piece of the mechanism in closed form.

```
rng = np.random.default_rng()
```

36.10.1 Uniform Convergence

Evaluating a fixed model on a test set and training a learner differ only in the order of two events: choosing the function and drawing the sample. Here we measure how much reversing that order distorts the score, and how much more data we need once the choice depends on the sample.

36.10.1.1 The Best of Many Coins

Let us first state the single-function guarantee in the notation we will use throughout this section. A sample $S = \{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)\}$ consists of n independent draws from an unknown distribution, and a loss ℓ taking values in $[0, 1]$ scores each prediction, for instance the 0-1 loss, which is 1 for a mistake and 0 otherwise.

The *empirical risk* of a function f is its average loss on the sample, $\hat{R}(f) = \frac{1}{n}\sum_{i=1}^{n} \ell(f(\mathbf{x}_i), y_i)$, while its *risk* is its expected loss on a fresh draw, $R(f) = \mathbb{E}[\ell(f(\mathbf{x}), y)]$. (For the 0-1 loss, these are the empirical error $\epsilon_S(f)$ and population error $\epsilon(f)$ of the main text.) When f is fixed before S is drawn, $\hat{R}(f)$ is an average of n independent terms in $[0, 1]$ with mean $R(f)$. Hoeffding's inequality (Section 36.9) therefore bounds $P(|\hat{R}(f) - R(f)| \geq t)$ by $2e^{-2nt^2}$. Setting this failure probability to $\delta = 0.05$ and solving for t yields the half-width $\sqrt{\log(2/\delta)/(2n)}$, which the main text calls the Hoeffding radius. For $n = 10{,}000$ this gives 0.0136, the 1.4 percentage points quoted above.

A learner reverses the order of events. It searches a class $\mathcal{F}$ of functions and returns the predictor $\hat{f}$ with the smallest empirical risk $\hat{R}$ on the observed sample. Each $\hat{R}(f)$ fluctuates around its true risk $R(f)$, and the learner naturally picks a function whose random fluctuation happened to be favorable. Because $\hat{R}(\hat{f})$ is the minimum of many fluctuating quantities, an inequality that holds for each fixed f individually no longer covers it.

Our coin-flipping experiment shows just how large this selection effect can be. Suppose we flip each of $k = 1{,}000$ fair coins $m = 100$ times. For a single coin named in advance, Hoeffding's inequality at $\delta = 0.05$ allows a deviation of $\sqrt{\log 40/200} = 0.136$ in the fraction of heads, so that coin lands between 37 and 63 heads with probability at least 95% (the exact probability is 99.3%). The best of the 1,000 coins carries no such guarantee. Let us repeat the experiment 2,000 times and measure how far the winning coin strays and how often it escapes the single-coin band.

```
k, m, repeats = 1000, 100, 2000          # coins, flips per coin, experiments
best = rng.binomial(m, 0.5, size=(repeats, k)).max(axis=1) / m
one = np.sqrt(np.log(2 / 0.05) / (2 * m))       # Hoeffding, one fixed coin
many = np.sqrt(np.log(2 * k / 0.05) / (2 * m))  # union bound over k coins
print(f'best of {k} coins: median {np.median(best):.2f}, 90% of experiments '
      f'in [{np.percentile(best, 5):.2f}, {np.percentile(best, 95):.2f}]')
print(f'95% half-width: one coin {one:.3f}, all {k} coins {many:.3f}')
```

```
print(f'experiments in which the best coin leaves the one-coin band: '
      f'{np.mean(best > 0.5 + one):.0%}')
```

```
best of 1000 coins: median 0.66, 90% of experiments in [0.64, 0.69]
95% half-width: one coin 0.136, all 1000 coins 0.230
experiments in which the best coin leaves the one-coin band: 97%
```

In a typical experiment, the best coin shows 66 heads, and in nine experiments out of ten it lands between 64 and 69. It escapes the single-coin band in 96% to 97% of runs, so a guarantee that holds for every coin named in advance fails almost every time for the coin chosen afterwards. None of the coins is biased. The winner is off by 16 percentage points simply because we inspected 1,000 random fluctuations and kept the largest.

Figure 36.46 makes the reason visible. Any individual coin crosses the upper edge of the band with probability 0.0033, a rare event on its own; among 1,000 coins, however, about 3.3 are expected to do so. A tail that one coin almost never reaches becomes well populated once a thousand coins are tossed.

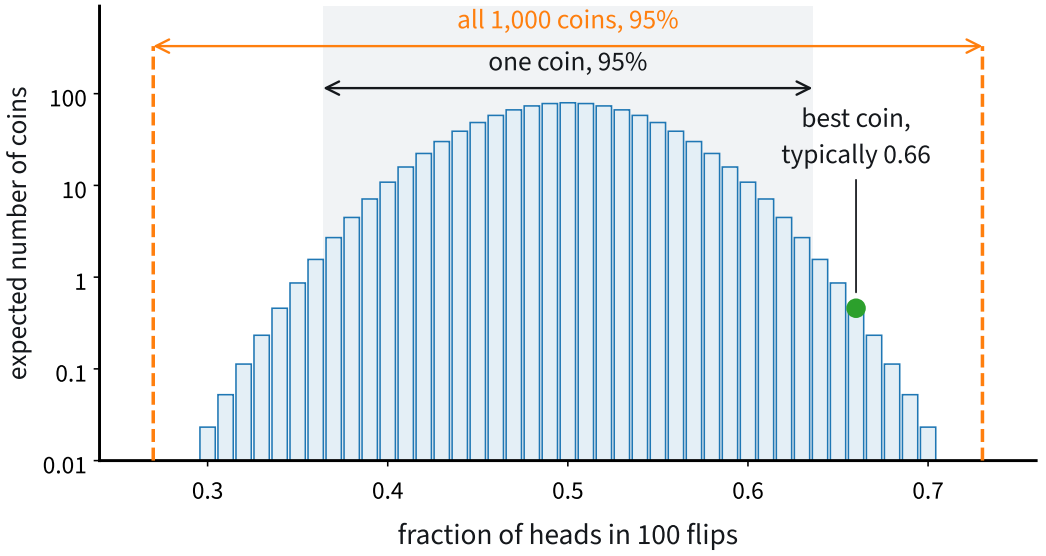


Figure 36.46: Expected histogram of the fraction of heads for 1,000 fair coins flipped 100 times each, shown on a logarithmic scale. Although the single-coin 95% Hoeffding band covers 99% of the coins, about 3.3 coins are expected above its upper edge, and the best coin typically lands near 0.66. The wider union-bound band for all 1,000 coins covers the best coin.

Model selection on a validation set is the exact same experiment with different labels: each configuration is a coin, and each validation example is a flip. Validation configurations share the same examples and have different true errors, unlike our independent fair coins, yet the bound we derive next requires neither independence across candidates nor equal means.

How, then, can we obtain a guarantee that still covers the chosen function? Suppose the largest deviation across the entire class,

$$\sup_{f \in \mathcal{F}} \left| \hat{R}(f) - R(f) \right|,$$

is at most t. Then every function's empirical risk lies within t of its true risk, and in particular so does the function the learner picked. Bounds on this supremum are called *uniform convergence* bounds (Vapnik and Chervonenkis 1971). Choosing after seeing the data becomes safe once the guarantee holds for all candidates before the data arrive, because whichever candidate we pick is already covered (Figure 36.47).

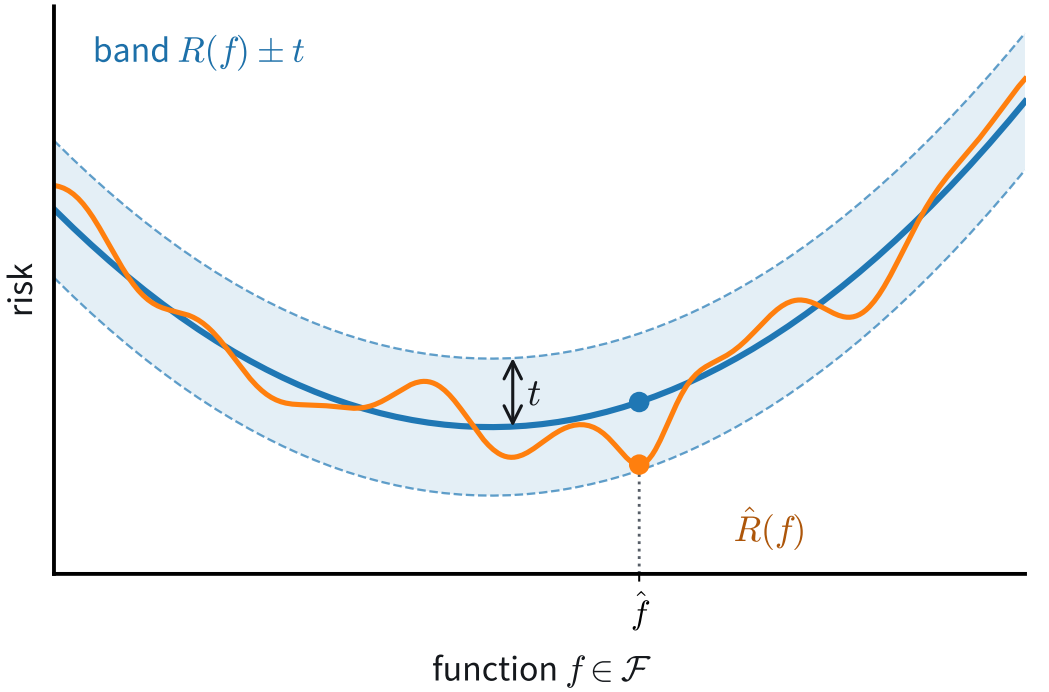


Figure 36.47: Uniform convergence. The empirical risk $\hat{R}(f)$ (orange) stays within t of the true risk $R(f)$ (blue) for every function in the class at once. The learner picks the minimizer $\hat{f}$ of the empirical risk, which sits in a downward dip below the true risk. The band covers that function as well, so $R(\hat{f}) \le \hat{R}(\hat{f}) + t$.

36.10.1.2 Finite Classes and the Union Bound

The simplest setting is a finite list of candidates fixed in advance, such as a pre-specified grid of hyperparameter configurations. For this case the main text already stated the result in Chapter 3.5, and its proof takes only two lines. Hoeffding's inequality bounds the probability that a single fixed f deviates by t or more by $2e^{-2nt^2}$. Applying the union bound (Section 36.9) across all $|\mathcal{F}|$ candidates bounds the probability that *any* of them deviates by t or more by $2|\mathcal{F}|e^{-2nt^2}$. Setting this total failure probability equal to δ and solving for t gives our first uniform convergence theorem: for a loss taking values in $[0, 1]$ and an i.i.d. sample of size n, with probability at least $1 - \delta$,

$$\left|\hat{R}(f) - R(f)\right| \le \sqrt{\frac{\log(2|\mathcal{F}|/\delta)}{2n}} \qquad \text{simultaneously for every } f \in \mathcal{F}. \tag{36.124}$$

Compared with evaluating a single fixed function, choosing among $|\mathcal{F}|$ candidates adds $\log|\mathcal{F}|$ inside the square root and nothing more.

On our coin experiment, setting $|\mathcal{F}| = 1{,}000$ and $n = 100$ produces a half-width of $\sqrt{\log(2{,}000/0.05)/200} = 0.230$, so with probability at least 95% every coin, the winning coin included, falls between 27 and 73 heads. The bound is admittedly conservative: the true probability that some coin falls outside 27 to 73 heads is about 0.17%, well below the 5% budget. Almost all of this factor-of-30 slack comes from Hoeffding's inequality on a single coin, which leaves that range with probability 1.7×10^{-6} where Hoeffding allows 5×10^{-5}; the union bound itself is nearly tight for rare independent events. Conservative or not, the band is valid, whereas the single-coin band was not.

For a validation set of 10,000 examples, the main text records the half-widths: 0.014 for a single function and 0.030 for a million. Solving Equation 36.124 for the sample size n makes the cost of searching more transparent:

$$n = \frac{\log(2/\delta) + \log|\mathcal{F}|}{2t^2}.$$

Each additional unit of $\log|\mathcal{F}|$ requires $1/(2t^2)$ more examples. At a tolerance of $t = 0.01$, doubling the number of candidate models costs $\log 2/(2 \cdot 10^{-4}) \approx 3{,}500$ extra validation examples.

This logarithmic dependence is a direct gift of Hoeffding's exponential tail. If we used Chebyshev's inequality instead (noting that a $[0,1]$-valued loss has variance at most $\tfrac{1}{4}$), the same union bound would yield $t = \sqrt{|\mathcal{F}|/(4n\delta)}$. For 2,000 configurations on 10,000 examples at $\delta = 0.05$, Chebyshev gives $t = 1$, spanning the entire range of the loss and telling us nothing, whereas Hoeffding gives $t = 0.024$.

The proof relies on one crucial hypothesis in its very first step: the candidate class $\mathcal{F}$ must be fixed before the sample is drawn. A public benchmark leaderboard violates this assumption whenever each new submission is chosen after inspecting the test scores of earlier ones (Dwork et al. 2015). Once the candidates adapt to the holdout set, the error guarantee degrades from order $\sqrt{\log k/n}$ for k fixed candidates to order $\sqrt{k/n}$ for an adaptive majority vote built from k random guessers (exercise 5).

36.10.2 Rademacher Complexity

The function classes we actually train are infinite. Every weight vector $\mathbf{w} \in \mathbb{R}^d$ defines a distinct linear classifier, and a union bound over infinitely many events is vacuous. Classical learning theory gets around this either by approximating the class with finitely many representatives (covering numbers) or by counting how many distinct labelings it can realize on n points (the VC dimension, sketched in Chapter 3.5). Here we take a more direct approach (Bartlett and Mendelson 2002) that asks an operational question any practitioner can test: how well can the class fit pure noise?

The two central results of this section live in this part: the uniform convergence theorem Equation 36.126 and the complexity bound $Br/\sqrt{n}$ for norm-bounded linear models, Equation 36.128. The one subtle step is the symmetrization argument in the proof of the first.

36.10.2.1 Correlation with Random Signs

If a function class can fit labels that carry no signal at all, it can just as easily fit the noise in real labels, and its empirical risk then tells us very little about its true risk. To test a class's capacity for memorizing noise, we feed it fictitious random labels. Let $\varepsilon_1, \dots, \varepsilon_n$ be independent fair random signs, each taking the value $+1$ or -1 with probability $\frac{1}{2}$; these are called *Rademacher variables*. For a given sign vector ε, the average $\frac{1}{n}\sum_i \varepsilon_i f(\mathbf{x}_i)$ measures how well f correlates with those signs on the sample, and the supremum over $f \in \mathcal{F}$ measures the best correlation the class can achieve.

Definition. The *empirical Rademacher complexity* of a class $\mathcal{F}$ of real-valued functions on the sample points $S = (\mathbf{x}_1, \dots, \mathbf{x}_n)$ is the expected best correlation the class achieves with random signs,

$$\widehat{\mathfrak{R}}_S(\mathcal{F}) = \mathbb{E}_\varepsilon\left[\sup_{f\in\mathcal{F}} \frac{1}{n}\sum_{i=1}^{n} \varepsilon_i\, f(\mathbf{x}_i)\right], \tag{36.125}$$

and its *Rademacher complexity* $\mathfrak{R}_n(\mathcal{F}) = \mathbb{E}_S[\widehat{\mathfrak{R}}_S(\mathcal{F})]$ is the average of this quantity over random samples of size n.

A class that can match any sign pattern scores high, whereas a class with little freedom scores near zero. The exact ceiling depends on the range of the functions. If every $f \in \mathcal{F}$ takes values in $[-1, 1]$, then every correlation is at most 1, and the class of *all* functions into $[-1, 1]$ achieves 1 on distinct points by setting $f(\mathbf{x}_i) = \varepsilon_i$ for every sign pattern. For a class taking values in $[0, 1]$, the maximum possible complexity is $\frac{1}{2}$: the best choice sets $f(\mathbf{x}_i) = 1$ wherever $\varepsilon_i = +1$ and 0 elsewhere, scoring the fraction of plus signs, which averages to $\frac{1}{2}$. An unbounded class, such as the linear functions we study shortly, has no such ceiling: if it can interpolate every sign pattern on the sample, its complexity is at least 1, and can be larger still.

To get a feel for the numbers, consider the simplest nontrivial class: the two constant functions $\{+1, -1\}$. Its best correlation with a sign vector is $\max(\bar\varepsilon, -\bar\varepsilon) = |\bar\varepsilon|$, where $\bar\varepsilon$ is the sample average of the signs. For $n = 2$, the four equally likely patterns $(+,+), (+,-), (-,+), (-,-)$ give $|\bar\varepsilon| = 1, 0, 0, 1$, so $\widehat{\mathfrak{R}}_S = \frac{1}{2}$. For $n = 4$, the sum of the signs is ± 4 with probability $\frac{2}{16}$, ± 2 with probability $\frac{8}{16}$, and 0 otherwise, giving $\widehat{\mathfrak{R}}_S = \frac{3}{8}$. For large n, $\bar\varepsilon$ is approximately normal with mean 0 and variance $1/n$, so $\widehat{\mathfrak{R}}_S \approx \sqrt{2/(\pi n)}$; at $n = 100$, the exact value is 0.0796 compared with 0.0798 from the normal approximation. Even a class with just two functions correlates with noise at rate $1/\sqrt{n}$, simply because the average of n random signs fluctuates at that scale. By contrast, a singleton class $\{f\}$ has complexity 0: with nothing to choose from, $\mathbb{E}[\frac{1}{n}\sum_i \varepsilon_i f(\mathbf{x}_i)] = 0$.

For a linear class, the best correlation has a clean geometric interpretation, previewed in Figure 36.48: it is the length of a signed sum of the input vectors, divided by n. We prove this shortly in Section 36.10.2.3.

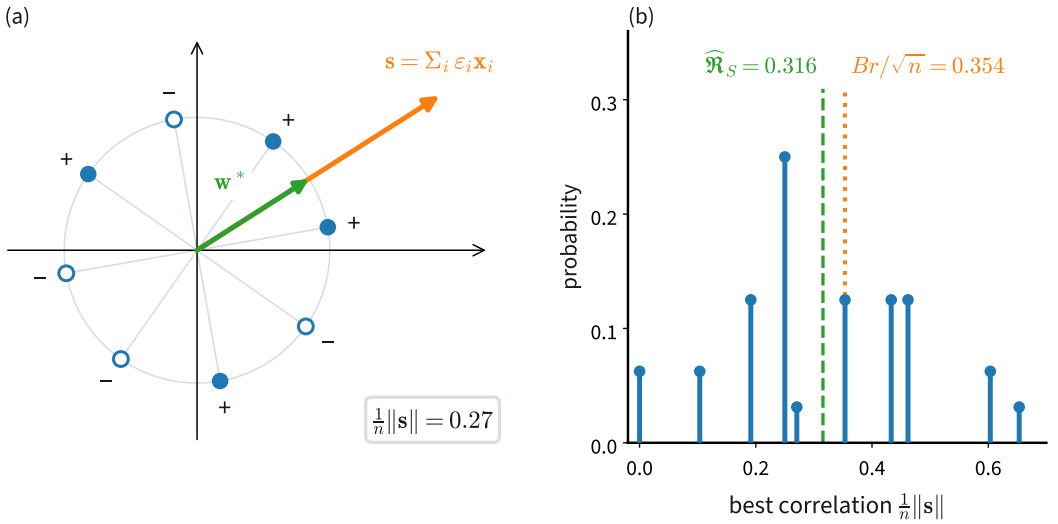


Figure 36.48: Rademacher complexity of a linear class for eight unit vectors in the plane and the class of unit-norm linear functions. (a) For a single sign pattern, filled dots denote $+1$ and hollow dots -1. The best correlation is achieved by the unit weight vector $\mathbf{w}^*$ aligned with the signed sum $\mathbf{s} = \sum_i \varepsilon_i \mathbf{x}_i$, and equals $\|\mathbf{s}\|/n = 0.27$. (b) The exact distribution of the best correlation across all 256 sign patterns. Its mean, the empirical Rademacher complexity, is 0.316, compared with the bound $Br/\sqrt{n} = 0.354$ proved in the text.

A finite class taking values in $[-1, 1]$ satisfies *Massart's lemma*, $\widehat{\mathfrak{R}}_S(\mathcal{F}) \le \sqrt{2 \log |\mathcal{F}| / n}$ (exercise 2), so its complexity grows only logarithmically with the number of functions. More generally, Rademacher complexity captures how freely a class can vary on the actual sample points, regardless of how many parameters or members it has.

36.10.2.2 The Uniform Convergence Theorem

We are now ready to show that Rademacher complexity controls the largest gap between empirical risk and true risk across the entire class. One small bookkeeping distinction comes first, because predictions and losses belong to different function classes. The *predictor class* $\mathcal{F}$ contains the models we fit, whereas the quantities whose averages we compare are the losses. We therefore define the *loss class*

$$\mathcal{G} = \{(\mathbf{x}, y) \mapsto \ell(f(\mathbf{x}), y) : f \in \mathcal{F}\},$$

whose functions take values in $[0, 1]$. Writing $z = (\mathbf{x}, y)$ for a labeled example, let $R(g) = \mathbb{E}[g(z)]$ and $\hat{R}_S(g) = \frac{1}{n}\sum_i g(z_i)$ for $g \in \mathcal{G}$. The risk of a predictor is simply the risk of its corresponding loss function.

Proposition (uniform convergence via Rademacher complexity) (Bartlett and Mendelson 2002). *Let $S = (z_1, \dots, z_n)$ be an i.i.d. sample and let $\mathcal{G}$ be a class of functions taking values in $[0, 1]$. Then with probability at least $1 - \delta$,*

$$R(g) \le \hat{R}_S(g) + 2\,\mathfrak{R}_n(\mathcal{G}) + \sqrt{\frac{\log(1/\delta)}{2n}} \qquad \text{simultaneously for every } g \in \mathcal{G}. \tag{36.126}$$

Notice that this bound has the exact same three-part anatomy as the finite-class bound: the empirical risk on the training sample, a complexity term measuring the richness of the class, and a confidence term that would appear even for a single fixed function. Here, the richness term is twice the class's ability to fit random signs.

Proof. We first bound the expected worst-case gap, $\mathbb{E}_S\big[\sup_g (R(g) - \hat{R}_S(g))\big] \le 2\mathfrak{R}_n(\mathcal{G})$, in three steps; the high-probability statement follows from concentration.

I. *Ghost sample.* Introduce an independent copy $S' = (z'_1, \dots, z'_n)$ of the sample S, which serves purely as an analytical device inside the proof. Because $R(g) = \mathbb{E}_{S'}[\hat{R}_{S'}(g)]$, and because the supremum of an expectation is always at most the expectation of the supremum, we obtain

$$\mathbb{E}_S\Big[\sup_{g\in\mathcal{G}} (R(g) - \hat{R}_S(g))\Big] \le \mathbb{E}_{S,S'}\Big[\sup_{g\in\mathcal{G}} \frac{1}{n}\sum_{i=1}^{n} (g(z'_i) - g(z_i))\Big]. \tag{36.127}$$

The unknown population risk has been replaced by the empirical risk on a fresh sample.

II. *Sign swap.* Swapping z_i with z'_i leaves the joint distribution of (S, S') completely unchanged, since both points are independent draws from the same distribution. Yet that swap negates the i-th difference $g(z'_i) - g(z_i)$ for every $g \in \mathcal{G}$ simultaneously. Consequently, for any fixed choice of signs $\varepsilon_1, \dots, \varepsilon_n \in \{-1, +1\}$, multiplying the i-th difference by ε_i does not change the right-hand side of Equation 36.127. Averaging over random signs leaves it unchanged as well:

$$\mathbb{E}_{S,S'}\Big[\sup_g \frac{1}{n}\sum_i (g(z'_i) - g(z_i))\Big] = \mathbb{E}_{S,S',\varepsilon}\Big[\sup_g \frac{1}{n}\sum_i \varepsilon_i (g(z'_i) - g(z_i))\Big].$$

III. *Split.* Since the supremum of a sum is at most the sum of the suprema, the right-hand side is bounded above by

$$\mathbb{E}_{S',\varepsilon}\Big[\sup_g \frac{1}{n}\sum_i \varepsilon_i\, g(z'_i)\Big] + \mathbb{E}_{S,\varepsilon}\Big[\sup_g \frac{1}{n}\sum_i (-\varepsilon_i)\, g(z_i)\Big] = 2\,\mathfrak{R}_n(\mathcal{G}),$$

because $-\varepsilon$ has the same distribution as ε, making both expectations equal to $\mathfrak{R}_n(\mathcal{G})$. This split is where the factor of 2 comes from.

To pass from the expectation to a high-probability bound, define $\Phi(S) = \sup_g (R(g) - \hat{R}_S(g))$. Replacing a single sample point z_i changes each empirical risk $\hat{R}_S(g)$ by at most $1/n$ (since g takes values in $[0,1]$), and therefore changes the supremum $\Phi(S)$ by at most $1/n$. *McDiarmid's inequality* (McDiarmid 1989), which generalizes Hoeffding's inequality from sums to any function that no single input can shift by much, then gives $P(\Phi - \mathbb{E}\Phi \ge t) \le e^{-2nt^2}$ (exercise 1). Setting the right-hand side to δ and combining with $\mathbb{E}\Phi \le 2\mathfrak{R}_n(\mathcal{G})$ yields Equation 36.126. ■

The intuition behind this *symmetrization* argument is remarkably clean. The ghost sample stands in for the population. Because real and ghost points are exchangeable, swapping any pair changes nothing in distribution, and once those swaps are randomized, only coin flips remain: the worst-case gap between training and population risk is governed by how well the class can chase random signs.

One practical detail remains. The theorem states the bound in terms of the expected complexity $\mathfrak{R}_n(\mathcal{G})$, whereas on a real dataset we can only evaluate the empirical complexity $\widehat{\mathfrak{R}}_S(\mathcal{G})$ on the sample at hand. Because changing one sample point also moves $\widehat{\mathfrak{R}}_S(\mathcal{G})$ by at most $1/n$, a second application of McDiarmid's inequality lets us substitute $\widehat{\mathfrak{R}}_S(\mathcal{G})$ for $\mathfrak{R}_n(\mathcal{G})$ in Equation 36.126 at the cost of widening the confidence term to $3\sqrt{\log(2/\delta)/(2n)}$. Our numerical checks below use this empirical form.

It is always worth checking a theorem at its extremes. For a singleton class, $\mathfrak{R}_n = 0$ and Equation 36.126 reduces directly to one-sided Hoeffding. For a finite class, Massart's lemma recovers the shape of Equation 36.124, though with looser constants: for 2,000 configurations on 10,000 examples, the complexity term alone is $2\sqrt{2\log 2000/n} = 0.078$, compared with the union bound's entire half-width of 0.024. Rademacher complexity pays a constant-factor penalty on finite classes, in exchange for handling infinite classes effortlessly. For binary classifiers, counting the labelings realizable on n points bounds $\mathfrak{R}_n$ by a constant times $\sqrt{d_{\mathrm{VC}} \log n / n}$ (Boucheron et al. 2005), where d_{VC} is the VC dimension from the main text, the largest number of points on which the class can realize every possible labeling.

At the opposite extreme, any loss class taking values in $[0,1]$ satisfies $\mathfrak{R}_n(\mathcal{G}) \le \tfrac{1}{2}$. When equality holds, the complexity term $2\mathfrak{R}_n(\mathcal{G})$ equals 1, and the bound becomes vacuous, allowing any risk up to 100%. For binary classifiers taking values in $\{-1,+1\}$ under the 0-1 loss $g(\mathbf{x},y) = (1 - yf(\mathbf{x}))/2$, the loss complexity and predictor complexity stand in an exact factor-of-two ratio. Expanding the correlation gives $\sum_i \varepsilon_i g(z_i) = \tfrac{1}{2}\sum_i \varepsilon_i + \tfrac{1}{2}\sum_i (-\varepsilon_i y_i) f(\mathbf{x}_i)$. The first sum has mean zero and does not depend on f, while for fixed labels $y_i \in \{-1,+1\}$, the products $-\varepsilon_i y_i$ are again independent fair random signs. Taking the supremum and expectation yields

$$\widehat{\mathfrak{R}}_S(\mathcal{G}) = \tfrac{1}{2}\,\widehat{\mathfrak{R}}_S(\mathcal{F}).$$

Therefore, if a binary predictor class can fit every sign pattern on the sample, we have $\widehat{\mathfrak{R}}_S(\mathcal{F}) = 1$, twice the loss complexity is 1, and the uniform convergence theorem promises nothing better than an error rate below 100%.

36.10.2.3 The Linear Class

A generalization theorem is only as useful as our ability to evaluate its complexity term on classes we actually care about. For linear predictors with a bounded weight norm, the calculation takes just four lines, and its answer is surprisingly independent of the input dimension.

Proposition (Rademacher complexity of a norm-bounded linear class). *Let* $\mathcal{F} = \{\mathbf{x} \mapsto \langle \mathbf{w}, \mathbf{x}\rangle : \|\mathbf{w}\|_2 \le B\}$ *and suppose every sample input satisfies* $\|\mathbf{x}_i\|_2 \le r$. *Then*

$$\widehat{\mathfrak{R}}_S(\mathcal{F}) = \frac{B}{n}\,\mathbb{E}_\varepsilon\Big\|\sum_{i=1}^{n}\varepsilon_i\mathbf{x}_i\Big\| \le \frac{B\,r}{\sqrt{n}}. \tag{36.128}$$

Proof. The inner supremum has a closed form. By linearity of the inner product and the Cauchy–Schwarz inequality, the maximum is attained by the weight vector $\mathbf{w}$ of length B pointing in the direction of the signed sum:

$$\sup_{\|\mathbf{w}\|\le B}\frac{1}{n}\sum_{i=1}^{n}\varepsilon_i\langle\mathbf{w},\mathbf{x}_i\rangle = \sup_{\|\mathbf{w}\|\le B}\frac{1}{n}\Big\langle\mathbf{w},\,\sum_i\varepsilon_i\mathbf{x}_i\Big\rangle = \frac{B}{n}\Big\|\sum_{i=1}^{n}\varepsilon_i\mathbf{x}_i\Big\|.$$

Taking the expectation over ε yields the equality in the proposition. To obtain the upper bound, invoke the concavity of the square root ($\mathbb{E}\|\cdot\| \le \sqrt{\mathbb{E}\|\cdot\|^2}$) and expand the squared norm:

$$\mathbb{E}_\varepsilon\Big\|\sum_i\varepsilon_i\mathbf{x}_i\Big\| \le \sqrt{\sum_{i,j}\mathbb{E}[\varepsilon_i\varepsilon_j]\,\langle\mathbf{x}_i,\mathbf{x}_j\rangle} = \sqrt{\sum_i\|\mathbf{x}_i\|^2} \le r\sqrt{n}.$$

All cross terms vanish because independent fair signs satisfy $\mathbb{E}[\varepsilon_i\varepsilon_j] = 0$ for $i \ne j$. Multiplying by B/n gives $Br/\sqrt{n}$. ■

Panel (a) of Figure 36.48 illustrates the first step of this proof: for any sign pattern, the optimal weight vector simply aligns with the signed sum of the inputs.

Remarkably, only one step in this proof introduces any slack. Because the identity $\mathbb{E}\|\sum_i\varepsilon_i\mathbf{x}_i\|^2 = \sum_i\|\mathbf{x}_i\|^2$ is exact, the entire gap comes from replacing the mean length of the signed sum by its root-mean-square length. That inequality is tight whenever the length varies little across sign patterns. For the eight points in the figure, the slack is 11% (0.316 versus 0.354). In two dimensions, a signed sum of many well-spread unit vectors behaves like a two-dimensional Gaussian vector, whose mean length is $\sqrt{\pi}/2 \approx 0.89$ times its root-mean-square length, so the slack stays near 11% regardless of n. In twenty dimensions, however, the length of the signed sum concentrates tightly around its root-mean-square by the thin-shell phenomenon of Section 36.9.2.1, and the slack drops to about 1%.

Notice what is missing from Equation 36.128: the input dimension d does not appear at all. It enters only indirectly through B and r. In this bound, the capacity of a linear model is governed by the product of its weight norm and its input radius, whether it operates on ten features or on a million. This gives a clean theoretical motivation for weight decay (Section 2.7): keeping $\|\mathbf{w}\|$ small shrinks the function class that contains the trained model. Nor is this dimension-free behavior restricted to linear models: a two-layer ReLU network whose output weights have ℓ_1 norm at most B_1 and whose hidden weight vectors have ℓ_2 norm at most B_2 has Rademacher complexity at most $2B_1B_2r/\sqrt{n}$, regardless of its width (exercise 8).

A careful reader may raise a natural objection: the proposition requires fixing the radius B *before* seeing the data, whereas the norm of a trained weight vector is known only *after* training. That objection is valid, and a union bound over a ladder of radii resolves it. Apply the theorem to each radius $B_j = 2^j$ with failure probability $\delta_j = \delta/(j(j+1))$ (which sum to δ), and evaluate the bound at the smallest B_j containing the fitted weight vector. For $\|\mathbf{w}\| >$

1, this costs at most a factor of 2 in the complexity term and adds a mild $\sqrt{\log\log\|\mathbf{w}\|/n}$ penalty (exercise 3).

One link remains before we can turn Equation 36.128 into a generalization bound for linear classifiers. The uniform convergence theorem requires the complexity of the *loss* class $\mathcal{G}$, whereas our proposition bounded the *predictor* class $\mathcal{F}$. A Lipschitz loss bridges the two.

Lemma (contraction) (Boucheron et al. 2013). *Let $\phi_1, \dots, \phi_n : \mathbb{R} \to \mathbb{R}$ each be L-Lipschitz. Then for any function class $\mathcal{F}$ and points $\mathbf{x}_1, \dots, \mathbf{x}_n$,*

$$\mathbb{E}_\varepsilon\Big[\sup_{f\in\mathcal{F}} \sum_{i=1}^{n} \varepsilon_i\, \phi_i(f(\mathbf{x}_i))\Big] \le L\, \mathbb{E}_\varepsilon\Big[\sup_{f\in\mathcal{F}} \sum_{i=1}^{n} \varepsilon_i\, f(\mathbf{x}_i)\Big]. \tag{36.129}$$

In plain terms, composing a function class with transformations that stretch distances by at most L can multiply its Rademacher complexity by at most L. Allowing each ϕ_i to be different is precisely what a supervised loss requires: at example i, ϕ_i is the map $u \mapsto \ell(u, y_i)$ with the label y_i held fixed. No centering condition is needed, because adding a constant c to ϕ_i shifts the sum by $c\,\varepsilon_i$, which has expectation zero. We state the contraction lemma without proof.

The uniform convergence theorem also requires losses bounded in $[0, 1]$. Common training losses such as hinge and logistic loss are unbounded: on our norm-bounded linear class they can reach roughly $1 + Br$, so applying the theorem directly would require dividing by that maximum and bounding a rescaled loss. A cleaner choice for analysis is a loss that is bounded from the outset: the *ramp loss* applied to the classification margin $u = yf(\mathbf{x})$,

$$\phi_\gamma(u) = \min\{1, \max\{0,\ 1 - u/\gamma\}\}.$$

The ramp loss equals 1 on a mistake ($u \le 0$), drops to 0 once a prediction is correct by at least the margin γ ($u \ge \gamma$), and interpolates linearly in between. It upper-bounds the 0-1 loss everywhere and is $(1/\gamma)$-Lipschitz, as Figure 36.49 shows.

Write $\hat{R}_\gamma(\mathbf{w}) = \frac{1}{n}\sum_i \phi_\gamma(y_i\langle\mathbf{w}, \mathbf{x}_i\rangle)$ for the empirical ramp loss on the sample. When every input satisfies $\|\mathbf{x}\| \le r$, combining the contraction lemma with our linear-class proposition bounds the loss-class complexity by $Br/(\gamma\sqrt{n})$. Plugging this into Equation 36.126 yields a *margin bound*: with probability at least $1 - \delta$, simultaneously for every $\mathbf{w}$ with $\|\mathbf{w}\| \le B$,

$$P(y\langle\mathbf{w}, \mathbf{x}\rangle \le 0) \le \hat{R}_\gamma(\mathbf{w}) + \frac{2Br}{\gamma\sqrt{n}} + \sqrt{\frac{\log(1/\delta)}{2n}}. \tag{36.130}$$

The left-hand side is the true classification error rate, while the right-hand side is the training ramp loss plus two computable terms. To see the scale of these numbers, take an illustrative setting with $Br/\gamma = 10$, $n = 50{,}000$, and $\delta = 0.05$ (choosing the ratio rather than measuring it on a specific dataset). The complexity term is 0.089 and the confidence term is 0.005, totaling 0.095. Thus, with $B = 10\gamma/r$ fixed in advance, any linear classifier in the class that separates its training set with margin γ is guaranteed, with 95% confidence, to have a true error rate below 9.5%. By contrast, forcing a linear model to fit random labels with margin γ drives Br/γ upward: past $Br/\gamma \approx 10.6$ the two penalty terms exceed 0.1, and near $Br/\gamma \approx 110$ the complexity term alone reaches 1, rendering the bound vacuous.

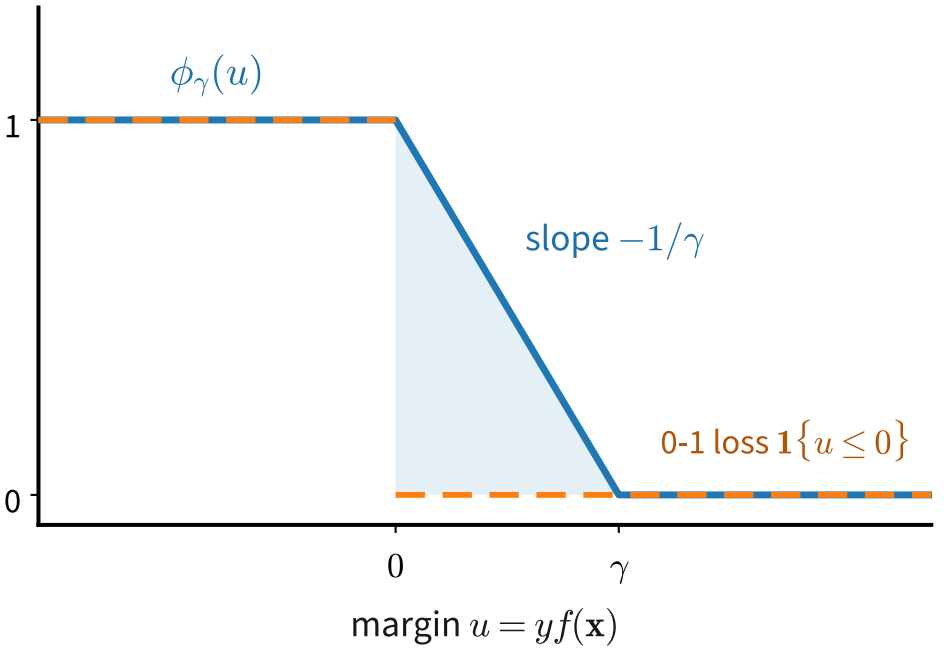


Figure 36.49: The ramp loss ϕ_γ (blue) as a function of the margin $u = yf(\mathbf{x})$, together with the 0-1 loss $\mathbf{1}\{u \le 0\}$ (dashed). The two agree except for $0 < u < \gamma$ (shaded), where a prediction is correct by less than the margin and the ramp descends with slope $-1/\gamma$.

36.10.2.4 Rademacher Complexity in Numbers

Because Equation 36.128 expresses the empirical Rademacher complexity of a linear class as an expected Euclidean norm, we can estimate it directly by averaging over random sign vectors. Below we do this for 50 unit-norm points in $d = 2$, 20, and 1,000 dimensions. Our slack analysis predicts an estimate roughly 11% below the bound $Br/\sqrt{n} = 1/\sqrt{50}$ in two dimensions, and within about 1% of the bound once we reach twenty dimensions.

```
n, B, num_signs = 50, 1.0, 2000
signs = rng.choice([-1.0, 1.0], size=(num_signs, n))  # rows: sign vectors
for d in (2, 20, 1000):
    X = rng.standard_normal((n, d))
    X /= np.linalg.norm(X, axis=1, keepdims=True)       # ||x_i|| = r = 1
    sup_corr = B * np.linalg.norm(signs @ X, axis=1) / n  # closed-form sup
    print(f'd = {d:4d}: Monte-Carlo R_S = {sup_corr.mean():.3f}, '
          f'bound B r / sqrt(n) = {B / np.sqrt(n):.3f}')
```

```
d =    2: Monte-Carlo R_S = 0.123, bound B r / sqrt(n) = 0.141
d =   20: Monte-Carlo R_S = 0.141, bound B r / sqrt(n) = 0.141
d = 1000: Monte-Carlo R_S = 0.141, bound B r / sqrt(n) = 0.141
```

The Monte Carlo estimate lands around 0.125 in the plane and between 0.139 and 0.141 in 20 and 1,000 dimensions, right against the bound of 0.141, just as predicted. Increasing the dimension fifty-fold from 20 to 1,000 leaves the complexity unchanged.

At the opposite end of the spectrum lies a class rich enough to *interpolate*, meaning it can reproduce any assignment of labels on the training sample exactly. Consider least squares on $p = 2n$ standard Gaussian features, where the feature matrix $\Phi \in \mathbb{R}^{n\times p}$ holds the feature vectors of the n points as rows. Using the pseudoinverse from Section 33.5.3.1, the solution $\mathbf{w} = \Phi^{+}\varepsilon$ returns the minimum-norm interpolant for every sign vector ε, achieving a correlation of 1 on every pattern by construction. Our proposition tells us how large a weight ball is needed to hold all of these fits: any class that interpolates every sign pattern must have $\widehat{\mathfrak{R}}_S \geq 1$, while Equation 36.128 bounds it by $Br/\sqrt{n}$, so any ball containing all the interpolants must satisfy $Br/\sqrt{n} \geq 1$. How close to this lower limit do actual fits come? Next, we fit 2,000 random sign vectors, compare their weight norms with the theoretical prediction $\sqrt{n/(p-n-1)}$ (derived in exercise 4), and evaluate $Br/\sqrt{n}$ for the smallest ball enclosing them.

```
p = 2 * n                                   # more features than points
Phi = rng.standard_normal((n, p))
W = np.linalg.pinv(Phi) @ signs.T           # min-norm fit of each sign vector
w_norm = np.linalg.norm(W, axis=0)
r = np.linalg.norm(Phi, axis=1).max()       # input radius: largest row norm
row_norm = np.linalg.norm(Phi, axis=1).mean()
print(f'mean ||w|| = {w_norm.mean():.2f} (predicted '
      f'{np.sqrt(n / (p - n - 1)):.2f}), largest {w_norm.max():.2f}')
print(f'typical fit: mean ||w|| * mean row norm / sqrt(n) = '
      f'{w_norm.mean() * row_norm / np.sqrt(n):.2f}')
print(f'r = {r:.1f}, B r / sqrt(n) with B = largest ||w||: '
      f'{w_norm.max() * r / np.sqrt(n):.1f}')
```

```
mean ||w|| = 0.99 (predicted 1.01), largest 1.31
```

```
typical fit: mean ||w|| * mean row norm / sqrt(n) = 1.40
r = 12.4, B r / sqrt(n) with B = largest ||w||: 2.3
```

Each fit has a weight norm near 1, and the rows of Φ have average length near $\sqrt{p} = 10$, placing a typical fit at $\|\mathbf{w}\| r/\sqrt{n} \approx 1.4$. Across 2,000 sign patterns, the largest weight norm falls between 1.3 and 1.8 and the longest row between 11 and 12.5, so the ball enclosing all 2,000 fits has $Br/\sqrt{n}$ between 2 and 3. Interpolating pure noise therefore requires a norm only 1.5 to 3 times the theoretical floor: the linear-class bound $Br/\sqrt{n}$ remains nearly tight even in the regime where it ceases to be informative.

The same arithmetic applies directly to neural networks. Consider the class of all classifiers a given architecture can represent. Whenever training on random signs ε finds a function matching them at nearly every point, the supremum in Equation 36.125 is close to 1 for that ε; if this happens for typical sign vectors, the Rademacher complexity itself is close to 1. In Section 4.5.1.1, an MLP fit random labels on its training images with zero error, mirroring the image-classification experiments of Zhang et al. (2021). (Those experiments tested a few labelings with ten classes, so they are empirical evidence rather than a formal proof.) Twice the loss-class complexity is then close to 1, and Equation 36.126 allows any test error whatsoever. Yet the very same network generalizes well when trained on real labels. Whatever governs its test error is therefore invisible to any worst-case bound over the entire function class: it depends on the data, the optimizer, and the specific solution training reaches. To see how that works, we turn to a model where the learned solution can be analyzed in closed form.

36.10.3 Interpolation and Double Descent

In Section 4.5.2, we swept the width h of a network whose hidden layer remained frozen at its random initialization. The test error spiked at $h = n$, the exact threshold where interpolation first became possible and the smallest singular value of the hidden representation collapsed, and then descended again as h grew further. Here we recreate that construction on synthetic data where every piece of the mechanism can be computed explicitly, adding two ingredients to the main text's account: a closed-form formula for the fitted weights in terms of singular values, and a decomposition of the test error into the part caused by the true target and the part caused by label noise. The features are $\phi(\mathbf{x}) = \mathrm{ReLU}(\mathbf{V}\mathbf{x})$ with a frozen random matrix $\mathbf{V}$, known as a *random-features* model (Rahimi and Recht 2007), and we fit only the linear output head $\mathbf{w}$. Readers on a first pass may skip this part without losing the thread of the chapter.

36.10.3.1 The Minimum-Norm Interpolant

To understand why the interpolation threshold $p = n$ behaves so dramatically, let us write the solution in closed form. With p features and n training points, stack the training feature vectors as the rows of $\Phi \in \mathbb{R}^{n \times p}$, and fit $\mathbf{w} = \Phi^{+}\mathbf{y}$. Below $p = n$, this is ordinary least squares; above $p = n$, it is the interpolant of smallest Euclidean norm.

The minimum-norm interpolant is not an arbitrary choice among the many interpolants: it is the one that training finds. The gradient of the squared loss, $\Phi^{\top}(\Phi\mathbf{w} - \mathbf{y})$, is a combination of the training feature vectors, so gradient descent started at $\mathbf{w} = \mathbf{0}$ never leaves the row space of Φ. The only interpolant in that row space is $\Phi^{+}\mathbf{y}$, and gradient descent converges to it without any penalty on the norm. This preference of an optimizer for one solution among many that fit the data equally well is called its *implicit bias*.

The singular value decomposition illuminates all three regimes: below, at, and above the threshold. Suppose Φ has full rank $q = \min(n, p)$ and singular value decomposition $\Phi = \sum_{i=1}^{q} \sigma_i \mathbf{u}_i \mathbf{v}_i^\top$, with $\sigma_1 \geq \cdots \geq \sigma_q > 0$. Then the pseudoinverse solution is

$$\mathbf{w} = \Phi^+ \mathbf{y} = \sum_{i=1}^{q} \frac{\langle \mathbf{u}_i, \mathbf{y} \rangle}{\sigma_i} \mathbf{v}_i. \tag{36.131}$$

Each component of $\mathbf{y}$ along a left singular vector $\mathbf{u}_i$ is divided by its singular value σ_i, so the smallest singular value $\sigma_{\min} = \sigma_q$ acts as the strongest amplifier. Whatever component of $\mathbf{y}$ falls along $\mathbf{u}_q$ enters $\mathbf{w}$ magnified by $1/\sigma_{\min}$, and this amplification hits both label noise and the part of the true target that the features approximate poorly.

When $p < n$, the fitted predictions $\Phi\mathbf{w}$ simply project $\mathbf{y}$ onto the p-dimensional span of the $\mathbf{u}_i$, leaving the remainder as training residual. At $p = n$, however, those n directions span all of $\mathbb{R}^n$: the fit becomes exact, and no residual is left to absorb the poorly represented component. A square random ReLU-feature matrix is almost surely invertible, yet rarely *well* conditioned: its smallest singular value is typically tiny, and by the condition-number analysis of Section 33.5.3.2, the solution shoots far out along $\mathbf{v}_q$. Adding features beyond $p = n$ gives the matrix extra room to span $\mathbb{R}^n$, and $\sigma_{\min}$ climbs back up.

Past the threshold, the solution norm can only decrease as we add features, provided the feature sets are nested (the $(p+1)$-feature model keeps all p previous features and appends one more). This follows from the padding argument of Section 4.5.2: assigning weight zero to the new feature turns the minimum-norm p-feature interpolant into a valid $(p+1)$-feature interpolant of identical norm, so the new minimum norm can be no larger. Notice that this monotonicity relies on exact interpolation. Below $p = n$, the least-squares fit does not interpolate, and its norm actually *grows* with p: in the sweep below, it climbs from roughly 0.5 at $p = 2$ to nearly 2 at $p = 15$.

Our linear-class bound suggests a natural quantity to monitor along the sweep: the *capacity product* $\|\mathbf{w}\| \cdot \max_i \|\phi(\mathbf{x}_i)\|/\sqrt{n}$, which corresponds to $Br/\sqrt{n}$ for the smallest ball containing the fitted weights. We borrow it purely as a diagnostic, not as a formal bound, because this sweep uses squared loss on unbounded regression targets rather than a bounded classification loss. Past the threshold, the feature radius grows like $\sqrt{p}$ while $\|\mathbf{w}\|$ shrinks, so the product's net behavior is not obvious in advance.

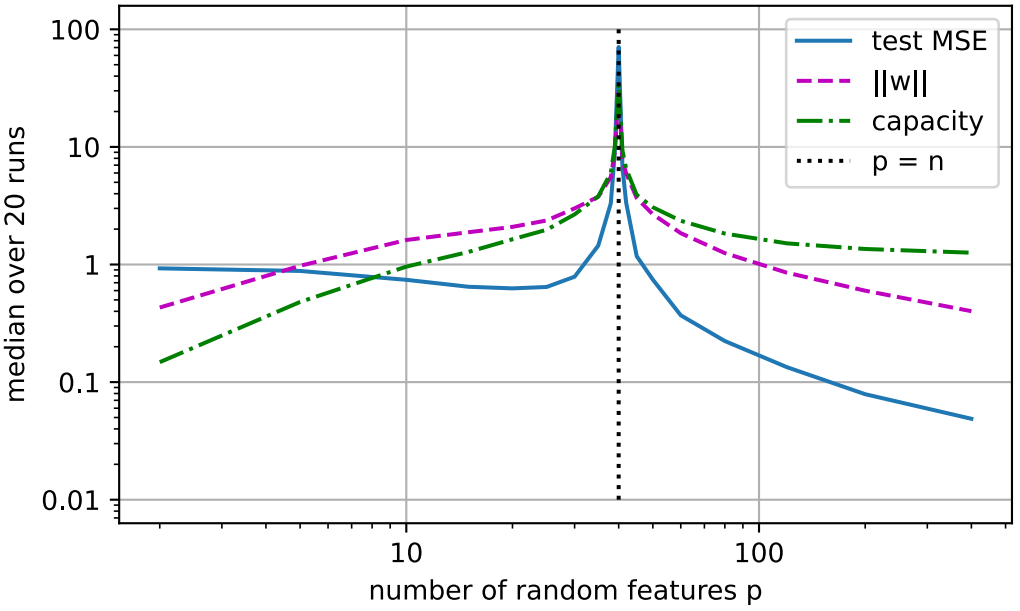
36.10.3.2 A Double-Descent Sweep

Let us watch these quantities move together on a concrete problem. We draw $n = 40$ training points from a linear teacher $y = \langle \beta, \mathbf{x} \rangle$ in $d = 15$ dimensions, with $\|\beta\| = 1$ and Gaussian label noise of standard deviation 0.1. We construct nested ReLU random features and fit the linear head via `pinv`, giving least squares below $p = 40$ and the minimum-norm interpolant above it. We sweep p from 2 to 400, including $p = 39$, 40, and 41 so the threshold peak is sharply resolved. At each p, we log the test MSE, training MSE, $\|\mathbf{w}\|$, $\sigma_{\min}$, and the capacity product, along with the two components of test error analyzed in Section 36.10.3.3: the *clean part* (the test error of fitting noise-free labels) and the *noise part* (the additional test error injected by label noise). Taking medians across 20 independent trials keeps the curves readable.

```
n, n_test, d_in, p_max, trials, noise = 40, 500, 15, 400, 20, 0.1
p_grid = np.array([2, 5, 10, 15, 20, 25, 30, 35, 38, 39, 40, 41, 42, 45, 50,
                   60, 80, 120, 200, 400])
keys = ('test', 'train', 'norm', 's_min', 'capacity', 'clean', 'noise',
        'ratio_q', 'ratio')
stats = np.zeros((len(keys), trials, len(p_grid)))
ols = np.zeros(trials)                              # least squares on x itself
for tr in range(trials):
    beta = rng.standard_normal(d_in)
    beta /= np.linalg.norm(beta)                    # teacher: y = <beta, x>
    X, X_te = (rng.standard_normal((size, d_in)) for size in (n, n_test))
    y_clean, y_te = X @ beta, X_te @ beta
    y = y_clean + noise * rng.standard_normal(n)    # noisy training labels
    ols[tr] = np.mean((X_te @ np.linalg.lstsq(X, y, rcond=None)[0] - y_te) ** 2)
    V = rng.standard_normal((p_max, d_in)) / np.sqrt(d_in)
    Phi, Phi_te = np.maximum(X @ V.T, 0), np.maximum(X_te @ V.T, 0)
    for j, p in enumerate(p_grid):                  # nested feature prefixes
        F, F_te = Phi[:, :p], Phi_te[:, :p]
        P = np.linalg.pinv(F)                       # least squares / min norm
        w = P @ y
        r = np.linalg.norm(F, axis=1).max()         # feature radius
        _, S, Vt = np.linalg.svd(F, full_matrices=False)
        ratio = ((np.linalg.norm(F_te @ Vt.T, axis=0) ** 2 / n_test)
                 / (S ** 2 / n))                    # test/train feature power
        stats[:, tr, j] = (
            np.mean((F_te @ w - y_te) ** 2),                # test MSE
            np.mean((F @ w - y) ** 2),                      # training MSE
            np.linalg.norm(w),                              # ||w||
            S.min(),                                        # sigma_min
            np.linalg.norm(w) * r / np.sqrt(n),             # capacity
            np.mean((F_te @ (P @ y_clean) - y_te) ** 2),    # clean part
            noise ** 2 * np.linalg.norm(F_te @ P) ** 2 / n_test,  # noise part
            ratio[-1], ratio.mean())                # along v_q, average
med = dict(zip(keys, np.median(stats, axis=1)))
```

The figure below plots the medians of test MSE, solution norm, and the capacity product on logarithmic axes.

```
d2l.plot([p_grid] * 3 + [[n, n]],
         [med['test'], med['norm'], med['capacity'], [1e-2, 1e2]],
         'number of random features p', 'median over 20 runs',
         legend=['test MSE', '||w||', 'capacity', 'p = n'], xscale='log',
         yscale='log', fmts=('-', 'm--', 'g-.', 'k:'), figsize=(6, 3.5))
```



The table below prints the full breakdown at seven representative feature counts. Its final two columns belong to Section 36.10.3.3, and the last line reports the classical baseline that fits least squares directly on the raw inputs.

```
print('   p ' + ' '.join(f'{key:>8}' for key in keys))
for p in (15, 38, 40, 42, 50, 80, 400):
    row = [med[key][p_grid == p][0] for key in keys]
    print(f'{p:4d} ' + ' '.join(f'{v:8.2g}' for v in row))
print(f'least squares on the {d_in} raw inputs: test MSE {np.median(ols):.4f}')
```

```
  p     test    train     norm    s_min capacity    clean    noise  ratio_q    ratio
 15     0.65     0.28      1.9        1      1.3     0.67   0.0066      3.7      1.8
 38      3.3   0.0081      5.5     0.08        6      2.9     0.31  6.8e+02       32
 40       71  2.3e-28       29    0.016       34       58      4.2  1.5e+04  4.2e+02
 42      3.4  1.3e-29        6    0.076      6.5      2.9     0.26  5.8e+02       26
 50     0.74  6.6e-30      2.7     0.25      3.1     0.64    0.063       62      6.3
 80     0.22    7e-30      1.3     0.71      1.8     0.19    0.021       10      2.1
400    0.049  5.3e-30      0.4      2.6      1.3     0.04   0.0095      2.3     0.95
least squares on the 15 raw inputs: test MSE 0.0066
```

The table reads best in three regimes: below the threshold, at the threshold, and beyond it.

Below the threshold, the model behaves classically, with fewer parameters than training points. Its test error first drops from roughly 1 at $p = 2$ to about 0.65–0.85 near $p = 15$, and then climbs again as p approaches 40, tracing the familiar classical U-curve. Two caveats are essential before we compare anything against this regime. First, even at its best, this classical fit is weak: the targets have variance 1 (the error of always predicting zero), so the minimum of the U-curve removes no more than about a third of the variance. Second, a far better classical model exists. Because the true teacher is linear in the 15 raw inputs, ordinary least squares on those raw inputs has expected test error $0.01 \cdot 15/(40-15-1) \approx 0.006$ (the noise variance times $d/(n-d-1)$ for $d = 15$), confirmed by the last line of the output.

The random-features model is handicapped by design: it must reconstruct a linear function from ReLU ridges, and forty points are far too few for that.

At the threshold, three columns move in lockstep. The training error plummets from a few times 10^{-3} at $p = 38$ to roughly 10^{-28}, exact interpolation up to floating-point precision. Simultaneously, the smallest singular value drops to around 0.01–0.04, several times smaller than at $p = 38$ or $p = 42$. Because Equation 36.131 divides by $\sigma_{\min}$, the solution explodes: $\|\mathbf{w}\|$ jumps to between 10 and 30, and test MSE spikes to between 10 and 70, one to two orders of magnitude above the U-curve minimum. Those ranges are wide because the peak is heavy-tailed, just as for a square Gaussian matrix where $\mathbb{E}[1/\sigma_{\min}^2] = \infty$; even a median over 20 trials fluctuates noticeably from run to run.

Beyond the threshold, everything that spiked comes back down. The smallest singular value recovers to roughly 2.8 at $p = 400$. As the padding argument guarantees, $\|\mathbf{w}\|$ declines monotonically in every trial (not just in the medians), reaching roughly 0.4. Test MSE follows it downward, to about 0.2 at $p = 80$ and 0.04–0.07 at $p = 400$. Meanwhile, one column goes completely silent: training error sits at zero (up to floating point) everywhere past $p = 40$, so no criterion based on training loss can tell $p = 42$ apart from $p = 400$, even though their test errors differ by more than an order of magnitude. The solution norm, $\sigma_{\min}$, and the capacity product all distinguish them clearly.

What, then, does this sweep actually prove? Less than the dramatic curve might suggest at first glance. Within the nested family of random features, the widest model beats every narrower one, including the best underparameterized model, by roughly a factor of ten, even far past the point where it memorizes all label noise. That is a genuine and surprising phenomenon. Yet it still loses to ordinary least squares on the raw inputs by another factor of ten. Whether an interpolating model can outperform a well-specified small model is a separate question, and here the answer is no.

The capacity product peaks near 10 to 35 at $p = 40$ and drops back to roughly 1.2–1.3 at $p = 400$. It mirrors the shape of the test curve, but we should not mistake it for a generalization certificate: no theorem in this section covers unbounded squared loss, the product is not invariant to rescaling the targets, and B was read off the fitted solution rather than fixed before seeing the data.

Finally, notice that label noise is *not* what causes the peak. Rerunning the sweep with `noise = 0` produces a peak of almost the same height, and the table shows why: at $p = 40$, the clean part of the test error is already larger than the noise part, typically several times larger. Because a linear teacher cannot be written as a finite combination of ReLU features, the approximation error along $\mathbf{u}_q$ is what $\sigma_{\min}$ divides. If we replace the teacher with an exact linear combination of the first ten features and remove the noise, the peak disappears altogether (exercise 6).

36.10.3.3 Benign Overfitting

One puzzle remains. At $p = 400$, the model interpolates every noisy training label with zero residual, yet it achieves the lowest test error in the sweep. Why does fitting the noise do so little damage there?

We can answer this by splitting the test error into two computable parts. Write the noisy training labels as $\mathbf{y} = \mathbf{y}_0 + \xi$, where $\mathbf{y}_0$ are the noise-free labels and ξ is independent noise with variance $\sigma_\xi^2 = 0.01$ per example. By linearity of the pseudoinverse, $\mathbf{w} = \Phi^+\mathbf{y}_0 + \Phi^+\xi$, so the test predictions split into $\Phi_{\text{te}}\Phi^+\mathbf{y}_0 + \Phi_{\text{te}}\Phi^+\xi$. Because ξ has mean zero, the cross term

vanishes in expectation, leaving an exact decomposition of the expected test MSE over n_{te} test points:

$$\mathbb{E}_\xi[\text{test error}] = \underbrace{\frac{1}{n_{\text{te}}}\left\|\Phi_{\text{te}}\Phi^+\mathbf{y}_0 - \mathbf{y}_{\text{te}}\right\|^2}_{\text{clean part}} + \underbrace{\frac{\sigma_\xi^2}{n_{\text{te}}}\left\|\Phi_{\text{te}}\Phi^+\right\|_F^2}_{\text{noise part}}.$$

The `clean` and `noise` columns of our table print these two terms. At $p = 40$, the noise part is roughly 1 to 4 (a hundred times or more the label-noise variance of 0.01), and the clean part is larger still. By $p = 42$ the noise part drops to a few tenths, at $p = 80$ to about 0.02, and at $p = 400$ to just under 0.01. Over the same stretch, the clean part falls from roughly 8–60 at the peak to about 0.05 at $p = 400$, so most of the improvement past the threshold comes from approximating the target better.

Why does dividing by small singular values, the very mechanism that created the spike at $p = 40$, become nearly harmless at $p = 400$? By Equation 36.131, a noise component along $\mathbf{u}_i$ enters the weights scaled by $1/\sigma_i$ along $\mathbf{v}_i$, and shifts a test prediction in proportion to the test feature's component along $\mathbf{v}_i$. Using $\|\Phi_{\text{te}}\Phi^+\|_F^2 = \sum_i \|\Phi_{\text{te}}\mathbf{v}_i\|^2/\sigma_i^2$ together with $\sigma_i^2 = \|\Phi\mathbf{v}_i\|^2$, the noise part for $p \geq n$ takes the illuminating form

$$\sigma_\xi^2 \cdot \frac{1}{n}\sum_{i=1}^{n} \frac{\|\Phi_{\text{te}}\mathbf{v}_i\|^2/n_{\text{te}}}{\|\Phi\mathbf{v}_i\|^2/n},$$

the noise variance multiplied by the average ratio of mean-squared test feature energy to mean-squared training feature energy along each right singular direction $\mathbf{v}_i$. A tiny singular value σ_i only causes trouble when test inputs carry substantial energy in that same direction. At $p = n$, a tiny $\sigma_{\min}$ is an accident of the forty training points: along $\mathbf{v}_q$, the test features are typically thousands of times larger in mean square than the training features. At $p = 400$, by contrast, a small σ_i marks a direction where the features are small for training and test points alike. The final two columns of the table confirm this: `ratio_q` (along $\mathbf{v}_q$) is only about 2, and `ratio` (averaged over all directions) sits just below 1, so the small denominator is cancelled by a small numerator. This is the sense in which overfitting is called *benign*. Notice that benign means *bounded*, not zero: at $p = 400$, the predictions still carry roughly the variance of a single noisy training label.

Bartlett et al. (2020) make this phenomenon precise for linear regression through two effective ranks of the input covariance matrix, which exercise 7 computes directly. In deep neural networks, the features themselves are learned and their spectrum evolves throughout training, so a first-principles theory that predicts a trained network's test error remains out of reach. Every formal bound in this section covers the worst function in a class chosen before seeing the data, whereas the predictor we deploy is the specific solution an optimizer finds. Bridging that gap, with guarantees that depend on the algorithm and the data rather than on the function class alone, is the central open question this section leaves behind.

36.10.4 Summary

A concentration bound for a single function fixed before the data does not cover a model chosen because it scored best on the sample. Uniform convergence closes that gap by bounding the estimation error of every function in the class simultaneously. For a finite class $\mathcal{F}$, Hoeffding's inequality and the union bound achieve this at a cost of $\log|\mathcal{F}|/(2t^2)$ additional examples at tolerance t. For infinite classes, symmetrization with a ghost sample reduces the worst-case gap to correlation with random signs, adding twice the Rademacher complexity of the loss class to the empirical risk. For norm-bounded linear models ($\|\mathbf{w}\|_2 \le B$, $\|\mathbf{x}_i\|_2 \le r$), this complexity is at most $Br/\sqrt{n}$, independent of dimension. Uniform convergence reaches its boundary on classes flexible enough to interpolate arbitrary sign patterns, such as deep networks that fit random labels, where the bound becomes vacuous. Our random-features sweep illustrates why generalization can still succeed in that regime: test error spikes where $\sigma_{\min}$ collapses and falls as the minimum-norm interpolant shrinks, properties of the learned solution rather than of the whole function class.

36.10.5 Exercises

1. **McDiarmid's inequality.** ● The inequality of McDiarmid (1989) extends Hoeffding's from sums to functions with *bounded differences*. Suppose that changing only the i-th argument of $g(x_1, \dots, x_n)$ moves its value by at most c_i, and that $X_1, \dots, X_n$ are independent. Then

$$P(g(X_1, \dots, X_n) - \mathbb{E}[g(X_1, \dots, X_n)] \ge t) \le \exp\left(-\frac{2t^2}{\sum_{i=1}^n c_i^2}\right), \tag{36.132}$$

   and the same bound holds for the lower tail. Take this statement on faith; its proof uses martingales.

   1. Recover Hoeffding's inequality for an average of independent terms $X_i \in [a_i, b_i]$ from it.
   2. Let $\Phi(S) = \sup_{g\in\mathcal{G}}(R(g) - \hat{R}_S(g))$ for a loss class $\mathcal{G}$ with values in $[0,1]$. Show that Φ has bounded differences $c_i = 1/n$. Combine McDiarmid's inequality with the symmetrization bound $\mathbb{E}_S[\Phi(S)] \le 2\mathfrak{R}_n(\mathcal{G})$ of the text and derive Equation 36.126.
   3. For data in $[0,1]$, the sample mean has bounded differences $1/n$. Construct a sample of odd size n in $[0,1]$ in which changing one point moves the sample median by 1. What does this imply for using McDiarmid's inequality to put an error bar on the bootstrap estimate of the median in Section 36.5?

2. **Massart's lemma and ℓ_1 balls.** ● Massart's lemma for finite classes and the Rademacher complexity of ℓ_1-bounded linear classes rest on one bound for the expected maximum of sub-Gaussian variables.

   1. Let $Z_1, \dots, Z_N$ be centered and sub-Gaussian with variance proxy σ^2, that is, $\mathbb{E}[e^{\lambda Z_j}] \le e^{\lambda^2\sigma^2/2}$ for all λ, but not necessarily independent. Show that $\mathbb{E}[\max_j Z_j] \le \sigma\sqrt{2\log N}$ by bounding $e^{\lambda\mathbb{E}[\max_j Z_j]}$ by $\sum_j \mathbb{E}[e^{\lambda Z_j}]$ and optimizing over $\lambda > 0$. Deduce Massart's lemma, $\widehat{\mathfrak{R}}_S(\mathcal{F}) \le \sqrt{2\log|\mathcal{F}|/n}$, for a finite class with values in $[-1,1]$.

2. Let $\mathcal{F} = \{\mathbf{x} \mapsto \langle \mathbf{w}, \mathbf{x} \rangle : \|\mathbf{w}\|_1 \le B\}$ and $\|\mathbf{x}_i\|_\infty \le r$ for all i. Follow the proof of Equation 36.128, with the duality $\sup_{\|\mathbf{w}\|_1 \le B} \langle \mathbf{w}, \mathbf{v} \rangle = B\|\mathbf{v}\|_\infty$ in place of Cauchy–Schwarz, and show that $\widehat{\mathfrak{R}}_S(\mathcal{F}) \le Br\sqrt{2\log(2d)/n}$. Why does the dimension d now enter, and why only logarithmically?
3. Let the inputs lie in $\{-1, +1\}^d$ and let the target weight vector have k nonzero entries, each equal to ± 1. Compare the bound of (b) with Equation 36.128, each with the smallest radius B that contains the target. For which k is the ℓ_1 bound smaller when $d = 10^4$?

3. **Margin bound.** ● Let $\mathcal{F} = \{\mathbf{x} \mapsto \langle \mathbf{w}, \mathbf{x} \rangle : \|\mathbf{w}\|_2 \le B\}$ with $\|\mathbf{x}\|_2 \le r$, let the labels be $y \in \{-1, +1\}$, and let the ramp loss $\phi_\gamma(u) = \min\{1, \max\{0, 1 - u/\gamma\}\}$ with a fixed $\gamma > 0$ act on the margin $u = yf(\mathbf{x})$.

1. Show that $\phi_\gamma(yf(\mathbf{x}))$ is at least the 0-1 loss $\mathbf{1}\{yf(\mathbf{x}) \le 0\}$ and that ϕ_γ is $(1/\gamma)$-Lipschitz. Show that multiplying each prediction $f(\mathbf{x}_i)$ by a fixed label y_i leaves $\widehat{\mathfrak{R}}_S(\mathcal{F})$ unchanged.
2. Combine (a), the contraction lemma Equation 36.129, Equation 36.126, and Equation 36.128 into Equation 36.130. For $n = 50{,}000$ and $\delta = 0.05$, find the largest ratio Br/γ for which the complexity and confidence terms together stay below 0.1.
3. The bound holds for a radius B fixed before the data are drawn, whereas the norm of a trained weight vector is known only afterwards. Apply the bound with radius $B_j = 2^j$ and confidence $\delta_j = \delta/(j(j+1))$ for every $j \ge 1$, and take a union bound over j. State the bound that holds for a weight vector with $\|\mathbf{w}\| > 1$ chosen after seeing the data. Show that the freedom to choose the radius costs at most a factor 2 in the complexity term and an additional term of order $\sqrt{\log\log \|\mathbf{w}\|/n}$.

4. code **Fitting random signs.** ● The second cell of Section 36.10.2.4 fits $2{,}000$ sign vectors $\varepsilon \in \{-1, +1\}^n$ with $\mathbf{w} = \Phi^+ \varepsilon$ on $p = 2n$ standard Gaussian features and measures the norms of the fits.

1. Let $p < n$ and let Φ have full column rank. Show that the correlation $\frac{1}{n}\sum_i \varepsilon_i (\Phi\mathbf{w})_i$ has expectation p/n over the signs. The matrix $\Phi\Phi^+$ is a projection.
2. Let $p > n$, let r be the largest row norm of Φ, and let B be large enough that the class $\{\|\mathbf{w}\| \le B\}$ contains an interpolant of every sign vector. Explain why $Br/\sqrt{n}$ cannot then be smaller than 1. For a standard Gaussian Φ with $p > n+1$, $\mathbb{E}[(\Phi\Phi^\top)^{-1}] = \mathbf{I}_n/(p-n-1)$. Use this identity to predict the typical $\|\mathbf{w}\|$, and the typical value of $\|\mathbf{w}\|$ times a typical row norm of Φ divided by $\sqrt{n}$, as functions of p and n.
3. Rerun the cell with $n = 50$ for $p \in \{10, 25, 45, 55, 100, 400, 1000\}$ (for $p < n$ print the correlation instead of the norm) and compare with (a) and (b). What happens to the capacity value of (b) as $p \to \infty$, and why does the version with the largest row norm stay above 1?

5. code **Leaderboard attack.** ●● A leaderboard reports the 0-1 loss of each submitted label vector on a holdout set of $n = 4{,}000$ examples with labels $\mathbf{y} \in \{0, 1\}^n$. The *boosting attack* of Blum and Hardt (2015) uses no knowledge of the task: submit k uniformly random label vectors, keep those with reported loss at most $\frac{1}{2}$, and submit their coordinatewise majority vote. On fresh data, the majority vote has loss $\frac{1}{2}$.

1. Implement the attack for uniformly random $\mathbf{y}$ and $k \in \{10, 30, 100, 300, 1000, 3000\}$, and record the reported loss of the majority vote. Compare its distance from $\frac{1}{2}$ with the deviation that Equation 36.124 allows for a pre-registered pool of $k+1$ models at $\delta = 0.05$. From which k on does the attack exceed this deviation, and which hypothesis of the proposition does it violate?
2. Blum and Hardt (2015) prove that the reported loss is $\frac{1}{2} - \Omega(\sqrt{k/n})$ but do not give the constant. Derive it heuristically with Gaussian approximations: estimate the expected excess agreement with $\mathbf{y}$ of a kept vector, then the probability that the majority of about $k/2$ such vectors is correct on one coordinate. Compare the constant with the measurements of (a).
3. The *Ladder* mechanism of Blum and Hardt (2015) releases a new score only when a submission beats the best released score by more than a step η, rounds it to a multiple of η, and otherwise repeats the previous score. Against the Ladder, the attacker can only keep the vectors that lowered the released score. Implement both for $\eta \in \{0.005, 0.01\}$ and $k \in \{100, 1000, 3000\}$. How far below $\frac{1}{2}$ does the majority vote now get, and how does this compare with the deviation of (a)?

6. code **The square case.** ● Use the sweep of the double-descent experiment in Section 36.10.3.2.

   1. Show from the stored norms that the nesting monotonicity fails below the threshold, and explain why the padding argument does not apply there.
   2. At $p = n$, decompose $\mathbf{w}$ along Equation 36.131. Report, as a median over trials, the coefficient $|\langle \mathbf{u}_n, \mathbf{y} \rangle| / \sigma_{\min}$ of the smallest singular direction next to $\|\mathbf{w}\|$. Which share of the norm does this one direction carry?
   3. Rerun the sweep with `noise = 0`. Then replace the teacher by a target $y = \langle \mathbf{c}, \phi_{1:10}(\mathbf{x}) \rangle$ that is an exact combination of the first ten features, with $\|\mathbf{c}\| = 1$ and no noise. Describe the two curves. In each case, what does $\sigma_{\min}$ divide? Why does the minimum-norm interpolant become *worse* as features are added when the target uses only ten of them?
   4. Add label noise of standard deviation 0.1 to the target of (c). Where is the test error now highest, how does the peak compare with the noise variance, and where is the error lowest?

7. [extended] **Benign spectra.** ●● Bartlett et al. (2020) characterize when the minimum-norm interpolant in linear regression has small excess risk, the mechanism that Section 36.10.3.3 describes in words. Let $\mathbf{x} \sim \mathcal{N}(\mathbf{0}, \Sigma)$ with $\Sigma = \operatorname{diag}(\lambda_1, \dots, \lambda_p)$ and $\lambda_1 \geq \lambda_2 \geq \cdots$, let $y = \langle \theta^*, \mathbf{x} \rangle + \xi$ with $\xi \sim \mathcal{N}(0, \sigma^2)$, stack $n < p$ training inputs into $\mathbf{X} \in \mathbb{R}^{n \times p}$, and let $\widehat{\theta} = \mathbf{X}^{+}\mathbf{y}$. The excess risk is $(\widehat{\theta} - \theta^*)^{\top} \Sigma (\widehat{\theta} - \theta^*)$.

   1. Show that its expectation over the noise is the sum of a bias term $\theta^{*\top}(\mathbf{I} - \mathbf{P})\Sigma(\mathbf{I} - \mathbf{P})\theta^*$, where $\mathbf{P} = \mathbf{X}^{+}\mathbf{X}$, and a variance term $\sigma^2 \operatorname{tr}\big((\mathbf{X}^{+})^{\top} \Sigma \mathbf{X}^{+}\big)$. Implement both terms; no test set is needed.
   2. The paper's effective ranks are $\rho_k = \sum_{i>k} \lambda_i / \lambda_{k+1}$ and $\varrho_k = \big(\sum_{i>k} \lambda_i\big)^2 / \sum_{i>k} \lambda_i^2$. With k^* the smallest k such that $\rho_k \geq bn$, the variance term is of order $\sigma^2 (k^*/n + n/\varrho_{k^*})$ up to constants; take $b = 1$. Use $n = 100$, $\sigma = 1$, five strong directions with $\lambda_i = 1$ that contain a unit-norm θ^*, and q weak directions with $\lambda_i = 1/q$ each, for $q \in \{100, 300, 10^3, 3 \cdot 10^3, 10^4\}$. Compare the measured variance with $\sigma^2 (k^*/n + n/\varrho_{k^*})$. What happens at $q = 100$, and

why?

3. Repeat with power-law spectra $\lambda_i = i^{-\alpha}$ truncated at $p = 40n$, for $\alpha \in \{0.5, 1, 2\}$, $\theta^* = \mathbf{e}_1$, and $n \in \{50, 100, 200, 400\}$. For which spectra does the excess risk decrease with n? For the others, does the bias or the variance fail, and how do ρ_0/n, k^*/n, and n/ϱ_{k^*} show it?

8. **Norms, not widths: a two-layer network.** ●● Let $\mathcal{F}_1 = \{\mathbf{x} \mapsto \mathrm{ReLU}(\langle \mathbf{v}, \mathbf{x}\rangle) : \|\mathbf{v}\|_2 \le B_2\}$ and $\mathcal{F}_2 = \{\mathbf{x} \mapsto \sum_{j=1}^{h} w_j \,\mathrm{ReLU}(\langle \mathbf{v}_j, \mathbf{x}\rangle) : \|\mathbf{w}\|_1 \le B_1,\ \|\mathbf{v}_j\|_2 \le B_2\}$, on points with $\|\mathbf{x}_i\| \le r$.

   1. Show that $\widehat{\mathfrak{R}}_S(\mathcal{F}_1) \le B_2 r/\sqrt{n}$.
   2. Show that $\mathcal{F}_2$ is contained in B_1 times the convex hull of $\mathcal{F}_1 \cup (-\mathcal{F}_1)$, and that taking the convex hull of a class does not change its empirical Rademacher complexity. Hint: $\sum_j w_j f_j = \|\mathbf{w}\|_1 \sum_j \frac{|w_j|}{\|\mathbf{w}\|_1}(\pm f_j)$.
   3. Using $0 \in \mathcal{F}_1$, show that $\mathbb{E}_\varepsilon \sup_{f \in \mathcal{F}_1} |\sum_i \varepsilon_i f(\mathbf{x}_i)| \le 2n\,\widehat{\mathfrak{R}}_S(\mathcal{F}_1)$, and conclude that $\widehat{\mathfrak{R}}_S(\mathcal{F}_2) \le 2B_1 B_2 r/\sqrt{n}$.
   4. The width h does not appear in the bound. What does the bound say about weight decay on both layers? Why does it not explain the random-label experiment of Section 4.5.1.1?

36.11 Naive Bayes

In Section 3.3.5, we trained softmax regression on Fashion-MNIST by running gradient descent directly on the probability of the label given the image. That fit produced 7,850 parameters (one weight for each pixel and class, plus ten offsets), achieved about 83% accuracy, and passed the calibration check of Section 3.4. Yet gradient descent is not the only way to build a classifier. We can instead describe how each class generates its images, capturing what a typical coat or sandal looks like, and let Bayes' rule invert that description into a prediction. How well does this alternative work, and what do we give up by choosing it?

The immediate obstacle is scale. Describing how coats look means assigning a probability to every possible picture. Even if we simplify each pixel to a single bit (inked or blank), there are far more possible 28×28 pictures than atoms in the observable universe, while we have only 60,000 training images. The *naive Bayes* classifier sidesteps this obstacle with a single bold assumption: once the class is known, the features tell us nothing about one another.

This idea goes back to Maron's early work on sorting documents by the words they contain (Maron 1961), and it powered the spam filters of the 1990s (Sahami et al. 1998). Under the independence assumption, a spam filter needs to know only how often each word appears in spam versus ordinary email, and a garment classifier needs only how often each pixel is inked in each class. Training reduces to counting, and prediction reduces to multiplying.

Naive Bayes is the simplest *generative* classifier, built from a model of how each class produces its inputs. It fits in a single pass through the data without an optimizer, and it handles missing features cleanly by dropping their factors. Its independence assumption is obviously false for images, as it is for most real-world data, and no other model illustrates as clearly what a false independence assumption does to a classifier's decisions and probabilities.

This section serves as the chapter's case study. After the generalization bounds of Sec-

tion 36.10, the mathematics here is light; the value lies in bringing the chapter's tools together on one concrete model. Maximum likelihood fits its parameters, a Beta posterior smooths them, the bootstrap puts an error bar on its accuracy, and a reliability check tests its probabilities.

The implementation below uses plain NumPy; the deep learning framework supplies only the data loader.

```
d2l.use_svg_display()
```

36.11.1 Bayes' Rule for Classification

There are two routes to the probability of a label given an input. Earlier in the book we learned that probability directly. Here we take the indirect route through a model of the inputs, see why it is intractable in full generality, and make it workable with the assumption that gives the classifier its name.

36.11.1.1 The Generative Route

A classifier assigns a label $y \in \{1, \dots, K\}$ to an input $\mathbf{x}$. If we knew the true posterior $p(y \mid \mathbf{x})$ (the probability of each label given the input), then the decision rule that is right most often simply picks the most probable label,

$$\hat{y} = \operatorname*{argmax}_y \, p(y \mid \mathbf{x}). \tag{36.133}$$

For Fashion-MNIST, $K = 10$. If each of the 784 pixels is recorded as inked or blank, then $\mathbf{x}$ is one of 2^{784} binary patterns. Because $784 \log_{10} 2 = 236.0$, that amounts to roughly 10^{236} distinct images. A lookup table of $p(y \mid \mathbf{x})$ would need a row for every pattern, and 60,000 training images leave virtually all of them empty.

Bayes' rule (Section 36.2) rewrites the posterior as $p(y \mid \mathbf{x}) = p(\mathbf{x} \mid y)\, p(y)/p(\mathbf{x})$. Since the denominator $p(\mathbf{x})$ does not depend on y, it cannot change which class achieves the maximum:

$$\hat{y} = \operatorname*{argmax}_y \, p(\mathbf{x} \mid y)\, p(y). \tag{36.134}$$

For each class, the classifier multiplies how probable the image is under that class by how common the class is overall. We call this product $p(\mathbf{x} \mid y)\, p(y)$ the *Bayes numerator*. Dividing the K numerators by their sum restores $p(\mathbf{x})$ and recovers the full posterior.

This splits classification into two broad families. A *generative classifier* models how the inputs of each class arise, combining the class-conditional distribution $p(\mathbf{x} \mid y)$ with the prior $p(y)$, and then applies Bayes' rule. A *discriminative classifier* models $p(y \mid \mathbf{x})$ directly and never describes how the inputs themselves are generated; softmax regression (Section 3.1) is the prototypical example.

You might reasonably object that Bayes' rule has not solved the scaling problem: $p(\mathbf{x} \mid y)$ is still a distribution over 2^{784} patterns. That objection is right, and the chain rule of probability pinpoints where the combinatorial explosion lives. Expanding one feature at a time,

$$p(\mathbf{x} \mid y) = p(x_1 \mid y)\, p(x_2 \mid x_1, y) \cdots p(x_d \mid x_1, \ldots, x_{d-1}, y). \tag{36.135}$$

Each factor conditions on the entire history of earlier features, so the final factor alone requires a table over 2^{d-1} histories. The next subsection drops those histories altogether.

36.11.1.2 The Naive Assumption

To estimate $p(\mathbf{x} \mid y)$ from a realistic dataset, we need far fewer parameters. The naive assumption states, in plain words, that once we know the garment class, learning that one pixel is inked tells us nothing further about any other pixel. In symbols, the features are *conditionally independent given the label*, so every factor in the chain rule forgets its history:

$$p(\mathbf{x} \mid y) = \prod_{i=1}^{d} p(x_i \mid y). \tag{36.136}$$

For binary pixels, each factor is determined by a single number: the probability that pixel i is inked in class y. An unrestricted $p(\mathbf{x} \mid y)$ requires $2^{784} - 1$ free probabilities per class, whereas the factorized model requires only 784. Each of those 784 numbers is an ordinary frequency, which 6,000 training images per class estimate with a standard error of at most $0.5/\sqrt{6000} \approx 0.0065$ (Section 36.5). Both models also need $K - 1 = 9$ numbers for the class prior.

Notice that this assumption is a joint condition on all d features at once; checking pairs of features is not enough to verify it. Within a single class, suppose x_1 and x_2 are independent fair bits and $x_3 = x_1 \oplus x_2$ equals 1 when exactly one of them is 1. Every pair of features is independent, yet the third feature is completely determined by the other two. As a result, the pattern $(1, 1, 1)$ has true probability 0, while the product of the three marginals assigns it $\frac{1}{8}$ (exercise 1).

Drawn as a directed graph with an arrow from each variable to those it directly influences, the factorization forms a star: the label y is the parent of every feature, and no edges connect the features to one another (Figure 36.50, left). A richer model that preserves conditional dependencies among features adds edges between them (right).

Substituting the factorization Equation 36.136 into Bayes' rule yields the *naive Bayes rule* (Maron 1961):

$$\hat{y} = \operatorname*{argmax}_{y} \; p(y) \prod_{i=1}^{d} p(x_i \mid y). \tag{36.137}$$

For images, this assumption is clearly false. Neighboring pixels of a garment are inked together or blank together because they belong to the same sleeve or the same hem. Below we will measure just how badly the assumption fails on Fashion-MNIST, and what that failure does to the classifier.

36.11.1.3 A Two-Word Spam Filter

Before scaling up to 784 pixels, let us work through the smallest possible example by hand. Consider an email filter that inspects only two binary features: whether a message contains

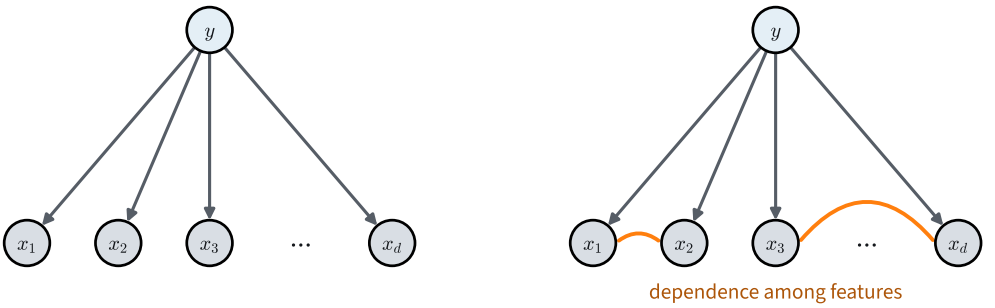


Figure 36.50: Conditional-independence structures. Left: in naive Bayes, the label y is a parent of every feature and no edges connect the features, representing $p(\mathbf{x} \mid y) = \prod_i p(x_i \mid y)$. Right: a model that retains conditional dependence among features adds edges between them.

the word “free” and whether it contains “meeting”. Suppose our training set has ten messages (4 spam and 6 ordinary) with the following counts:

	spam (4 emails)	ordinary (6 emails)
contains “free”	3	1
contains “meeting”	0	4

Simple counting supplies every number the model needs. The prior is $\hat{p}(\text{spam}) = 4/10 = 0.4$. The word “free” appears in $3/4$ of the spam and $1/6$ of the ordinary emails, while “meeting” appears in $0/4$ of the spam and $4/6$ of the ordinary emails. Now suppose a new message contains “free” and does not contain “meeting”. Its two Bayes numerators are

$$\text{spam: } 0.4 \cdot \tfrac{3}{4} \cdot (1 - \tfrac{0}{4}) = 0.30, \qquad \text{ordinary: } 0.6 \cdot \tfrac{1}{6} \cdot (1 - \tfrac{4}{6}) = \tfrac{1}{30} \approx 0.033,$$

which gives a posterior spam probability of $0.30/(0.30 + 0.033) = 0.90$. The same calculation is even more transparent in terms of odds. Each observed feature multiplies the prior odds by its likelihood ratio—the probability of what we saw under spam divided by its probability under ordinary email:

$$\frac{p(\text{spam} \mid \mathbf{x})}{p(\text{ordinary} \mid \mathbf{x})} = \underbrace{\frac{0.4}{0.6}}_{\text{prior}} \cdot \underbrace{\frac{3/4}{1/6}}_{\text{“free”}} \cdot \underbrace{\frac{1}{2/6}}_{\text{no “meeting”}}$$
$$= \frac{2}{3} \cdot 4.5 \cdot 3 = 9.$$

Odds of 9 to 1 correspond to a probability of $9/10$. The 784-pixel classifier we build below performs this exact same calculation, only with 784 likelihood ratios instead of two. We will return to this tiny filter three more times: when a count hits zero, when we move to logarithms, and when a feature is counted twice.

36.11.2 Learning by Counting

The naive Bayes rule requires the class prior and one probability per class and feature. Where do these estimates come from, and how can a computer evaluate a product of 785 probabilities without losing them to numerical underflow?

36.11.2.1 Maximum Likelihood and Smoothing

In our spam filter, we simply plugged in observed frequencies. Maximum likelihood justifies that choice and also reveals where it breaks down. Write $\theta_{iy} = p(x_i = 1 \mid y)$ for the probability that feature i is active in class y. Suppose class y appears n_y times among n training examples, and feature i is active in n_{iy} of those examples. Under the naive Bayes factorization, the log-likelihood of the entire training set decomposes into one term for the class prior and one independent term for each class and feature:

$$\sum_y n_y \log p(y) + \sum_y \sum_{i=1}^{d} \Big[n_{iy} \log \theta_{iy} + (n_y - n_{iy}) \log(1 - \theta_{iy}) \Big]. \tag{36.138}$$

Each bracketed term is simply the log-likelihood of seeing n_{iy} heads in n_y flips of a coin with bias θ_{iy}, so every parameter can be fitted on its own. Since the maximum-likelihood estimate of a coin's bias is its observed frequency (Section 36.6), we obtain $\hat{\theta}_{iy} = n_{iy}/n_y$ and $\hat{p}(y) = n_y/n$.

Raw frequencies, however, carry a serious hazard. If feature i never appears in the training examples of class y, then $\hat{\theta}_{iy} = 0$, and any future example with feature i active receives a Bayes numerator of 0 for class y, regardless of what all its other features say. One feature vetoes the class. Our two-word spam filter already has such a zero: "meeting" never occurred in the four training spam emails, so a message containing both "free" and "meeting" gets a spam numerator of $0.4 \cdot \frac{3}{4} \cdot 0 = 0$ and is declared ordinary with complete certainty. We will check below how often a Fashion-MNIST pixel is never inked in a given class.

The standard remedy is *Laplace smoothing* (Laplace 1814): add one imagined example with the feature active and one with it inactive, yielding

$$\hat{\theta}_{iy} = \frac{n_{iy} + 1}{n_y + 2}. \tag{36.139}$$

This nudges every estimate slightly toward $\frac{1}{2}$ and guarantees that no estimated probability is ever 0 or 1.

Laplace smoothing also has a clean Bayesian interpretation. Under a uniform prior on θ, observing n_{iy} successes in n_y trials yields the posterior $\text{Beta}(n_{iy}+1,\ n_y - n_{iy} + 1)$, whose mean is precisely $(n_{iy}+1)/(n_y+2)$ (Section 36.4); Laplace called this the rule of succession.

Returning to the two-word spam filter, smoothing changes the estimated probabilities of "free" and "meeting" to $4/6$ and $1/6$ in spam, and to $2/8$ and $5/8$ in ordinary email, while leaving the prior at 0.4. The message containing both words now has Bayes numerators $0.4 \cdot \frac{4}{6} \cdot \frac{1}{6} = 0.044$ and $0.6 \cdot \frac{2}{8} \cdot \frac{5}{8} = 0.094$, giving a spam probability of 0.32 instead of 0. Meanwhile, the first message ("free" without "meeting") drops from 0.90 to 0.80, reflecting the fact that four spam emails are modest evidence that spam never mentions a meeting.

36.11.2.2 Scores in Log Space

Even with smoothing, evaluating the naive Bayes rule directly multiplies $d + 1$ numbers between 0 and 1, and for an image that product is vanishingly small. If every factor were $\frac{1}{2}$, the product of 784 of them would be $2^{-784} \approx 10^{-236}$. The smallest positive single-precision floating-point number is about $1.4 \cdot 10^{-45}$ and the smallest positive double-precision number is about $4.9 \cdot 10^{-324}$, so the product immediately underflows to 0 in single precision. Double precision survives 2^{-784}, but not the real image scores we will compute below, some of which drop near 10^{-1000}.

Taking logarithms turns the product into a sum. Because log is strictly increasing, it leaves the winning class unchanged (Section 35.8 discusses this technique in general). Defining the *class score* $s_y(\mathbf{x})$ as the logarithm of the Bayes numerator, the decision rule becomes

$$\hat{y} = \operatorname*{argmax}_y \; s_y(\mathbf{x}), \qquad s_y(\mathbf{x}) = \log p(y) + \sum_{i=1}^{d} \log p(x_i \mid y). \tag{36.140}$$

A sum of 785 logarithms replaces a product of 785 probabilities, and every term stays finite as long as no probability is 0, which smoothing guarantees. For the unsmoothed spam filter, the log scores of the first message are $\log 0.30 = -1.20$ and $\log \frac{1}{30} = -3.40$; their difference, $2.20 = \log 9$, is simply the logarithm of the posterior odds we computed by hand.

36.11.2.3 Linear Decision Boundaries

For binary features, the class score simplifies into a remarkably familiar form, and that form reveals which classifiers naive Bayes can represent. A binary feature contributes $\log \theta_{iy}$ when it is active and $\log(1 - \theta_{iy})$ when it is inactive, so $\log p(x_i \mid y) = x_i \log \theta_{iy} + (1 - x_i) \log(1 - \theta_{iy})$. Grouping the terms that multiply x_i and the terms that do not gives

$$s_y(\mathbf{x}) = \mathbf{w}_y^\top \mathbf{x} + b_y, \qquad w_{iy} = \log \frac{\theta_{iy}}{1 - \theta_{iy}}, \qquad b_y = \log p(y) + \sum_{i=1}^{d} \log(1 - \theta_{iy}). \tag{36.141}$$

Each inked pixel adds w_{iy}, the log-odds that pixel i is inked in class y, and the offset b_y is the score of a completely blank image. Because every class score is affine in $\mathbf{x}$, the decision boundary between any two classes, where their scores tie, is a hyperplane (Bishop 2006). This is the first of two results in this section worth remembering: Bernoulli naive Bayes is a linear classifier in the very same family as softmax regression, with parameter arrays of the exact same shapes (784×10 weights and 10 offsets). The two methods differ only in how they choose those numbers, by counting frequencies or by running gradient descent on $p(y \mid \mathbf{x})$. For two classes, the posterior reduces to a sigmoid of the score difference, $p(y = 1 \mid \mathbf{x}) = \text{sigmoid}\big((\mathbf{w}_1 - \mathbf{w}_0)^\top \mathbf{x} + b_1 - b_0\big)$, which is identical in form to logistic regression (exercise 2).

Whether the boundary stays linear depends on the feature distribution. Gaussian naive Bayes models each continuous feature x_i in class y as a Gaussian with mean μ_{iy} and variance σ_{iy}^2, contributing $-(x_i - \mu_{iy})^2/(2\sigma_{iy}^2)$ to the class score. If every feature shares the

same variance across all classes, the x_i^2 terms are identical in every score, cancel in every pairwise comparison, and leave a linear boundary. With class-specific variances, the x_i^2 terms survive and the boundary becomes quadratic in each x_i, though still without cross terms $x_i x_j$ (exercise 3).

The Gaussian case lets us visualize how much the fitting method matters. In Figure 36.51, the two features are strongly correlated within each class, tilting both point clouds diagonally. Naive Bayes fits each class with an axis-aligned Gaussian (equal variances in both classes, so its boundary is a straight line). Because axis-aligned ellipses ignore the tilt, the resulting boundary cuts across both clouds and misclassifies 19% of points. The right panel shows the linear boundary obtained by fitting $p(y \mid \mathbf{x})$ directly with enough data. It aligns with the tilt and achieves a 7% error rate, which is optimal for this problem: no classifier of any kind can do better.

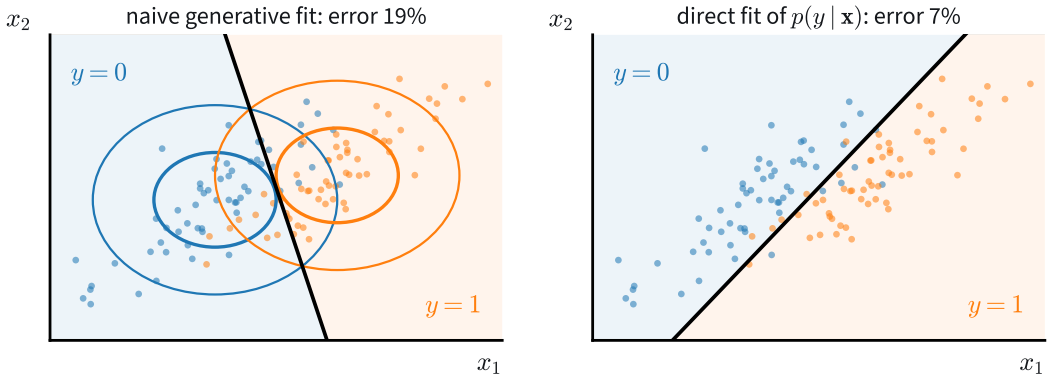


Figure 36.51: Two routes to a linear classifier on the same data, where the two features are correlated within each class. Left: the naive generative fit models each class as a Gaussian with independent features (ellipses at one and two standard deviations) and derives the boundary via Bayes' rule. Right: the linear boundary fitted directly to $p(y \mid \mathbf{x})$. Error rates are population values for equal priors.

Historically, text rather than pixels has been the flagship application of naive Bayes: a real spam filter is just our two-word example scaled to tens of thousands of words (Manning et al. 2008). A document can be represented either by which words appear or by how many times each word occurs, and the class score is linear in both cases. Here we stay with pixels; exercise 4 develops both text event models and shows how they can disagree on the same document.

36.11.3 Fashion-MNIST by Counting

A two-word spam filter is easy to check by hand. Whether the same simple counting still works on 784 visibly correlated pixels can only be settled by trying it, and Fashion-MNIST lets us compare the outcome directly against softmax regression on the very same images.

36.11.3.1 Binary Pixels

Fashion-MNIST (Xiao et al. 2017) consists of 60,000 training and 10,000 test images across ten garment classes, each a 28×28 grayscale grid (Section 3.2). Because background pixels

are strictly 0, we can turn each image into a binary vector by setting $x_i = 1$ wherever pixel i is positive (part of the garment) and $x_i = 0$ on the background. The loader provides the raw images for both splits, which we flatten into vectors of 784 binary features.

```
data = d2l.FashionMNIST()
def binarize(images):  # 1 where the pixel belongs to the garment
    return (images.reshape(-1, 784) > 0).astype(np.float32)

X, Y = binarize(data.train.data.numpy()), data.train.targets.numpy()
X_test, Y_test = binarize(data.val.data.numpy()), data.val.targets.numpy()
print(X.shape, X_test.shape, f'ink fraction {X.mean():.2f}')
```

```
(60000, 784) (10000, 784) ink fraction 0.50
```

Roughly half of all pixels are inked. Every class has 6,000 training images and 1,000 test images, so the estimated prior is a uniform 0.1 and adds the same constant to every class score; we keep the prior term in the code so the implementation remains general.

36.11.3.2 Templates and Samples

Training requires just two counts: the number of images in each class, n_y, and the number of those images in which each pixel is inked, n_{iy}. The cell below stores the Laplace-smoothed estimates $\widehat{\theta}_{iy}$ in `P_xy`, one row per class. At the same time, it answers the practical question we raised about smoothing: how many pixels are never inked in the training images of a class, and how many test images happen to ink one of them?

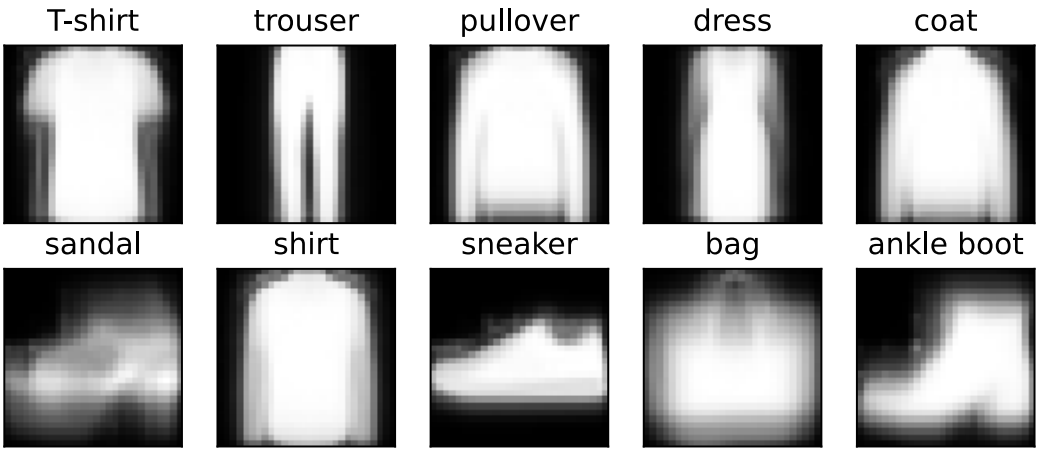
```
n_y = np.bincount(Y, minlength=10)              # images per class
n_x = np.stack([X[Y == y].sum(axis=0) for y in range(10)])
P_y = n_y / n_y.sum()                           # prior p(y)
P_xy = (n_x + 1) / (n_y + 2)[:, None]           # smoothed theta_iy
print('prior:', P_y)
print('never-inked pixels per class:', (n_x == 0).sum(axis=1))
unseen = (X_test * (n_x[Y_test] == 0)).sum(axis=1) > 0
print('test images inking one of their class:', unseen.sum())
```

```
prior: [0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1]
never-inked pixels per class: [ 0 13  1  0  3  3  0 57  0  1]
test images inking one of their class: 5
```

Zero counts are fairly rare here: four of the ten classes have none at all, trousers have 13, and sneakers have 57 (since sneakers are flat and leave the top of the frame blank). Across the entire test set, only 5 images ink a pixel that their true class never inked during training. Without smoothing, each of those five images would receive a score of $-\infty$ for its true class and be misclassified no matter how strongly the other 783 pixels supported it.

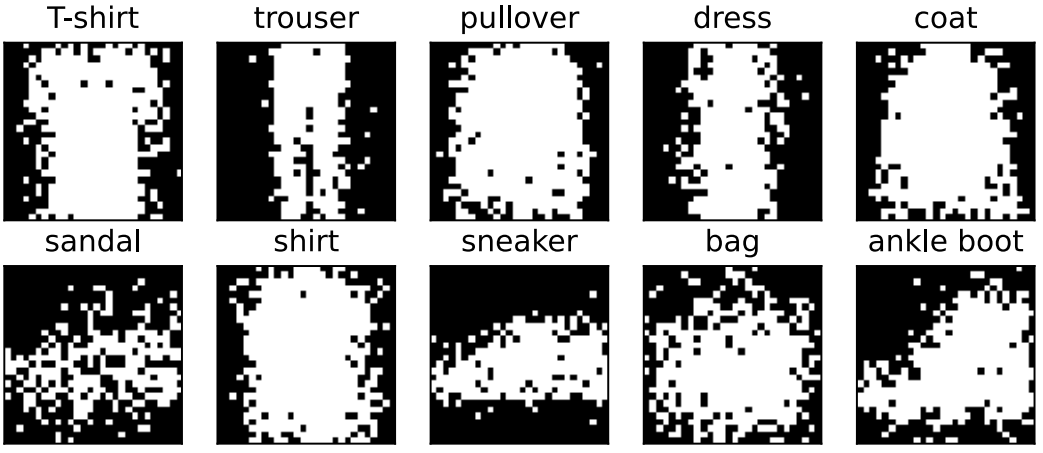
Because each row of `P_xy` records the fraction of training images in which each pixel is inked, we can reshape the ten rows back into 28×28 images to see what the model has learned for each garment class.

```
d2l.show_images(P_xy.reshape(10, 28, 28), 2, 5,
                titles=data.text_labels(range(10)), cmap='gray');
```



The resulting templates look like soft, blurred outlines of the ten garments. Because each template is simply the per-pixel average of its class, that blur reflects genuine variation in how garments are shaped and positioned. Any model that matches the per-pixel frequencies of the data, whether or not it assumes independence, produces the exact same templates; the blur belongs to the data, not to the assumption. What the naive assumption adds is the claim that these averages, together with the prior, are all there is to know. To see what that claim implies, we can draw synthetic images from the model by flipping an independent coin with probability $\hat{\theta}_{iy}$ for every pixel.

```
rng = np.random.default_rng()
samples = (rng.random((1000, 10, 784)) < P_xy).astype(np.float32)
d2l.show_images(samples[0].reshape(10, 28, 28), 2, 5,
                titles=data.text_labels(range(10)), cmap='gray');
```



Each sample captures the right overall silhouette, but its edges are ragged and speckled because every boundary pixel is decided by its own independent coin flip. We can quantify that visual difference directly. Scanning across a row of a real image, ink and background switch only a few times; in a model sample they should switch much more frequently. Below

we compare both the overall ink fraction and the average number of ink-to-background switches per row across the training set and across 1,000 generated samples per class.

```
def switches(images):  # ink/background changes along a row, mean per row
    return np.abs(np.diff(images.reshape(-1, 28, 28), axis=2)).sum(axis=2).mean()

print(f'ink fraction: data {X.mean():.3f}, samples {samples.mean():.3f}')
print('switches per row, data:   ',
      np.round([switches(X[Y == y]) for y in range(10)], 1))
print('switches per row, samples:',
      np.round([switches(samples[:, y]) for y in range(10)], 1))
```

```
ink fraction: data 0.498, samples 0.498
switches per row, data:    [3.8 3.2 3.1 3.1 3.1 3.  3.3 1.5 3.1 2.6]
switches per row, samples: [5.1 4.1 4.8 4.7 4.8 8.2 5.3 4.  8.2 5.9]
```

The samples carry the same overall amount of ink as the real data, as they must since every pixel has the right marginal probability. Yet they jump between ink and background far more often: roughly 5 switches per row instead of 3 for pullovers, and about 8 instead of 3 for sandals and bags. Real garments have smooth contours because neighboring pixels tend to agree, and that local agreement is precisely what the independence assumption discards.

36.11.3.3 Checking the Assumption

Generated samples show the independence assumption failing across the image as a whole; a single pair of pixels pins it down with concrete numbers. Consider two neighboring pixels along the bottom hem of pullovers, in row 26 at columns 14 and 15. If those two pixels were conditionally independent given the class, the fraction of pullovers with both inked would equal the product of their individual ink fractions. Let us compare that product with the actual joint frequency in both the training pullovers and the 1,000 sampled pullovers.

```
i, j = 26 * 28 + 14, 26 * 28 + 15     # two neighbouring hem pixels
for name, Z in (('data', X[Y == 2]), ('model', samples[:, 2])):
    p_i, p_j, both = Z[:, i].mean(), Z[:, j].mean(), (Z[:, i] * Z[:, j]).mean()
    print(f'{name:5s}: P(i)={p_i:.2f}  P(j)={p_j:.2f}  '
          f'P(i)P(j)={p_i * p_j:.2f}  P(i,j)={both:.2f}')
```

```
data : P(i)=0.55  P(j)=0.55  P(i)P(j)=0.30  P(i,j)=0.54
model: P(i)=0.54  P(j)=0.55  P(i)P(j)=0.30  P(i,j)=0.31
```

Each pixel is inked in roughly 55% of the pullovers. Under independence, both should be inked together in $0.55 \times 0.55 \approx 30\%$ of them. In the actual data, both are inked in 54% of pullovers—nearly as often as either pixel alone. A pullover's hem either reaches row 26 or stops short of it, so these two neighboring pixels act almost like a single feature recorded twice. The model's samples, by construction, match the product.

36.11.3.4 Test Accuracy

To classify the test set, we use the affine form Equation 36.141. We compute the weights w_{iy} and offsets b_y from `P_xy` and `P_y`, score every test image against all ten classes in a single matrix multiplication, and save the predictions for the analysis that follows. Softmax

regression reached about 83% accuracy with scores of this exact same form; how much do we give up by counting instead?

```
W = np.log(P_xy) - np.log1p(-P_xy)              # w_iy, log-odds per pixel
b = np.log1p(-P_xy).sum(axis=1) + np.log(P_y)   # b_y, score of a blank image
s = X_test @ W.T + b                            # s_y(x), shape (10000, 10)
pred = s.argmax(axis=1)
print(f'test accuracy {(pred == Y_test).mean():.4f}, training accuracy '
      f'{((X @ W.T + b).argmax(axis=1) == Y).mean():.4f}')
```

```
test accuracy 0.7059, training accuracy 0.7171
```

Naive Bayes classifies about 71% of the test images correctly, far above the 10% baseline of random guessing. Because counting involves no randomness, this number is completely deterministic from run to run; we will put an error bar on its sampling uncertainty shortly.

The training accuracy is just as informative. On the 60,000 images used to fit it, the model is right 72% of the time—only one percentage point higher than on unseen test images. A model that had overfit its training set would show a wide gap between the two; here there is almost none. The errors are therefore not caused by a shortage of training data. The model makes the same mistakes on its own training set as on new images because its assumptions are too restrictive: in the language of Section 36.5, the error is dominated by bias rather than variance. Collecting more images would barely move the needle. What we need is a better set of parameters, and our softmax regression comparison below will confirm that a much better linear classifier exists within the very same model family.

These log scores also let us check what would have happened had we evaluated the raw product Equation 36.137 directly. Since each score is the natural logarithm of a Bayes numerator, dividing by $\log 10$ gives its base-10 exponent. Let us compare the numerators against the smallest positive single-precision number and the smallest *normal* double-precision number, below which double precision begins to lose accuracy.

```
log10 = s / np.log(10)                    # decimal exponent of each numerator
print(f'log10 of the numerators: largest {log10.max():.0f}, smallest '
      f'{log10.min():.0f}, median best {np.median(log10.max(axis=1)):.0f}')
f32 = -149 * np.log(2)     # log of 2**-149, the smallest positive float32
f64 = -1022 * np.log(2)    # log of 2**-1022, the smallest normal float64
print(f'images whose best numerator underflows float32: '
      f'{(s.max(axis=1) < f32).mean():.2f}')
print(f'numerators below the smallest normal float64: {(s < f64).mean():.2f}, '
      f'images whose best one is: {(s.max(axis=1) < f64).mean():.4f}')
print(f'accuracy of the raw float64 product: '
      f'{(np.exp(s).argmax(axis=1) == Y_test).mean():.4f}')
```

```
log10 of the numerators: largest -32, smallest -1002, median best -81
images whose best numerator underflows float32: 0.96
numerators below the smallest normal float64: 0.44, images whose best one is: 0.0003
accuracy of the raw float64 product: 0.7058
```

The Bayes numerators span roughly 10^{-32} down to 10^{-1000}, with the winning class of a typical image sitting near 10^{-81}. For 96% of test images, even the largest of the ten numerators falls below the smallest positive single-precision float, so raw single-precision products

would see ten zeros and fail to classify them at all. In double precision, 44% of all numerators drop below the smallest normal float64, though almost all of those belong to losing classes. The largest numerator stays representable for all but three images in ten thousand, so the raw float64 product matches the log-space accuracy to within one image in ten thousand. Still, double precision survives 784 pixels only with modest headroom: at 10^{-81} for a typical winning score, an input four times as long would underflow float64 as well. Working in log space keeps every score comfortably within ordinary floating-point range.

36.11.4 Evaluating the Classifier

A headline accuracy of 71% immediately raises three questions. How precise is that estimate? Which confusions account for the remaining 29%? And can we trust the probabilities the classifier reports?

36.11.4.1 Error Bars

Our test accuracy is an average over 10,000 random test images, so a different test sample would produce a slightly different number. Each test image yields a single Bernoulli outcome, right or wrong. This is the same coin toss that opened the chapter, flipped once per test image, so the observed accuracy $\hat{p}$ is the sample mean of n flips and has standard error $\sqrt{\hat{p}(1-\hat{p})/n}$ (Section 36.5). For $\hat{p} = 0.706$ and $n = 10{,}000$,

$$\sqrt{\frac{0.706 \cdot 0.294}{10^4}} \approx 0.0046, \qquad 1.96 \times 0.0046 \approx 0.009,$$

giving a 95% confidence interval extending about 0.9 percentage points in either direction. The bootstrap (Section 36.5) provides a second, formula-free check: we resample the test images with replacement, recompute the accuracy on each resample, and take the middle 95% of the resulting distribution.

```
rng = np.random.default_rng()
correct = pred == Y_test                          # per-image 0/1 outcomes
n, p = len(correct), correct.mean()
boot = correct[rng.integers(0, n, (1000, n))].mean(axis=1)  # 1,000 resamples
print(f'accuracy {p:.4f}, bootstrap 95% interval '
      f'({np.percentile(boot, 2.5):.3f}, {np.percentile(boot, 97.5):.3f})')
print(f'binomial half-width 1.96 se = {1.96 * np.sqrt(p * (1 - p) / n):.4f}')
```

```
accuracy 0.7059, bootstrap 95% interval (0.698, 0.715)
binomial half-width 1.96 se = 0.0089
```

The bootstrap interval runs from about 0.70 to 0.715, matching the binomial half-width of 0.009. We should therefore report the accuracy as $71\% \pm 0.9$ points; the digits after the decimal point in "70.59%" are sampling noise. Note that when comparing two classifiers on the exact same test set, their errors are correlated, so a paired analysis of per-image differences is much sharper than comparing two separate confidence intervals (exercise 5).

36.11.4.2 Confusions and Weights

Overall accuracy tells us how often the classifier errs; the confusion matrix of Section 3.4 tells us *how* it errs. Following the convention of Section 3.3, rows correspond to predicted

classes and columns to true classes, so each column traces where the 1,000 test images of a given garment end up.

```
conf = np.zeros((10, 10), dtype=int)
np.add.at(conf, (pred, Y_test), 1)       # N[i, j]: class j predicted as i
print(conf)
```

```
[[748   9   8  50   2   0 218   0   0   0]
 [  0 907   0  53   3   0   4   0   3   0]
 [ 14  18 665   5 372   0 232   0  14   0]
 [122  48  12 808 104   2  93   0  68   7]
 [  8  16 242  45 483   0 222   0   6   0]
 [  0   0   0   0   0 680   0   5   1   0]
 [ 83   0  50  35  25   2 167   0  20   0]
 [  1   0   0   0   0 232   1 893  55 113]
 [ 24   2  23   4  11   9  63   0 832   4]
 [  0   0   0   0   0  75   0 102   1 876]]
```

A 10×10 table of counts is much easier to read when we pull out its largest mistakes. Below we print the six largest off-diagonal entries, formatted as true class to predicted class, along with each class's individual accuracy (its diagonal entry divided by its column sum).

```
names = data.text_labels(range(10))
off = conf - np.diag(np.diag(conf))      # errors only
order = np.argsort(off, axis=None, kind='stable')[::-1][:6]  # largest first
pairs = [f'{names[t]}->{names[q]} {off[q, t]}'  # true -> predicted
         for q, t in zip(*np.unravel_index(order, off.shape))]
print(', '.join(pairs[:3]))
print(', '.join(pairs[3:]))
print('per-class accuracy:', np.round(np.diag(conf) / conf.sum(axis=0), 2))
```

```
coat->pullover 372, pullover->coat 242, sandal->sneaker 232
shirt->pullover 232, shirt->coat 222, shirt->T-shirt 218
per-class accuracy: [0.75 0.91 0.66 0.81 0.48 0.68 0.17 0.89 0.83 0.88]
```

The mistakes are heavily concentrated. Coats are misclassified as pullovers 372 times and pullovers as coats 242 times; shirts are mistaken for pullovers, coats, and t-shirts more than 200 times each; and sandals are mistaken for sneakers 232 times. Trousers (91%) and sneakers (89%) are easy, whereas coats (48%) and shirts (17%) are much harder. These are the exact same two confusion groups that softmax regression struggled with in Section 3.3—the four upper-body garments and the three types of footwear—but the counted model fares far worse on the upper-body group: softmax regression recognized shirts 40% to 60% of the time, while naive Bayes manages only about one in six.

Is 71% simply the ceiling for a linear classifier on binary pixels? Softmax regression defines hyperplanes of the exact same form, but chooses its weights by minimizing the cross-entropy of $p(y \mid \mathbf{x})$. Let us train softmax regression on these same binary pixels with mini-batch stochastic gradient descent so that only the fitting procedure differs. The update uses the result from Section 3.1 that the gradient of the cross-entropy with respect to the logits is simply the predicted probability vector minus the one-hot label.

```
rng = np.random.default_rng()
W_sr, b_sr = np.zeros((10, 784)), np.zeros(10)
for epoch in range(20):                  # minibatches of 256 images
    for idx in np.array_split(rng.permutation(len(X)), len(X) // 256):
        O = X[idx] @ W_sr.T + b_sr                    # logits
        E = np.exp(O - O.max(axis=1, keepdims=True))
        G = E / E.sum(axis=1, keepdims=True) - np.eye(10)[Y[idx]]
        W_sr -= 0.05 * G.T @ X[idx] / len(idx)        # mean-loss gradient
        b_sr -= 0.05 * G.mean(axis=0)
pred_sr = (X_test @ W_sr.T + b_sr).argmax(axis=1)
conf_sr = np.zeros((10, 10), dtype=int)
np.add.at(conf_sr, (pred_sr, Y_test), 1)
print(f'accuracy: softmax regression {(pred_sr == Y_test).mean():.3f}, '
      f'naive Bayes {(pred == Y_test).mean():.3f}')
print('pullover<->coat errors: softmax regression',
      conf_sr[2, 4] + conf_sr[4, 2], 'naive Bayes', conf[2, 4] + conf[4, 2])
```

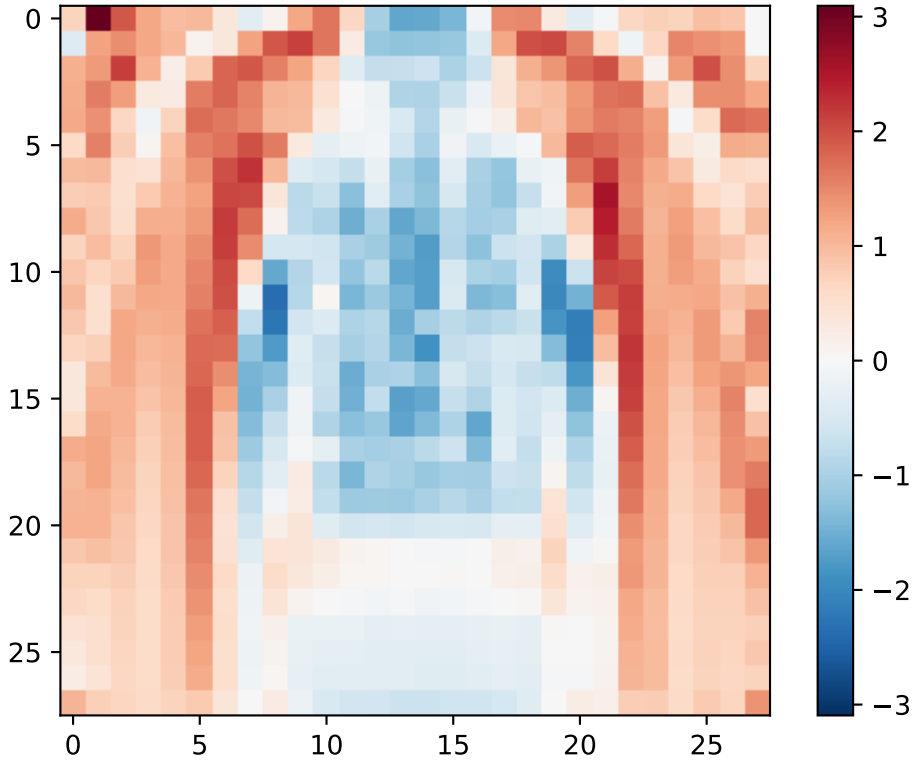
```
accuracy: softmax regression 0.830, naive Bayes 0.706
pullover<->coat errors: softmax regression 312 naive Bayes 614
```

Softmax regression reaches about 83% accuracy, twelve percentage points ahead of naive Bayes, and roughly halves the errors between pullovers and coats. The linear model family itself is therefore not the bottleneck; the gap comes entirely from how counting sets the weights. We will pinpoint why counting distorts those weights when we inspect calibration next.

First, looking directly at the naive Bayes weights shows what features the classifier relies on to separate two classes. The score difference between pullover and coat is an affine function with weight vector $\mathbf{w}_{\text{pullover}} - \mathbf{w}_{\text{coat}}$: whenever pixel i is inked rather than blank, it shifts the score toward pullover by entry i of this vector. Let us plot this difference vector as a 28×28 image, with red favoring pullover and blue favoring coat, and print its largest entry along with the underlying counts.

```
w = (W[2] - W[4]).reshape(28, 28)        # pullover minus coat, in nats
m = np.abs(w).max()
d2l.plt.imshow(w, cmap='RdBu_r', vmin=-m, vmax=m)
d2l.plt.colorbar()
k = np.abs(w).argmax()
print(f'largest |weight| {m:.1f} nats at row {k // 28}, column {k % 28}: '
      f'inked in {n_x[2, k]:.0f} pullovers and {n_x[4, k]:.0f} coats')
```

```
largest |weight| 3.1 nats at row 0, column 1: inked in 21 pullovers and 0 coats
```



Red pixels run along both sides of the image and across the top rows, where pullovers are inked more often than coats. Blue pixels fill the torso with weights near -1, though it would be a mistake to read those blue pixels as strong evidence for a coat when inked. Remember that each weight measures the score of an inked pixel *relative* to a blank one. A torso pixel is inked in roughly 99% of pullovers and more than 99.5% of coats, so seeing it inked barely moves the needle—about -0.01 nats. The weight of -1 is actually what the model adds in favor of pullover on the rare occasion when a torso pixel is left blank. In practice, the decision is made along the sides. Meanwhile, the single largest weight, about 3 nats, sits in a corner that is inked in roughly twenty pullovers and zero coats, meaning its magnitude is controlled largely by the Laplace pseudo-count rather than by abundant data.

Before we critique the generative fit further, notice one genuine advantage it holds over a discriminative model. If some pixels are missing at test time, say hidden behind an occlusion, naive Bayes simply drops their factors from the product. Under the generative model this is mathematically exact, because marginalizing $p(x_i \mid y)$ over both values of an unobserved x_i sums to 1. Softmax regression has no corresponding marginalization rule and must impute values for the missing pixels (exercise 9).

36.11.4.3 Calibration

The class scores do more than rank the ten classes. Exponentiating them recovers the Bayes numerators, and normalizing recovers the denominator $p(\mathbf{x})$ we dropped earlier, so the model's posterior probabilities are simply the softmax of its scores:

$$p(y \mid \mathbf{x}) = \frac{\exp s_y(\mathbf{x})}{\sum_{y'} \exp s_{y'}(\mathbf{x})}. \tag{36.142}$$

Every prediction therefore comes with a stated confidence, namely the largest of these posterior probabilities. As in Section 3.4, a classifier is *calibrated* when its stated confidences match empirical frequencies: among all predictions made with confidence c, a fraction c should actually be correct. To test calibration, we group the test images into confidence bins and compare the average claimed confidence with the actual accuracy in each bin.

The *expected calibration error* (ECE) condenses this table into a single number: the weighted average of the gap between claimed confidence and achieved accuracy across bins, weighted by the fraction of test images in each bin. Below we compute the softmax in its numerically stable form by subtracting the largest score first (Section 35.8.2), and we also print the median gap between the highest and second-highest class scores.

```
post = np.exp(s - s.max(axis=1, keepdims=True))      # softmax, stably
post /= post.sum(axis=1, keepdims=True)
c = post.max(axis=1)                                 # claimed confidence
print(f'mean confidence {c.mean():.3f}, accuracy {correct.mean():.3f}')
edges, ece = [0, 0.9, 0.99, 0.999, 1], 0
for lo, hi in zip(edges[:-1], edges[1:]):
    m = (c > lo) & (c <= hi)
    ece += m.mean() * abs(c[m].mean() - correct[m].mean())
    print(f'({lo}, {hi}]: {m.sum():5d} images, claimed {c[m].mean():.3f}, '
          f'achieved {correct[m].mean():.3f}')
top2 = np.sort(s, axis=1)[:, -2:]
print(f'ECE {ece:.2f}, median top-two score gap '
      f'{np.median(top2[:, 1] - top2[:, 0]):.0f} nats')
```

```
mean confidence 0.992, accuracy 0.706
(0, 0.9]:   257 images, claimed 0.726, achieved 0.350
(0.9, 0.99]:   297 images, claimed 0.961, achieved 0.384
(0.99, 0.999]:   265 images, claimed 0.996, achieved 0.419
(0.999, 1]:  9181 images, claimed 1.000, achieved 0.735
ECE 0.29, median top-two score gap 61 nats
```

The result is striking: on average, the model claims 99% confidence while getting only 71% of its predictions right. Roughly 92% of all test images land in the top bin, where the model claims certainty to three decimal places yet is right only about 73% of the time. In the three lower bins, it claims between 73% and 99.6% confidence while achieving only 35% to 42% accuracy. Because every bin is overconfident, the expected calibration error equals the gap between mean confidence and accuracy, about 0.29—compared with under 0.03 for softmax regression in Section 3.4. The ranking of confidences still carries useful signal, since accuracy rises from bin to bin, but the stated probabilities cannot be taken at face value.

The culprit is the same phenomenon we saw at the pullover hem: correlated pixels are counted as independent pieces of evidence. The simplest version of this failure is a single feature duplicated several times, and the arithmetic takes three lines. Consider two classes with priors π_0 and π_1 and a feature x with likelihood ratio $r(x) = p(x \mid y = 1)/p(x \mid y = 0)$. Suppose our dataset contains k identical copies of x. Because all k copies always agree, they carry no more information than a single copy, so the true log posterior odds are

$\log(\pi_1/\pi_0)+\log r(x)$. Naive Bayes, however, treats the copies as conditionally independent, multiplies k identical factors $p(x \mid y)$, and reports

$$\log \frac{\pi_1}{\pi_0} + k \log r(x). \tag{36.143}$$

Duplicated evidence multiplies the log-odds by k. When the priors are equal, multiplying the log-odds by a positive constant never flips its sign, so naive Bayes predicts the exact same class for every k, yet it claims posterior odds of $\bar{r}^k$ where $\bar{r}$ is right, with $\bar{r} = \max(r, 1/r)$. This is the second result in this section worth remembering: the ranking survives, while the confidence collapses.

A concrete number makes the inflation vivid. Take a clue that is right 80% of the time, with $p(x = 1 \mid y = 1) = 0.8$ and $p(x = 1 \mid y = 0) = 0.2$, so that $r = 4$ when the clue is active. A single copy gives the right confidence of 0.80. Storing two, three, four, and five copies pushes naive Bayes's claimed confidence to 0.94, 0.985, 0.996, and 0.999, even though its predictions are still right only 80% of the time. Duplicating the word "free" in our two-word spam filter does the same thing: the odds of the first message jump from 9 to $\frac{2}{3} \cdot 4.5^2 \cdot 3 = 40.5$, raising its stated spam probability to 0.976.

Be careful not to overstate how safe the ranking is: the argument above relies on equal priors, and that caveat matters. With prior odds $\pi_1/\pi_0 = 1/5$ and $r = 4$, a single copy gives posterior odds of $4/5$ and correctly predicts class 0; two copies give $16/5$ and flip the prediction to class 1. Repeated evidence can overwhelm a prior that a single observation should not override. The ranking stays intact only when the evidence already agrees with the prior or when one copy alone outweighs it.

On Fashion-MNIST, the 784 pixels are not literal copies of one another, but large patches of them are tightly correlated. Our two hem pixels behaved almost like one feature counted twice, and dozens of such correlated regions pile up to create a median gap of about 60 nats between the highest and second-highest class scores. Since $e^{-60} \approx 10^{-26}$, the softmax turns a 60-nat gap into a reported confidence of 1.000. That naive Bayes still ranks the classes usefully despite extreme overconfidence is a classic property of the model (Domingos and Pazzani 1997). Even so, the ranking is not completely unscathed: correlated patches vary in size, so a broad region of redundant pixels can easily outvote a smaller region carrying independent information. This imbalance is a plausible explanation for the twelve percentage points lost to softmax regression, which can down-weight pixels that merely echo their neighbors, though our experiment alone does not isolate that mechanism.

Overconfidence has a remarkably simple post-hoc repair, which we only name here. Temperature scaling (Section 3.4) divides every class score by a single scalar T fitted on held-out validation data, leaving the class rankings completely unchanged; for k identical copies and equal priors, setting $T = k$ recovers the exact posterior. Exercise 8 fits T for our counted model. Nor is overconfidence unique to naive Bayes: deep neural networks can suffer from the same bias, though typically to a lesser degree and depending on their architecture and training recipe (C. Guo et al. 2017), so a reliability check is good practice for any classifier whose probabilities will drive decisions.

One natural question remains open. With 6,000 images per class, the discriminative fit won handily. With very few training examples, however, the ranking often reverses: Ng and Jordan (2002) proved that naive Bayes parameters converge after a number of examples logarithmic in the dimension d, whereas logistic regression can require a number linear in d. Exercise 10 explores this crossover on pullovers and coats.

36.11.5 Summary

Bayes' rule turns a generative model of how each class produces its inputs into a classifier, but estimating an unrestricted table of $p(\mathbf{x} \mid y)$ over 10^{236} binary images is impossible. Assuming conditional independence among features given the class reduces the model to a single frequency per class and feature, smoothed by one Laplace pseudo-count, so training reduces to counting. Because the log of the naive Bayes numerator is affine in binary features, Bernoulli naive Bayes is a linear classifier in the exact same model family as softmax regression.

On Fashion-MNIST, this counted classifier reaches about $71\% \pm 0.9$ percentage points, twelve points below softmax regression fitted directly on the same binary pixels. It reports an average confidence of 99%, because correlated pixels are treated as independent evidence and duplicated evidence multiplies the log-odds. That extreme overconfidence shows why calibration should be checked whenever a classifier's probabilities are used in practice.

36.11.6 Exercises

1. **XOR and the naive assumption.** The four points $(0,0)$, $(0,1)$, $(1,0)$ and $(1,1)$ carry the labels $y = x_1 \oplus x_2$, that is 0, 1, 1 and 0. Each point occurs once in the training set.
   1. Compute the maximum-likelihood estimates of $p(y)$ and $\theta_{iy} = p(x_i = 1 \mid y)$, and the class scores of all four points. Which predictions does the naive Bayes rule Equation 36.137 make?
   2. Show that no choice of Bernoulli naive Bayes parameters classifies all four points correctly, using the affine form Equation 36.141 of the scores. Why can degenerate parameters $\theta_{iy} \in \{0, 1\}$ not help? (Each class must give positive probability to its own two points.)
   3. Replace the two factors $p(x_1 \mid y)\, p(x_2 \mid y)$ by a joint table $p(x_1, x_2 \mid y)$. In the language of Bayesian networks (Koller and Friedman 2009), this adds the edge between x_1 and x_2 that the right panel of Figure 36.50 shows. Estimate this model from the four points and show that it classifies them correctly. Why is its class score no longer affine in $\mathbf{x}$?
   4. Within one class, let x_1 and x_2 be independent fair bits and let $x_3 = x_1 \oplus x_2$. Show that every pair of the three features is independent. Compare the probability of each of the eight patterns with the product of the three marginals. Why can a check of all pairwise correlations within a class not confirm the assumption Equation 36.136?
2. **Naive Bayes as logistic regression.** Consider two classes $y \in \{0, 1\}$ with the prior $\pi_1 = p(y = 1)$ and the Bernoulli parameters $\theta_{iy} = p(x_i = 1 \mid y)$.
   1. Show that the naive Bayes posterior is $p(y = 1 \mid \mathbf{x}) = \text{sigmoid}(\mathbf{w}^\top \mathbf{x} + b)$ with $\text{sigmoid}(z) = 1/(1 + e^{-z})$ and $w_i = \log \frac{\theta_{i1}(1-\theta_{i0})}{\theta_{i0}(1-\theta_{i1})}$. Express b through the θ_{iy} and π_1. Which sign does w_i have for a pixel that is inked more often in class 1 than in class 0?
   2. Show that, conversely, every pair $(\mathbf{w}, b)$ arises from some choice of $\theta_{iy} \in (0, 1)$ and $\pi_1 \in (0, 1)$. Naive Bayes and logistic regression therefore define the same set of posteriors and differ in how they fit it.
   3. Suppose the two classes of the training set are linearly separable. How does the conditional likelihood of logistic regression change when a separating $(\mathbf{w}, b)$ is

multiplied by a factor $t \to \infty$? What does this imply for the maximum-likelihood estimate of $(\mathbf{w}, b)$? Which $(\mathbf{w}, b)$ does naive Bayes with Laplace smoothing return on the same data?

3. **Gaussian class-conditionals.** A single feature has $x \mid y = 0 \sim \mathcal{N}(0, 1)$ and $x \mid y = 1 \sim \mathcal{N}(0, 4)$, and both classes have prior $1/2$.
   1. Find the set of x that the Bayes classifier assigns to class 1. Why can no threshold on x produce this set? Which transformed feature turns the boundary into a threshold?
   2. In d dimensions, Gaussian naive Bayes with class-specific variances has the class score
   $$\log p(y) - \sum_{i=1}^{d} \left[\frac{(x_i - \mu_{iy})^2}{2\sigma_{iy}^2} + \frac{1}{2} \log \left(2\pi\sigma_{iy}^2 \right) \right]. \tag{36.144}$$
   Show that the boundary between two classes is a quadratic surface without cross terms $x_i x_j$.
   3. Two classes in two dimensions have equal priors and the class-conditionals $\mathcal{N}(\mathbf{0}, \Sigma_+)$ and $\mathcal{N}(\mathbf{0}, \Sigma_-)$ with $\Sigma_\pm = \left(\begin{smallmatrix} 1 & \pm\rho \\ \pm\rho & 1 \end{smallmatrix} \right)$ for some $0 < \rho < 1$. Show that the difference of the two class scores is $2\rho x_1 x_2/(1 - \rho^2)$, so that the Bayes classifier assigns class $+$ exactly when $x_1 x_2 > 0$, and that each single score also depends on $x_1^2 + x_2^2$. The two classes have the same marginals, so Gaussian naive Bayes is at chance even with unlimited data. Under $\mathcal{N}(\mathbf{0}, \Sigma_+)$, $P(x_1 x_2 > 0) = \frac{1}{2} + \frac{1}{\pi} \arcsin \rho$ (Sheppard's formula). Find the Bayes error as a function of ρ, and evaluate it for $\rho = 0.1$ and $\rho = 0.9$.
4. **Two event models for text.** A vocabulary V has the four words *goal*, *match*, *vote* and *poll*. The training set contains three sports documents, "goal goal match", "goal match match" and "match vote", and three politics documents, "vote poll poll", "vote vote match" and "poll vote". The multinomial model treats a document as a sequence of independent word draws from a class-specific distribution over V. The Bernoulli model records for each word of V only whether it occurs in the document.
   1. Estimate both models with Laplace smoothing. For the multinomial model, add 1 to each word count of a class and $|V|$ to the class's total count. For the Bernoulli model, estimate the probability that a word occurs by $(n_{iy} + 1)/(n_y + 2)$, where n_{iy} counts the class-y documents that contain word i.
   2. Classify the document "goal match poll poll" with both models. Which words drive each decision?
   3. Concatenate a document with itself k times. How do the log posterior odds of the two models change? In real text, a word that has occurred once in a document tends to occur again (Church and Gale 1995), a finding that exercise 3 of Section 36.3 reproduces. Which assumption of the multinomial model does this violate?
5. code **Paired comparison.** The text compared naive Bayes with softmax regression by their accuracies, `pred` against `pred_sr`.
   1. Bootstrap the per-image difference in correctness, using the same resampled indices for both models, and report a 95% interval for the difference of the accuracies.
   2. Compare the width of this interval with that of the two marginal intervals and with the unpaired standard error $\sqrt{p_1(1 - p_1)/n + p_2(1 - p_2)/n}$. Compute the correlation of the two vectors of correctness, and use it to explain why pairing helps.

3. Count the test images that both models get wrong, that only naive Bayes gets right, and that only softmax regression gets right. Which of these counts determine the difference of the accuracies? What would the table look like for a model that differs from naive Bayes only by correcting some of its errors?

6. **Duplicated and correlated evidence.** Two classes have equal priors, and a binary feature has $p(x = 1 \mid y = 1) = 0.8$ and $p(x = 1 \mid y = 0) = 0.2$.
   1. A pipeline stores the feature k times. Find the smallest k for which naive Bayes claims at least 99.9% confidence for predictions that are right 80% of the time.
   2. Now store two copies that agree only partly: given y, the second copy equals the first with probability q and is otherwise a fresh draw from the same class-conditional distribution. Show that both copies have the same marginal as x, so naive Bayes still claims $16/17 \approx 0.94$ when both copies are 1. Show that the correct posterior is $r/(1 + r)$ with $r = 4(0.8 + 0.2q)/(0.2 + 0.8q)$, and evaluate it for $q \in \{1, 0.75, 0.5, 0.25, 0\}$. How does the over-count depend on the correlation of the copies?
   3. Show that dividing every class score by $T = k$ before the softmax restores the correct posterior in (a). Why are equal priors needed for this? (The prior term is divided by k too.)

7. code **Label shift.** Under label shift (Section 3.6), the target prior $p_T(y)$ differs from the source prior $p_S(y)$, while $p(\mathbf{x} \mid y)$ stays the same.
   1. Show that the posterior under the target prior is $p_T(y \mid \mathbf{x}) \propto p_S(y \mid \mathbf{x})\, p_T(y)/p_S(y)$. Why does naive Bayes need no retraining to use it?
   2. Build a test set in which pullovers make up 90% of the images: all 1,000 test pullovers and 111 images drawn at random from the other classes. Compare the accuracy of `pred` on it with that of the corrected rule of (a), with $p_T(\text{pullover}) = 0.9$ and $p_T(y) = 1/90$ for every other class.
   3. The correction adds $\log 9 \approx 2.2$ nats to the pullover score and subtracts 2.2 from the others. Compare this with the score gaps of the `#mdl-naive-bayes-calibration-3` cell and explain the size of the change you measured in (b).

8. code **Temperature scaling.** ● Exercise 6 shows that dividing the class scores by $T = k$ undoes a k-fold duplication of features. For Fashion-MNIST the effective duplication is unknown, and temperature scaling (C. Guo et al. 2017) fits T on held-out data instead: the posterior becomes the softmax of $s_y(\mathbf{x})/T$.
   1. Refit the model on the first $50{,}000$ training images, and choose T by minimizing the average negative log-likelihood of the posterior on the remaining $10{,}000$. Why does T not change the test accuracy?
   2. Apply the fitted T to the scores of the same $50{,}000$-image model on the test set, and repeat the reliability table of the `#mdl-naive-bayes-calibration-3` cell with the bin edges 0, 0.5, 0.7, 0.9, 0.99 and 1. Compare the mean claimed confidence with the accuracy.
   3. In the setting of exercise 6, show that the standard deviation of the score gap on real data is $\sqrt{k}$ times the standard deviation that the model predicts. Compute the square of this ratio for the pairs pullover/coat, t-shirt/shirt, and pullover/shirt, and compare it with the fitted T.
   4. Apply the prior correction of exercise 7 to the temperature-scaled posterior, on the test set built there. Why does the corrected prior change many more predictions after temperature scaling?

9. code **Missing pixels.** Hide a random half of the pixels of every test image with a Boolean mask `M` of the shape of `X_test` that is `True` for the visible pixels.
   1. Show that under the model, marginalizing a hidden pixel out of $p(\mathbf{x} \mid y)$ removes

its factor, so that the score of the visible pixels is $\log p(y) + \sum_{i \text{ visible}} \log p(x_i \mid y)$. Implement it with two matrix products, `(X_test * M) @ np.log(P_xy).T + ((1 - X_test) * M) @ np.log(1 - P_xy).T` plus the log prior, and compare the accuracy with the 71% of the full images.

2. Instead fill the hidden pixels with 0, with 1, or with the mean of the training images, and score with the affine form. Explain each accuracy from Equation 36.141; what does filling with 0 add to the score of every class?
3. Score softmax regression from the text on the images with hidden pixels set to 0. Which operation does a discriminative model lack that the generative model used in (a)?

10. [extended] **Generative versus discriminative learning.** ●● Ng and Jordan (2002) compare naive Bayes with logistic regression, which by exercise 2 fits the same family of posteriors by maximizing the conditional likelihood. They prove that the naive Bayes parameters approach their population values after a number of examples logarithmic in d. For logistic regression, the number of examples needed to approach its asymptotic error is at most linear in d, and in the worst case it is also at least linear in d. On many of their 15 datasets, naive Bayes has the lower test error for small training sets and logistic regression for large ones.
    1. Let each class have at least m training examples. Use Hoeffding's inequality (Section 36.9) and a union bound to show that all $2d$ unsmoothed estimates $\widehat{\theta}_{iy}$ of a two-class model lie within ε of their population values with probability at least $1 - \delta$ when
    $$m \geq \frac{\log(4d/\delta)}{2\varepsilon^2}. \tag{36.145}$$
    Evaluate the bound for $d = 784$, $\varepsilon = 0.05$ and $\delta = 0.05$.
    2. A uniform bound on $|\widehat{\theta}_{iy} - \theta_{iy}|$ does not by itself bound the change of the class scores. Why not, for the pixels with θ_{iy} close to 0 or 1, such as the corner pixel that carries the largest pullover-coat weight in the text? What must be assumed about the θ_{iy} for the scores to converge at the rate of the parameters?
    3. Restrict the data to pullovers and coats. For the total training-set sizes $n \in \{10, 20, 50, 100, 200, 500, 1000, 2000, 5000\}$ and for all $12{,}000$ images, draw random training subsets that contain both classes, 20 for each size up to $n = 200$ and 5 for each larger size. Fit naive Bayes with Laplace smoothing, and logistic regression without weight decay. For small n the training data are linearly separable, and by exercise 2c the logistic loss has no minimizer, so run a fixed number of full-batch gradient steps from zero initialization. Plot the average test errors of both classifiers against n on a logarithmic axis with `d2l.plot`. In which range of n, if any, does naive Bayes have the lower error?
    4. Repeat (c) with synthetic data for which the naive assumption holds: draw balanced labels and sample each pixel independently from Bernoulli(θ_{iy}), with the θ_{iy} estimated from all training images of the two classes. How do the two learning curves and their asymptotic errors change?
    5. Theorem 4 of Ng and Jordan (2002) bounds the excess error of naive Bayes by the probability that the score gap of the population model falls near zero. Compare this probability for real pixels and for the synthetic data of (d), and explain how this comparison, together with the asymptotic errors of (c) and (d), accounts for the two pairs of learning curves.

36.12 Further Reading

The list at the end of this section collects books and courses that accompany what this chapter teaches. First we turn to what the chapter leaves out. Following the order of the chapter, each short entry begins from a limitation of a method we covered and explains how the literature moves past it. Two of these ideas, conformal prediction and confidence sequences, are immediately useful in practice, repairing the intervals and tests of Section 36.5 under the messy conditions of real experiments. Others, such as the VC dimension, are chiefly of theoretical interest, and some, like whether gradient noise is truly heavy-tailed, remain actively debated. Nothing later in the book depends on this section, so treat it as a map for exploring further.

36.12.1 Probability Models

We start with three extensions to our treatment of distributions: a subtle class of distributions that slips between discrete and continuous, the limiting laws for averages whose variance is infinite, and a classic quirk of the softmax when options look alike.

36.12.1.1 Distributions Without Atoms or Densities

In Section 36.1, we noted that a random variable with a density has no point masses (atoms), and we treated every distribution as discrete, continuous with a density, or a mixture of the two. Surprisingly, the converse does not hold: having no atoms is not enough to guarantee a density. To see why, build a random number in $[0, 1]$ in base 3 using only the digits 0 and 2, choosing each digit independently with a fair coin. Its cumulative distribution function is continuous everywhere, so no single point carries positive probability, yet all of its mass sits on the Cantor set, whose total length is zero. No density function can integrate to 1 over a set of length zero.

The *Lebesgue decomposition* completes this classification: every probability distribution on the real line is a mixture of a discrete part, a part with a density, and a *singular continuous* part like the Cantor distribution. Having a density corresponds precisely to being *absolutely continuous* with respect to length, in which case the Radon–Nikodym theorem supplies the density. Billingsley (1995) develops probability on this measure-theoretic footing, where expectation becomes a single integral against the distribution and the sums and integrals of our chapter appear simply as two special cases.

36.12.1.2 Heavy Tails and Stable Laws

As exercise 5 shows, the average of independent Cauchy draws has the exact same Cauchy distribution, and averages of variables with power-law tails settle down far more slowly than the familiar $1/\sqrt{n}$ rate. If they do not approach a Gaussian, what do they converge to? When the tail $P(|X| > x)$ decays like $x^{-\alpha}$ with $0 < \alpha < 2$, the variance is infinite, and properly centered sums scaled by $n^{1/\alpha}$ rather than $\sqrt{n}$ converge to an α-*stable* law. The Cauchy distribution is the symmetric stable law with $\alpha = 1$, while the Gaussian sits at the boundary $\alpha = 2$. Gnedenko and Kolmogorov (1954) develop the classical theory of these limit laws.

In deep learning, heavy tails have attracted attention both in minibatch gradient noise, which Simsekli et al. (2019) model as α-stable, and in the eigenvalue spectra of trained

weight matrices (Martin and Mahoney 2021). Keep a grain of salt handy, though: every empirical claim of this kind relies on estimating the tail index α, and those estimates can shift dramatically depending on how far out in the tail one looks.

36.12.1.3 Choice Models and Irrelevant Alternatives

In Section 36.4.4, we derived the softmax by adding independent Gumbel noise to each option's score. A consequence of that independence is that the odds ratio between any two choices, $p_i/p_j = e^{z_i - z_j}$, ignores every other option on the menu. That property breaks down as soon as two options are near-duplicates. Imagine a commuter who splits evenly between driving a car and taking a red bus. If the city adds an identical blue bus, the commuter should still drive half the time and split the remaining half between the two buses. Under a softmax, however, all three options receive probability $\frac{1}{3}$.

Allowing the noise of similar options to be correlated (as in a nested logit model, or a probit model with correlated Gaussian noise) keeps the car's probability near $\frac{1}{2}$, at the cost of losing closed-form choice probabilities. McFadden (1974) derived the logit from Gumbel noise and highlighted this restriction, known as Luce's *independence of irrelevant alternatives*, as its central modeling limitation.

36.12.2 Estimation and Testing

The confidence intervals and hypothesis tests of Section 36.5 leaned on three conveniences: a Gaussian approximation, a known model for the errors, or a sample size fixed before seeing the data. The three topics below each show what happens when we give up one of those crutches.

36.12.2.1 Better Bootstrap Intervals

The simple percentile bootstrap of Section 36.5 inherits any bias or skewness in the resampled estimates, so on small samples or skewed statistics its actual coverage can fall noticeably short of its nominal level. Two refinements fix most of the gap. The *studentized* bootstrap resamples the self-normalized pivot $(\hat{\theta}^* - \hat{\theta})/\widehat{\mathrm{se}}^*$ rather than $\hat{\theta}^*$ itself, while the *BCa* (bias-corrected and accelerated) interval shifts the percentile cutoffs to account for both bias and the way the standard error varies with θ. For one-sided intervals, both refinements shrink the coverage error from order $1/\sqrt{n}$ down to order $1/n$ (Hall 1992). You rarely need them when reporting an accuracy on thousands of test examples, but they pay off immediately on small evaluation sets or ratios of error rates. DiCiccio and Efron (1996) give a clear review of these methods.

36.12.2.2 Conformal Prediction

A trained neural network hands us a point prediction, but it gives no prediction interval with guaranteed coverage unless we are willing to assume a parametric error model. *Split conformal prediction* sidesteps modeling assumptions altogether. Using n held-out calibration points, we score how badly the model misses each one, for instance by the residual $|y - f(\mathbf{x})|$. For a new input, we then return every label whose score is at most the $\lceil (n+1)(1-\alpha) \rceil$-th smallest calibration score.

As long as the calibration and test points are exchangeable, the resulting prediction set contains the true label with probability at least $1 - \alpha$, for *any* model and *any* data distribution.

The proof uses the same symmetry behind the permutation test: under exchangeability, the rank of the test score among all $n+1$ scores is uniform. One caveat matters in practice: the guarantee holds on average over inputs, not conditionally for each individual input. Vovk et al. (2022) develop the full theory, and Angelopoulos and Bates (2023) offer an accessible tutorial with code for both classification and regression.

36.12.2.3 Confidence Sequences

The z-test of Section 36.5 assumes that you chose the sample size n before looking at a single outcome. If instead you peek at a running experiment after every batch and stop the moment $p < 0.05$, a false alarm becomes virtually certain if you watch long enough. Exercise 10 builds a Gaussian mixture martingale test that stays valid no matter how often you check. *Confidence sequences* push that idea to full intervals that hold simultaneously at every sample size n, for any bounded or sub-Gaussian data rather than a single parametric model. You pay for this freedom with wider intervals, though for the sharpest boundaries the extra width grows remarkably slowly, only like $\sqrt{\log \log n}$. Howard et al. (2021) construct these sequences from nonnegative supermartingales, and Johari et al. (2017) describe how always-valid p-values were deployed in a commercial A/B testing platform.

36.12.3 Computing with Posteriors

In Section 36.8, we worked with a single Markov chain, the random-walk Metropolis algorithm, on a two-parameter posterior. Real Bayesian models have thousands or millions of parameters, which calls for sharper samplers and scalable approximations.

36.12.3.1 Gibbs Sampling and Hamiltonian Monte Carlo

Random-walk Metropolis (Section 36.8.3) requires tuning a proposal scale in every direction and inches forward by small diffusive steps. When a posterior is high-dimensional or tightly correlated, the chain wastes most of its time wandering back and forth. *Gibbs sampling* avoids proposal tuning by updating one coordinate at a time from its exact conditional distribution, so every move is accepted; the catch is that those conditionals must be easy to sample, and strong correlations between coordinates still slow it down (Geman and Geman 1984).

Hamiltonian Monte Carlo takes a more physical approach: it treats $-\log \tilde{p}$ as a potential energy, injects a random momentum, and uses gradients to simulate the resulting trajectory for many steps. Because the dynamics nearly conserve energy, a single proposal can glide far across the posterior and still be accepted with high probability. Neal (2011) gives a lucid introduction to the method and its tuning, and Hoffman and Gelman (2014) automate the trajectory length with the No-U-Turn Sampler, which became the default engine behind Stan.

36.12.3.2 Posteriors of Neural Networks

Every algorithm in Section 36.8 evaluates the full log-joint across the entire dataset at every step, and the Laplace approximation requires storing and inverting a $d \times d$ Hessian. Neither is feasible for a neural network with millions of parameters and millions of training examples. *Stochastic-gradient Langevin dynamics* adapts the unadjusted Langevin update of exercise 9 by plugging in minibatch gradients and a shrinking step size, trading away the Metropolis acceptance check for a per-step cost identical to ordinary SGD (Welling and Teh

2011). Laplace approximations scale up in a different way, by keeping only the most informative slice of the curvature: a Kronecker-factored block per layer, or the Hessian of the last layer alone. Daxberger et al. (2021) show that these lightweight approximations often match far costlier Bayesian methods at a tiny fraction of the compute.

36.12.4 Concentration and Learning Theory

The first two topics below generalize the tools of Section 36.9 and Section 36.10 beyond simple averages and Rademacher complexity. The last two take up the puzzle on which our theory sections closed: how to prove guarantees for the specific solution an algorithm actually finds, rather than for the worst function lurking in its hypothesis class.

36.12.4.1 Bounded Differences and Lipschitz Functions

The Chernoff method in Section 36.9 relies on having a sum of independent terms. Many quantities we care about, such as the largest gap between true risk and empirical risk across a whole model class, depend on independent inputs without being simple sums. *McDiarmid's inequality* generalizes Hoeffding's bound to such functions: replace "each term lies in an interval of length c_i" with "changing input i alone moves the function by at most c_i," and the very same tail bound $e^{-2t^2/\sum_i c_i^2}$ still holds, proved by replacing the product of moment generating functions with a martingale.

Gaussian vectors enjoy an even cleaner property: for a standard Gaussian vector in any dimension d, every L-Lipschitz function f obeys $P(|f - \mathbb{E}f| \geq t) \leq 2e^{-t^2/(2L^2)}$. Because the Euclidean norm is 1-Lipschitz, its fluctuations around the thin shell stay order 1 no matter how large d grows. Boucheron et al. (2013) give a comprehensive treatment of both results and the machinery behind them.

36.12.4.2 VC Dimension and Covering Numbers

In Section 36.10, we controlled infinite hypothesis classes through their Rademacher complexity. The classical alternative counts how many distinct labelings a class can produce on a finite set of points. Suppose a binary classifier family cannot realize all 2^{d+1} labelings of any $d+1$ points (for example, linear halfspaces in $\mathbb{R}^m$ have $d = m+1$). Sauer's lemma then guarantees that the class produces at most $(en/d)^d$ labelings on any $n \geq d$ points, and a union bound over those labelings yields a generalization gap of order $\sqrt{d \log n / n}$, the celebrated argument of Vapnik and Chervonenkis (1971) . For real-valued function classes, one instead covers the class at each resolution ϵ by finitely many representatives; a technique called *chaining* sums across scales and shaves off the extra $\log n$ factor (Vershynin 2018).

36.12.4.3 PAC-Bayes Bounds

Every bound in Section 36.10 guards against the worst function in a hypothesis class fixed before seeing the data. A *PAC-Bayes* bound shifts perspective: it evaluates a posterior distribution Q over functions chosen *after* observing the data, measured against a prior P fixed beforehand. The class complexity term is then replaced by $D_{\mathrm{KL}}(Q\|P)/n$ under a square root (McAllester 1999). Even if the full hypothesis class is enormous, a posterior that stays close to the prior pays only a small complexity penalty. Building on this idea, Dziugaite and Roy (2017) optimized a Gaussian posterior around the weights found by SGD and produced nonvacuous generalization bounds on a binary MNIST task for stochastic networks with far more parameters than training examples.

36.12.4.4 Algorithmic Stability

The random-label experiment (Section 4.5.1.1) makes clear that the class of functions a neural network *can* represent is vastly larger than the subset SGD actually visits on real data. *Algorithmic stability* bounds the learning algorithm rather than the hypothesis class. If swapping out any single training example shifts the loss of the learned model on any test point by at most β, then the expected generalization gap is also of order β, and McDiarmid's inequality lifts that expectation into a high-probability bound (Bousquet and Elisseeff 2002). For smooth, Lipschitz convex losses, SGD is stable with β proportional to the sum of its step sizes divided by n, giving tighter guarantees when training takes fewer or smaller steps (Hardt et al. 2016).

36.12.5 Classifiers from Counts

We close with two continuations of Section 36.11: repairing miscalibrated probabilities when a single temperature is not enough, and using a generative classifier to learn from unlabeled data, something a purely discriminative fit cannot do directly.

36.12.5.1 Calibration Beyond One Temperature

Temperature scaling (exercise 8) works when a classifier's scores are accurate up to a single multiplicative scale. When the distortion is more complex, richer maps help. *Platt scaling* fits both a slope and an offset inside a logistic curve mapping binary scores to probabilities (Platt 1999), while isotonic regression fits any nondecreasing function; Zadrozny and Elkan (2002) applied isotonic regression to calibrate naive Bayes and other classifiers. The trade-off is familiar: a more flexible calibrator requires more held-out data to avoid overfitting, and for modern deep networks a single temperature often works just as well (C. Guo et al. 2017).

36.12.5.2 Semi-Supervised Naive Bayes

In many real problems, labeled examples are expensive while unlabeled inputs are plentiful. Because naive Bayes models how each class generates its features, an unlabeled example fits naturally into the likelihood by treating its missing class as a latent variable. The EM algorithm of Section 36.7 then alternates between computing class posteriors for the unlabeled examples and updating the word or pixel counts using those posteriors as fractional labels. Nigam et al. (2000) demonstrated substantial accuracy gains in text classification when only a handful of documents were labeled, while also warning that unlabeled data can hurt when the generative model fits the data poorly.

36.12.6 General Probability and Statistics Resources

Books

- Introduction to Probability — Blitzstein & Hwang: an inviting introduction to random variables, expectation, conditioning, and the named distributions of Section 36.1 and Section 36.3; a free PDF accompanies Harvard's Stat 110.
- Mathematics for Machine Learning — Deisenroth, Faisal & Ong: Chapter 6 offers a self-contained review of probability and distributions, including the exponential family of Section 36.4.

- All of Statistics — Wasserman: a crisp, computer-scientist-friendly treatment of estimation, testing, confidence intervals, and the bootstrap, covering Section 36.5 and Section 36.6; the companion page includes errata, code, and datasets.
- Computer Age Statistical Inference — Efron & Hastie: explores the bootstrap, large-scale testing, and the relationship between frequentist and Bayesian inference, complementing Section 36.5.
- Pattern Recognition and Machine Learning — Bishop: covers the exponential family, mixture models and EM (Section 36.7), and the Laplace and variational approximations of Section 36.8; Microsoft Research provides a free PDF.
- Probabilistic Machine Learning: An Introduction — Murphy: develops machine learning through probability and decision theory, including MAP estimation (Section 36.6) and generative classifiers (Section 36.11), with a free draft and runnable notebooks.
- Information Theory, Inference, and Learning Algorithms — MacKay: weaves inference together with information theory; its chapters on Laplace's method, Monte Carlo methods, and variational inference accompany Section 36.8.
- Bayesian Data Analysis — Gelman et al.: covers priors, posteriors, Markov chain Monte Carlo with its diagnostics, and practical Bayesian workflow for Section 36.8; the third edition is available as a free PDF.
- High-Dimensional Probability — Vershynin: develops concentration inequalities, sub-Gaussian variables, and high-dimensional geometry, extending Section 36.9.
- Understanding Machine Learning: From Theory to Algorithms — Shalev-Shwartz & Ben-David: derives uniform convergence, Rademacher complexity, and margin bounds as in Section 36.10; the authors provide a free PDF.

Courses and video lectures

- Statistics 110: Probability — Harvard (Blitzstein): provides lecture videos, problem sets, and more than 250 practice problems with solutions for Section 36.1 and Section 36.3.
- Stat 110 lecture videos — YouTube: contains the complete lecture series on probability, distributions, and conditioning.

Tutorials and notes

- Probability Theory Review — Stanford CS229 (Maleki & Do): reviews random variables, expectation, variance, Gaussian distributions, and multivariate probability for machine learning, serving as a concise companion to Section 36.1.

Chapter 37

Information Theory and Divergences

Most models in this book are trained by minimizing a cross-entropy loss: it turns the informal goal of assigning high probability to observed outcomes into a smooth objective for gradient descent. Yet what does that loss number actually measure? When a language model reports 1.2 nats per character, how low could it possibly go, and what are we comparing when we replace cross-entropy with the KL penalty of a variational autoencoder or the critic of a generative adversarial network?

Claude Shannon answered these questions in 1948 while asking how efficiently a message can cross a noisy line. His measure of unpredictability, *entropy*, sets the hard floor below which no model's cross-entropy can fall (Section 3.1.2.3). Three consequences run throughout deep learning. First, cross-entropy exceeds data entropy by the *KL divergence* from data to model, vanishing only when the two match. Second, a model's total loss in bits is literally a compressed file size up to two overhead bits, giving language-model perplexity (Section 8.3.2) a direct operational meaning. Third, learned critics provide only lower bounds on divergences and shared information: in contrastive learning, identifying one partner among N candidates certifies at most $\log N$ nats, and weak adversarial critics (Section 16.2) can make distant distributions appear close.

The chapter unfolds in three parts. The first develops entropy and cross-entropy (Section 37.1) and connects log-loss directly to lossless compression and perplexity (Section 37.2). The second addresses a critical failure of density-ratio divergences (Section 37.3): whenever model samples fail to overlap the data, KL diverges and bounded f-divergences saturate with zero gradient. Comparing expectations of smooth test functions or optimal transport costs (Section 37.4) restores informative gradients, while matching log-density slopes (Section 37.5) compares unnormalized models without intractable partition functions. The third part studies pairs of random variables through mutual information, data processing, Fano's inequality, and the information bottleneck (Section 37.6), culminating in sample-based estimation and InfoNCE (Section 37.7); these final two sections are self-contained after Section 37.1 if you wish to save transport and scores until Chapter 16 and Chapter 17.

We assume familiarity with Jensen's inequality (Chapter 35), conditional distributions, and maximum likelihood (Chapter 36). Error-correcting codes, Kolmogorov complexity, and

information geometry are surveyed in Section 37.8.

37.1 Entropy, Cross-Entropy, and KL Divergence

A loss curve is the first thing we inspect when a classifier trains, and we quickly learn to read its shape: a falling curve is good news, while a flat one is suspicious. The numbers on its vertical axis are harder to interpret. So far we can explain only one of them. A freshly initialized classifier with ten classes reports a loss of about 2.30, a value that Section 4.2 used as a sanity check on an untrained network. That number is $\ln 10$, the cost of naming one of ten labels when the model still considers all ten equally likely. The other end of the curve raises deeper questions. Is a final loss of 0.6 good? When a curve levels off, is the plateau a bug in training or an unavoidable property of the data? And if some of the labels are wrong, how low can any model hope to go?

The cross-entropy loss is the average of $-\log q(y)$, which measures how surprised the model is by the label y that actually occurred. This average splits naturally into two pieces. The first is the *entropy*: the average surprise experienced by someone who already knows the true distribution of the labels. Because entropy is fixed by the data themselves, no model can reduce it. The second piece is the *Kullback–Leibler (KL) divergence*: the extra surprise incurred by using an imperfect model instead of the truth. It is never negative, and it vanishes only when the model matches the true distribution. A loss value can therefore be judged only against this irreducible floor.

We already encountered two of these ideas in the previous chapter: maximum likelihood minimizes a cross-entropy (Section 36.6.2.1), and the KL divergence is never negative (Section 36.4.2), a result known as Gibbs' inequality. Whereas those facts served as tools there, here they are our main subject. We begin with a deck of cards and a four-symbol source, where entropy simply counts yes/no questions. From there we compare pairs of distributions, return to our classifier to find its loss floor, and finally examine what changes when targets are softened or when outcomes are continuous real numbers.

The base of the logarithm is merely a choice of unit: base 2 measures information in *bits*, while base e measures it in *nats*. **We work in nats throughout**, matching the natural logarithm `log` used in numerical libraries and in reported training losses. Bits are more convenient when counting yes/no questions, so we mark quantities in bits with a subscript 2, as in H_2 for entropy. One bit equals $\ln 2 \approx 0.693$ nats. Table 37.1 collects the symbols used across this chapter.

Table 37.1: Symbols of the information theory chapter. The binary entropy $H_b(p)$ is a function of a number, not of a random variable. The mutual information $I(X;Y)$ and the last five rows are defined in later sections.

Symbol	Meaning	Unit
$I(x) = -\log p(x)$	self-information (surprisal) of the outcome x (by convention the letter is shared with the mutual information $I(X;Y)$)	nats
$H(X)$, $H(P)$	entropy of a discrete random variable or distribution; a subscript 2, as in H_2 and CE_2, marks bits	nats

Symbol	Meaning	Unit
$H_b(p)$	binary entropy of a coin with heads probability p	nats
$h(X)$	differential entropy of a continuous random variable	nats, depends on the units of X
$D_{\mathrm{KL}}(P\|Q)$	KL divergence from P to Q	nats
$\mathrm{CE}(P, Q)$	cross-entropy from P to Q	nats
k	number of classes (size of the set of outcomes)	
n	number of examples, or of symbols in a message; the dimension in $\mathbb{R}^n$	
$\mathbf{p}$, $\mathbf{q}$	probability vectors on k classes (truth, model)	
$\mathbf{z}$	logits, with $\mathbf{q} = \mathrm{softmax}(\mathbf{z})$	
ϵ	label-smoothing mass (in later sections, a noise variable)	
τ	temperature	
ρ	label-noise rate: the probability that a recorded label is not the clean class	
$H(X \mid Y)$, $I(X;Y)$	conditional entropy, mutual information	nats
$D_f(P\|Q)$	f-divergence with divergence function f	
T	critic: a function whose averages under two distributions bound a divergence	
g	test function of an integral probability metric	
γ	coupling (transport plan) of two distributions	
N	number of points in each of two samples that are compared; number of candidates in a contrastive batch	

37.1.1 Information and Entropy

37.1.1.1 Self-Information

Consider four reports about a freshly shuffled deck of cards. "I see a card" conveys nothing, because the event is certain. "I see a heart" narrows the outcome to one of four equally likely suits, "It is the three of spades" singles out one of 52 equally likely cards, and reporting the order of the entire deck identifies one of $52!$ possible orderings. The less probable an event is, the more we learn by observing it. Shannon (Shannon 1948) quantified this idea with the *self-information* of an outcome x that occurs with probability $p(x)$,

$$I(x) = -\log p(x).$$

This quantity is also called the *surprisal* of the outcome, the term used in Section 3.1.2.3:

the less we expected an outcome, the more it surprises us. Importantly, the surprise belongs to whoever assigned the probability, so a confidently wrong model finds ordinary events deeply surprising. A certain event carries zero information, and halving an outcome's probability adds $\ln 2$ nats (one bit) to its surprisal. The first three card reports carry 0, $\ln 4 \approx 1.39$, and $\ln 52 \approx 3.95$ nats. In bits, where one bit is the information gained from a single fair coin flip, they carry 0, 2, and 5.70.

Why choose a logarithm? We want three properties: a certain event carries zero information, rarer events carry more information, and the information of two independent events adds up. Because independent probabilities multiply, the third requirement reads $I(p_1 p_2) = I(p_1) + I(p_2)$. The only decreasing functions satisfying this identity are positive multiples of $-\log p$ (Csiszár (2008); Exercise 1 works through the proof). Choosing the constant multiple simply chooses the unit.

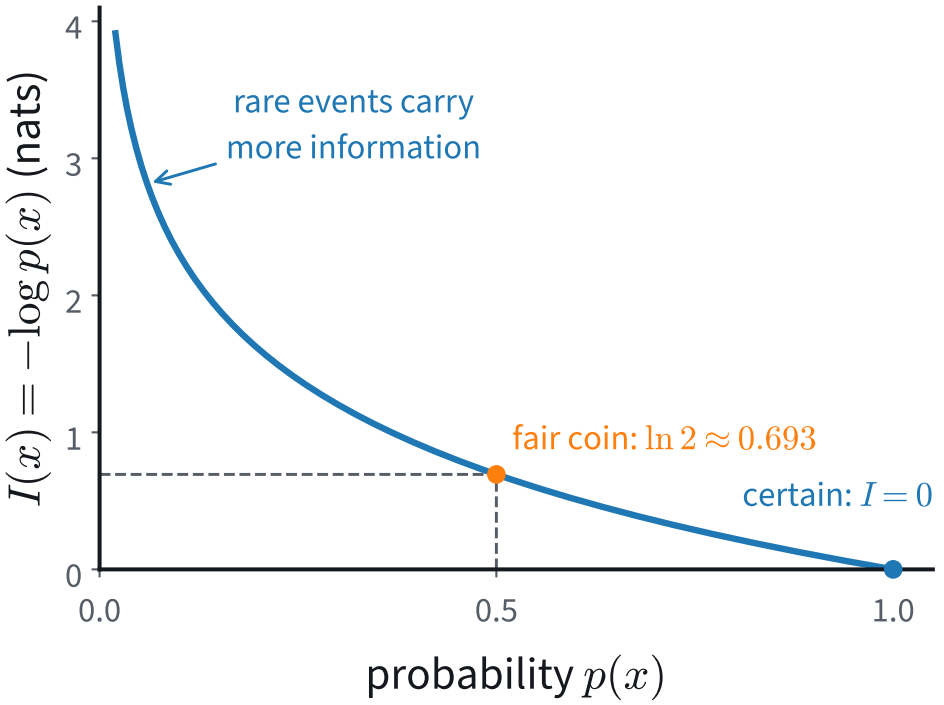


Figure 37.1: Self-information $I(x) = -\log p(x)$ in nats as a function of the probability p. Certain events ($p = 1$) carry zero information, a fair coin ($p = \frac{1}{2}$) carries $\ln 2 \approx 0.693$ nats, and the curve diverges as $p \to 0$.

Figure 37.1 plots the curve: it is decreasing and convex, hits zero at $p = 1$, and diverges as $p \to 0$. The fourth card report, the order of the entire deck, sits far out on this curve at $\ln 52!$. We can compute its value via the log-gamma function, $\ln 52! = \ln \Gamma(53)$, without ever forming the 68-digit integer $52!$.

```
deck_order = math.lgamma(53)          # ln 52! without forming 52!
print(f'the order of the deck: {deck_order:.2f} nats '
      f'= {deck_order / math.log(2):.1f} bits')
```

```
the order of the deck: 156.36 nats = 225.6 bits
```

Naming the order of a shuffled deck carries 156 nats, roughly as much information as 226 fair coin flips.

37.1.1.2 Entropy

We can compute a surprisal only after an outcome arrives. Beforehand, we can ask how surprised we expect to be on average. Let X be a discrete random variable taking the value x_i with probability p_i. Its *entropy* is the expected self-information of its outcome,

$$H(X) = -\mathbb{E}_{x\sim P}[\log p(x)] = -\sum_i p_i \log p_i, \tag{37.1}$$

with the convention $0 \log 0 = 0$, justified by continuity ($t \log t \to 0$ as $t \to 0$): an impossible outcome contributes nothing. Because the entropy depends only on the distribution P of X, we write $H(P)$ and $H(X)$ interchangeably.

On a finite set, entropy in bits has a concrete operational meaning: it counts the average number of yes/no questions a good questioner needs to identify the outcome. Take the distribution $P = (\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{8})$ on four symbols a, b, c, d. We will carry this example through the section and call it the *dyadic source*, since all of its probabilities are powers of $\frac{1}{2}$. A good questioner first asks "Is it a?", settling the matter half the time. If not, they ask "Is it b?", and then "Is it c?", where "no" leaves only d. The expected number of questions is

$$\tfrac{1}{2}\cdot 1 + \tfrac{1}{4}\cdot 2 + \tfrac{1}{8}\cdot 3 + \tfrac{1}{8}\cdot 3 = 1.75,$$

which matches the entropy in bits: $H_2(P) = \frac{1}{2}\log_2 2 + \frac{1}{4}\log_2 4 + 2\cdot\frac{1}{8}\log_2 8 = 1.75$ bits, or 1.213 nats. Each outcome is identified in $\log_2(1/p_i)$ questions, which is its self-information in bits.

By contrast, a questioner who ignores the probabilities asks "Is it a or b?" and then "Which of the two?", always needing 2 questions. Notice that a questioning strategy is a binary code in disguise: write 1 for yes and 0 for no, and the sequence of answers that identifies an outcome becomes its codeword, here 1, 01, 001, and 000 for a, b, c, and d. The average question count matched the entropy here because every probability was a power of $\frac{1}{2}$. In the language of codes, Section 37.2 proves that the count is never far off: for any distribution on a finite set, the best strategy needs between H_2 and $H_2 + 1$ questions on average.

In code, the function `entropy` masks out zero probabilities, implementing $0 \log 0 = 0$. We will reuse it throughout the section, starting with the dyadic source.

```python
def entropy(p):
    """Entropy of a probability vector, in nats (0 log 0 = 0 by masking)."""
    p = np.asarray(p, dtype=float)
    p = p[p > 0]
    return float(-np.dot(p, np.log(p)))

p_dyadic = np.array([1 / 2, 1 / 4, 1 / 8, 1 / 8])
h = entropy(p_dyadic)
print(f'H = {h:.3f} nats = {h / math.log(2):.2f} bits')
```

```
H = 1.213 nats = 1.75 bits
```

It returns the 1.75 bits that we counted in questions.

Because $0 \le p_i \le 1$, every term $-p_i \log p_i$ is non-negative, so $H(X) \ge 0$, with equality only for a point mass (a variable whose outcome is certain). Notice also that entropy depends only on the list of probabilities, not on the numerical values or names of the outcomes. A variable taking the values 1, 100, 10^6, and 10^9 with the dyadic probabilities still has an entropy of 1.75 bits.

37.1.1.3 Joint and Conditional Entropy

In machine learning, a label always comes paired with an input, and a character comes after the text preceding it. We therefore need both the entropy of two variables taken together and the entropy remaining in one variable once the other is known. The first requires nothing new: a pair (X, Y) with joint probability mass function $p(x, y)$ is simply a single random variable whose outcomes are pairs, so its *joint entropy* is $H(X, Y) = -\mathbb{E}[\log p(X, Y)]$. The *conditional entropy* is the average surprisal left in Y once X is observed,

$$H(Y \mid X) = -\mathbb{E}[\log p(Y \mid X)] = \sum_x p(x)\, H(Y \mid X = x),$$

where $H(Y \mid X = x)$ is the entropy of the conditional distribution $p(\cdot \mid x)$. Because $\log p(x, y) = \log p(x) + \log p(y \mid x)$, taking expectations on both sides yields the *chain rule* for entropy:

$$H(X, Y) = H(X) + H(Y \mid X). \tag{37.2}$$

In words, the entropy of a pair is the entropy of the first variable plus the uncertainty that remains in the second once the first is known. Applied repeatedly, the chain rule gives $H(X_1, \dots, X_n) = \sum_{i=1}^{n} H(X_i \mid X_1, \dots, X_{i-1})$: the entropy of a sequence is the sum of the entropies of its symbols, each conditioned on all preceding symbols. A language model predicts text in this same left-to-right order, one symbol at a time.

The questions game illustrates this identity directly. Let X be the answer to the first question, "Is it a?", and let Y be the outcome of the dyadic source. Because the answer is a fair coin, $H_2(X) = 1$ bit. After "yes", nothing remains to ask (0 bits). After "no", the outcome is b, c, or d with probabilities $(\frac{1}{2}, \frac{1}{4}, \frac{1}{4})$, whose entropy is 1.5 bits. Averaging over the two answers gives $H_2(Y \mid X) = \frac{1}{2} \cdot 0 + \frac{1}{2} \cdot 1.5 = 0.75$ bits. By the chain rule, $H_2(X, Y) = 1 + 0.75 = 1.75$ bits: the first answer supplies one bit and the remaining questions supply the rest. This sum equals $H_2(Y)$, as it must: knowing the outcome Y completely determines the answer X, so $H(X \mid Y) = 0$, and the chain rule in the reverse order reads $H(X, Y) = H(Y) + 0$.

37.1.1.4 Maximum Entropy on a Finite Set

How large can the entropy of a variable with k outcomes be? Let us start with the smallest case, a single coin flip. A Bernoulli variable with heads probability p has the *binary entropy*

$$H_b(p) = -p \log p - (1 - p) \log(1 - p),$$

which is a function of the scalar p rather than of a random variable. Figure 37.2 plots this curve. The entropy vanishes at $p \in \{0, 1\}$, where the outcome is certain, and reaches its peak at the fair coin, $H_b(\frac{1}{2}) = \ln 2 \approx 0.693$ nats (one bit). A biased coin with $p = 0.1$ has $H_b(0.1) \approx 0.325$ nats, because its outcome is much easier to guess.

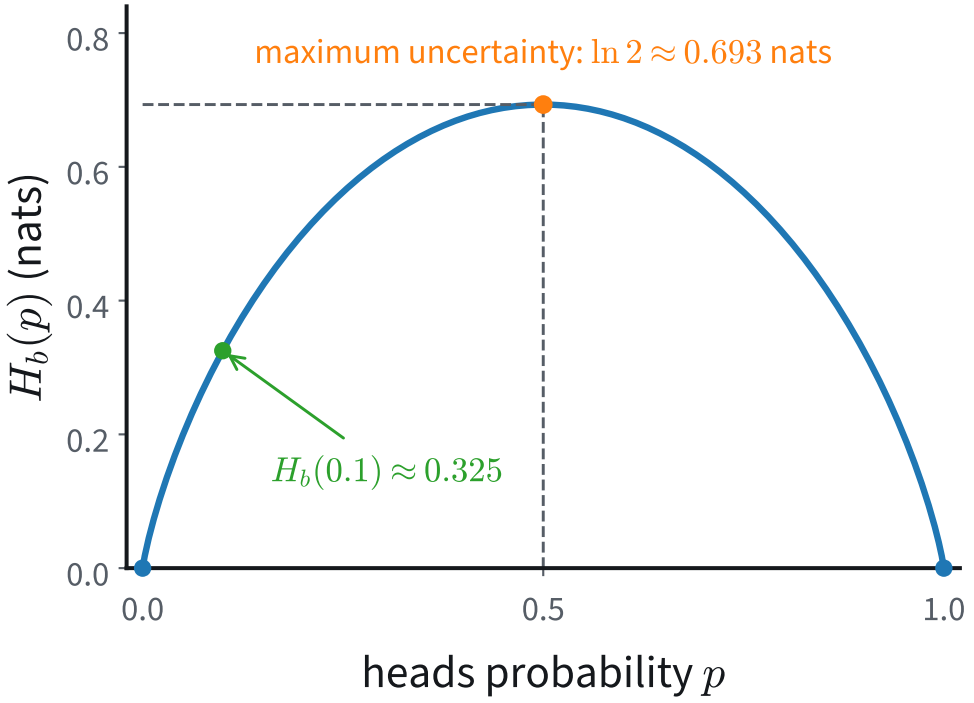


Figure 37.2: The binary entropy $H_b(p)$ in nats, symmetric about $p = \frac{1}{2}$, where it peaks at $\ln 2 \approx 0.693$ nats. At the deterministic endpoints $p = 0$ and $p = 1$, the outcome carries no surprise and the entropy vanishes. The curve is concave: any chord lies below it.

Notice that the curve is concave: the chord joining any two of its points lies below the curve. It peaks at the uniform coin, and the same principle holds for any finite number of outcomes. If X takes at most k values, then

$$H(X) \le \log k,$$

with equality if and only if X is uniform over all k values. The uniform distribution spreads its probability mass as evenly as possible, making the outcome maximally hard to guess. For four outcomes, the upper bound is $\log_2 4 = 2$ bits, which sits 0.25 bits above the 1.75 bits of the dyadic source. We postpone the proof for a moment until Gibbs' inequality is in place, at which point it takes just a single line. Since that inequality compares two distributions, we turn to comparing distributions next.

37.1.2 KL Divergence and Cross-Entropy

37.1.2.1 The Kullback–Leibler Divergence

Suppose we want to measure how badly a model Q misrepresents a true distribution P. Information theory answers this with the *Kullback–Leibler (KL) divergence*, also known as *relative entropy* (Kullback and Leibler 1951): the average extra surprisal incurred by someone who believes Q when outcomes are actually drawn from P. In the questions game, the KL divergence in bits is the average number of extra questions that a strategy designed for Q needs on outcomes drawn from P, up to rounding to whole questions. Let P and Q have probability mass functions (or densities) p and q. The KL divergence from P to Q is

$$D_{\mathrm{KL}}(P\|Q) = \mathbb{E}_{x\sim P}\left[\log \frac{p(x)}{q(x)}\right]. \tag{37.3}$$

The quantity inside the expectation is $\log p(x) - \log q(x) = (-\log q(x)) - (-\log p(x))$, which is the surprisal of the believer in Q minus the surprisal of the believer in P. It is positive where Q underestimates an outcome ($q(x) < p(x)$) and negative where Q overestimates it. The KL divergence averages this difference under the true distribution P.

Because P supplies both the samples and the numerator, the definition is asymmetric: in general, $D_{\mathrm{KL}}(P\|Q) \neq D_{\mathrm{KL}}(Q\|P)$. Outcomes with $p(x) = 0$ contribute nothing, following the convention $0 \log(0/q) = 0$. If any outcome has $p(x) > 0$ while $q(x) = 0$, however, then $D_{\mathrm{KL}}(P\|Q) = +\infty$.

A small numerical example makes the definition concrete, and we will carry it through the rest of the section. Let $P = (0.6, 0.3, 0.1)$ be the truth and $Q = (0.2, 0.5, 0.3)$ a model on three outcomes. Then

$$D_{\mathrm{KL}}(P\|Q) = 0.6 \ln \frac{0.6}{0.2} + 0.3 \ln \frac{0.3}{0.5} + 0.1 \ln \frac{0.1}{0.3} = 0.659 - 0.153 - 0.110 = 0.396 \text{ nats},$$

whereas the reverse direction gives $D_{\mathrm{KL}}(Q\|P) = 0.365$ nats. In bits, the two directions are 0.571 and 0.527. Not only do they differ, but the first value tells us that a questioner who believes Q asks 0.57 more questions per outcome than necessary, averaged over long runs of outcomes drawn from P.

Here is the discrete KL divergence in code, implementing $0 \log 0 = 0$ by masking out the zero probabilities of P, just as we did in `entropy`. We evaluate it on our running pair in both directions, as well as on a model $Q' = (0.5, 0.5, 0)$ that declares the third outcome impossible.

```
def kl_divergence(p, q):
    """KL(P || Q) for two probability vectors, in nats; inf where q = 0 < p."""
    p, q = np.asarray(p, dtype=float), np.asarray(q, dtype=float)
    m = p > 0
    if np.any(q[m] == 0):
        return float('inf')
    return float(np.dot(p[m], np.log(p[m] / q[m])))

p = np.array([0.6, 0.3, 0.1])
```

```
q = np.array([0.2, 0.5, 0.3])
print(f'KL(P||Q) = {kl_divergence(p, q):.4f} nats, '
      f'KL(Q||P) = {kl_divergence(q, p):.4f} nats')
print(f"KL(P||Q') = {kl_divergence(p, [0.5, 0.5, 0.0])}")
```

```
KL(P||Q) = 0.3961 nats, KL(Q||P) = 0.3653 nats
KL(P||Q') = inf
```

The second line is the one to remember. The model Q' assigns zero probability to an outcome that occurs one time in ten, and its KL divergence is infinite: a model that declares an event impossible is infinitely surprised when that event happens.

37.1.2.2 Gibbs' Inequality

Could the average extra surprisal ever be negative? If it could, a wrong model would be less surprised on average than the truth itself. Gibbs' inequality (Gibbs 1902) rules that out: no model beats the truth on average. Because everything that follows rests on this result, we prove it in full as a consequence of averaging a concave function. For any concave function f and any random variable X with finite mean, *Jensen's inequality* states that

$$\mathbb{E}[f(X)] \le f(\mathbb{E}[X]).$$

In words, averaging first and applying f second yields the larger value. When X takes two values, this is simply the statement that the chord lies below the curve in Figure 37.2. If f is strictly concave, equality holds only when X is constant with probability one (Section 35.3.3.1 proves the general case). We need Jensen's inequality for just one function, the logarithm, which is strictly concave: the logarithm of an average is at least the average of the logarithms.

Proposition (Gibbs' inequality). *For any distributions P and Q on the same space,*

$$D_{\mathrm{KL}}(P\|Q) \ge 0,$$

with equality if and only if $P = Q$.

Proof. If $q(x) = 0$ for some x with $p(x) > 0$, then the divergence is $+\infty$ and there is nothing to prove, so assume otherwise. Rewrite the divergence as minus the average of $\log(q(x)/p(x))$ under $x \sim P$, and apply Jensen's inequality to the logarithm:

$$\begin{aligned} D_{\mathrm{KL}}(P\|Q) &= -\mathbb{E}_{x\sim P}\left[\log \frac{q(x)}{p(x)}\right] \ge -\log \mathbb{E}_{x\sim P}\left[\frac{q(x)}{p(x)}\right] \\ &= -\log \sum_{x\,:\,p(x)>0} q(x) = -\log Q(\operatorname{supp} P) \ge 0. \end{aligned}$$

The leading minus sign flips Jensen's "at most" into "at least". The rest of the proof happens in the second line. The expectation runs over the support of P, the set of outcomes with $p(x) > 0$. The factors $p(x)$ cancel, leaving the total Q-probability of that support, a number at most 1, whose negative logarithm is therefore non-negative. For continuous densities, the same steps hold with integrals in place of sums. For equality to hold, both inequalities

must be tight. Because the logarithm is strictly concave, the first is an equality only when $q(x)/p(x)$ equals a constant c for P-almost every x (that is, outside a set of P-probability zero), and the second requires $Q(\operatorname{supp} P) = 1$. Summing (or integrating) $q = c\,p$ over the support of P then forces $c = 1$, so $q = p$ (again outside a set of probability zero for densities). ■

The proof rests on a single idea. Averaged over the truth, the ratio q/p is at most 1, so its logarithm averages to at most 0, with equality only when q/p does not vary at all (meaning $q = p$). A model incurs no extra surprisal only when it is right.

Gibbs' inequality immediately settles the uniform upper bound we left unproved. For any distribution P on k outcomes and the uniform distribution U on the same set, $D_{\mathrm{KL}}(P\|U) = \sum_i p_i \log(k\,p_i) = \log k - H(P)$, which is non-negative and vanishes only when $P = U$.

It also proves that conditioning cannot increase entropy on average. Compare the joint distribution of a pair (X, Y) with the product of its marginals. The log-ratio is $\log \frac{p(x,y)}{p(x)\,p(y)} = \log p(y \mid x) - \log p(y)$, whose expectation under the joint distribution is $H(Y) - H(Y \mid X)$. Because that expectation is a KL divergence,

$$H(Y) - H(Y \mid X) = D_{\mathrm{KL}}(p_{X,Y} \,\|\, p_X\, p_Y) \geq 0,$$

with equality if and only if X and Y are independent. On average, knowing X never makes Y more uncertain. A *particular* observation, however, can. Suppose Y is tails for certain when $X = 0$, and a fair coin when $X = 1$, where $X = 1$ occurs one time in five. Overall, Y lands heads with probability 0.1, so $H(Y) = H_b(0.1) \approx 0.325$ nats. Observing $X = 1$ raises our uncertainty about Y to $\ln 2 \approx 0.693$ nats, yet the conditional entropy averages over both outcomes of X, giving $H(Y \mid X) = 0.2 \cdot 0.693 \approx 0.139$ nats, less than half of $H(Y)$.

37.1.2.3 Cross-Entropy

Classifiers are trained with a close relative of the KL divergence. The *cross-entropy* from P to Q is the average surprisal of a believer in Q on outcomes drawn from P,

$$\mathrm{CE}(P, Q) = -\mathbb{E}_{x \sim P}[\log q(x)]. \tag{37.4}$$

Adding and subtracting $\log p(x)$ inside the expectation, which is valid whenever $H(P)$ is finite, splits the model's surprisal into the truth's own entropy plus the excess divergence:

$$\mathrm{CE}(P, Q) = H(P) + D_{\mathrm{KL}}(P\|Q). \tag{37.5}$$

Cross-entropy is therefore an irreducible entropy floor plus a non-negative excess: by Gibbs' inequality, $\mathrm{CE}(P, Q) \geq H(P)$, with equality if and only if $Q = P$. For our running pair, $\mathrm{CE}(P, Q) = -(0.6 \ln 0.2 + 0.3 \ln 0.5 + 0.1 \ln 0.3) = 1.294$ nats and $H(P) = 0.898$ nats; their difference, 0.396 nats, is the KL divergence we computed earlier. Because $H(P)$ does not depend on Q, minimizing the cross-entropy with respect to Q is equivalent to minimizing the KL divergence. Maximum likelihood on discrete data does precisely this: its average negative log-likelihood is the cross-entropy from the empirical data distribution to the model (Section 36.6.2.1).

Figure 37.3 illustrates this decomposition in the questions game for the dyadic source P and the uniform model $U = (\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4})$. Asking questions adapted to P takes $H_2(P) = 1.75$ questions on average. Asking two fixed questions under the uniform strategy takes $\mathrm{CE}_2(P, U) = 2$ questions, so the waste is $D_{\mathrm{KL}}(P\|U) = 0.25$ bits. Since a questioning strategy is a binary code, these same numbers are average codeword lengths in bits per symbol. (For this particular pair, the reverse divergence $D_{\mathrm{KL}}(U\|P)$ also happens to equal 0.25 bits, which is why we used a different pair earlier to show that KL divergence is asymmetric.)

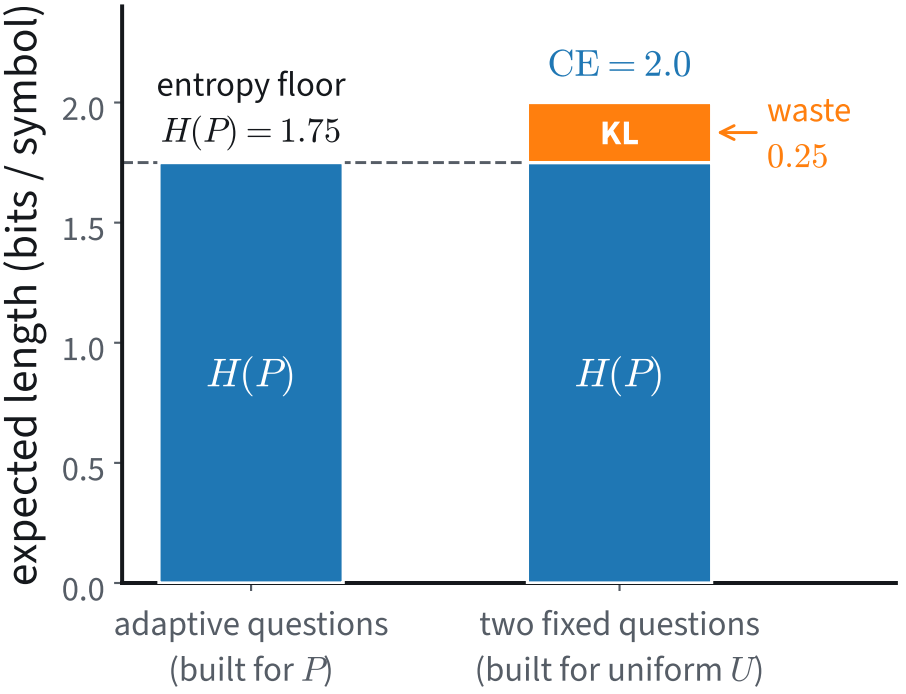


Figure 37.3: The decomposition Equation 37.5 for the dyadic source $P = (\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{8})$ and the uniform model U, in bits. Asking questions adapted to P takes $H_2(P) = 1.75$ questions on average, asking two fixed questions takes $\mathrm{CE}_2(P, U) = 2.0$, and the waste is $D_{\mathrm{KL}}(P\|U) = 0.25$ bits.

37.1.2.4 The Chain Rule for KL Divergence

Entropy splits along a pair: the surprise in (X, Y) is the surprise in X plus the remaining surprise in Y (Section 37.1.1.3). The KL divergence splits the same way, and for the same reason. Let $P_{X,Y}$ and $Q_{X,Y}$ be two joint distributions, factored as $p(x, y) = p(x)\, p(y \mid x)$ and $q(x, y) = q(x)\, q(y \mid x)$. The logarithm of the ratio $p(x, y)/q(x, y)$ is then a sum of two terms, and taking expectations under $P_{X,Y}$ yields the *chain rule* for KL divergence:

$$D_{\mathrm{KL}}(P_{X,Y}\|Q_{X,Y}) = D_{\mathrm{KL}}(P_X\|Q_X) + \mathbb{E}_{x\sim P_X}[D_{\mathrm{KL}}(P_{Y|x}\|Q_{Y|x})]. \tag{37.6}$$

A wrong model of the pair pays for its error about X, and then, on average, for its error about Y once X is known. Note that the average runs over the truth P_X, not over the model. The identity is one line of algebra, yet the rest of the book leans on three of its consequences.

First, KL divergence adds over independent coordinates. If X and Y are independent under both P and Q, the conditional term no longer depends on x, and the divergence of the pair is the sum of the two divergences. Ten independent draws from our running pair thus separate P from Q by $10 \cdot 0.396 = 3.96$ nats, which is why long samples tell two distributions apart so much better than short ones. Second, the divergence between two joint distributions is at least the divergence between their marginals, because the conditional term is non-negative by Gibbs' inequality. Throwing a variable away can only make two distributions harder to tell apart. Latent variable models rely on this: the joint divergence over data and latents is, up to a constant, the objective they can compute, and it exceeds the divergence over the data alone by the conditional term, the gap of the evidence lower bound (Section 36.7). Third, applying the rule repeatedly to a sequence gives

$$D_{\mathrm{KL}}(P_{X_1,\dots,X_n} \| Q_{X_1,\dots,X_n}) = \sum_{i=1}^{n} \mathbb{E}_{x_{<i} \sim P}\big[D_{\mathrm{KL}}(P_{X_i|x_{<i}} \| Q_{X_i|x_{<i}})\big],$$

so the excess loss of an autoregressive model on a whole sequence is the sum of its excess losses at the individual steps. Predicting each next token well is therefore the same task as modeling the entire sequence well, and Section 37.2.4.1 uses this decomposition to bound the loss of every language model from below.

37.1.3 The Classification Loss

In a k-class classification problem, the target for a single example is the one-hot distribution placing all its mass on the recorded label y, while the model outputs a predicted distribution $\mathbf{q} = (q_1, \dots, q_k)$. The cross-entropy Equation 37.4 between them collapses to a single term, $-\log q_y$, which is simply the model's surprisal at the observed label. Averaged over n examples with labels y_i and predictions $\mathbf{q}_i$, this gives the *cross-entropy loss*

$$\mathrm{CE} = -\frac{1}{n} \sum_{i=1}^{n} \log q_{i,y_i},$$

measured in nats. Deep learning frameworks compute this loss directly from the *logits* $\mathbf{z}$, the unnormalized scores whose softmax is $\mathbf{q}$. They apply a numerically stable log-softmax internally, because forming the probabilities first can overflow for large logits and underflow to $\log 0$ for very negative ones (Section 3.3.5.2; Section 35.8.2 derives the stable form). In the subsections that follow, we unpack what this loss tells us about noise in the data, probability calibration, and softened targets.

37.1.3.1 The Irreducible Loss

At the start of the section, we asked how low the loss can go when some labels are wrong. On a fixed training set, the answer is uninformative: a network large enough to memorize the dataset can fit every recorded label, the wrong ones included, and drive the training loss toward zero. The question has real content only on fresh examples that the model has not seen, and there the one-hot target is not the full truth. In real data, the label is rarely a deterministic function of the input: the same image may receive different labels due to genuine ambiguity or annotation errors. The truth for an input $\mathbf{x}$ is the conditional distribution from which its label is drawn, and the recorded label is a single sample from it.

Suppose the input $\mathbf{x}$ and label $y \in \{1, \dots, k\}$ follow a joint distribution, where $p(\cdot \mid \mathbf{x})$ is the true conditional distribution of the label and $q(\cdot \mid \mathbf{x})$ is any model's prediction. As the number of fresh examples grows, their average loss converges to its expectation over $(\mathbf{x}, y)$, which we call the *population loss*. No model can push this population loss below the average entropy of the label given the input.

Proposition (the irreducible classification loss). *The population cross-entropy loss of any model satisfies*

$$\begin{aligned} \mathbb{E}_{(\mathbf{x},y)}\big[-\log q(y \mid \mathbf{x})\big] &= \mathbb{E}_{\mathbf{x}}\big[H(p(\cdot \mid \mathbf{x}))\big] + \mathbb{E}_{\mathbf{x}}\big[D_{\mathrm{KL}}(p(\cdot \mid \mathbf{x}) \,\|\, q(\cdot \mid \mathbf{x}))\big] \\ &\geq \mathbb{E}_{\mathbf{x}}\big[H(p(\cdot \mid \mathbf{x}))\big], \end{aligned} \tag{37.7}$$

with equality if and only if $q(\cdot \mid \mathbf{x}) = p(\cdot \mid \mathbf{x})$ for almost every $\mathbf{x}$ (that is, for all inputs outside a set of probability zero).

Proof. Condition on $\mathbf{x}$. The inner expectation over y is the cross-entropy $\mathrm{CE}(p(\cdot \mid \mathbf{x}), q(\cdot \mid \mathbf{x}))$, so the decomposition Equation 37.5 applies pointwise at every $\mathbf{x}$ (all entropies are finite, at most $\log k$). Taking the outer expectation over $\mathbf{x}$ gives the identity. By Gibbs' inequality, every KL term is non-negative, and an average of non-negative terms vanishes if and only if almost all of them are zero. ■

This floor is the conditional entropy of the label given the input (Section 37.1.1.3). It is an intrinsic property of the data: no architecture, extra training data, or longer training schedule can lower it. Of all the results in this section, this is the one to keep in mind whenever you read a loss curve. Three cases illustrate what it implies:

- A model that predicts the uniform distribution suffers a cross-entropy of $\ln k$ on every input regardless of the labels, giving the familiar 2.30 nats for $k = 10$.
- When the label is a deterministic function of the input, every $H(p(\cdot \mid \mathbf{x}))$ is zero, and a perfect model achieves a loss of 0.
- Under symmetric label noise, the clean class is determined by the input, while the recorded label matches the clean class with probability $1-\rho$ and each of the remaining $k-1$ classes with probability $\rho/(k-1)$. At every input, the conditional entropy is $H = -(1-\rho)\log(1-\rho) - \rho \log \frac{\rho}{k-1} = H_b(\rho) + \rho \log(k-1)$. For $\rho = 0.1$ and $k = 10$, this floor is $0.325 + 0.220 = 0.545$ nats. As long as $1 - \rho > \rho/(k-1)$, predicting the clean class remains optimal, so the highest achievable accuracy is $1 - \rho$ (here 90%).

Before running the next cell, pause and guess the loss of a confident predictor that assigns probability 0.99 to the clean class in this noisy setting. It is right nine times out of ten, and nearly certain every time. The code compares it against the uniform predictor and the *Bayes predictor*, which reports the true conditional distribution $q = p$.

```
k, rho = 10, 0.1
p_noisy = np.array([1 - rho] + [rho / (k - 1)] * (k - 1))  # clean class first
predictors = {'uniform': np.full(k, 1 / k),
              'confident 0.99': np.array([0.99] + [0.01 / (k - 1)] * (k - 1)),
              'Bayes q = p': p_noisy}
print(f'floor = H(p) = {entropy(p_noisy):.3f} nats')
for name, q_hat in predictors.items():
    ce = float(-np.dot(p_noisy, np.log(q_hat)))
    print(f'{name:15s} CE = {ce:.3f} nats')
```

```
floor = H(p) = 0.545 nats
uniform         CE = 2.303 nats
confident 0.99  CE = 0.689 nats
Bayes q = p     CE = 0.545 nats
```

The overconfident predictor suffers a loss of 0.689, well above the floor: every flipped label catches it by surprise, and its high confidence costs far more on the tenth of labels that are flipped than it gains on the rest. Only the Bayes predictor reaches the floor.

We can now answer the question from the opening: is a final loss of 0.6 good? On clean labels, where the floor is zero, 0.6 still leaves plenty of room for a better model. On these noisy labels, however, it sits within 0.06 nats of the floor and leaves almost none. A plateau at the floor is simply a property of the data; only a plateau above the floor points to a limitation of the model or its training. Bear in mind that all of this applies to fresh data. On a held-out split, a loss *below* the floor cannot come from a better model, and Exercise 2 asks what it signals instead.

So far we derived the loss floor for a single penalty ($-\log q_y$), for unmodified one-hot labels, and for countable outcomes. In the remainder of this section we relax each of these three assumptions in turn, examining other penalties, softened targets, and continuous real-valued outcomes.

37.1.3.2 Proper Scoring Rules

The loss floor is achieved only by the true conditional probabilities, which is just the property we want if we intend to interpret a classifier's outputs as probabilities. Decision theory gives this property a name. A *scoring rule* assigns a penalty $S(\mathbf{q}, y)$ for reporting the probability vector $\mathbf{q}$ when class y actually occurs. The rule is *proper* if truthful reporting is optimal: when labels follow $\mathbf{p}$, the expected penalty $\mathbb{E}_{y\sim\mathbf{p}}[S(\mathbf{q}, y)]$ is minimized by reporting $\mathbf{q} = \mathbf{p}$. It is *strictly proper* if this truthful report is the unique minimizer. The cross-entropy loss is the *log score* $S(\mathbf{q}, y) = -\log q_y$, and Gibbs' inequality shows that it is strictly proper: its expected penalty is $\mathrm{CE}(\mathbf{p}, \mathbf{q})$, which is uniquely minimized at $\mathbf{q} = \mathbf{p}$.

Consequently, at the population optimum over a model class rich enough to contain the truth, a classifier trained with cross-entropy outputs the true conditional probabilities. In particular, its predictions are *calibrated*: among all inputs for which the model predicts probability 0.8 for a given class, that class actually occurs 80% of the time. (A finite training set or a restricted model class can still leave a trained network miscalibrated in practice.)

The log score is not the only strictly proper scoring rule. The *Brier score*, $S(\mathbf{q}, y) = \sum_j (q_j - \mathbf{1}(j = y))^2$, is strictly proper as well, and unlike the log score it imposes a bounded penalty on confident errors where $-\log q_y$ diverges (Gneiting and Raftery 2007). Exercise 3 computes its excess penalty and examines a seemingly reasonable rule that turns out to be improper.

37.1.3.3 Label Smoothing

Until now, our training target has been the one-hot label. Replacing it with a softer target changes both what the loss asks of the logits and where its floor lies, and there is a practical reason to do so. One-hot targets demand something a softmax layer cannot actually produce: assigning probability 1 to the true class requires its logit to be infinitely larger than all the others. As a result, the loss has no finite minimizer, and gradient descent keeps pushing the logits further apart for as long as training runs. *Label smoothing* (Szegedy et al. 2016)

fixes this by mixing the one-hot target $\mathbf{e}_y$ with a fraction ϵ of the uniform distribution over all k classes:

$$\mathbf{p}^{\epsilon} = (1-\epsilon)\,\mathbf{e}_y + \frac{\epsilon}{k}\,\mathbf{1}. \tag{37.8}$$

Under this target, the true class receives probability $1-\epsilon+\epsilon/k$ and every other class receives ϵ/k. The training objective is the ordinary cross-entropy $\mathrm{CE}(\mathbf{p}^{\epsilon}, \mathbf{q})$ against this smoothed target (the same loss used in Section 7.6). It is minimized at $\mathbf{q} = \mathbf{p}^{\epsilon}$, which a softmax achieves with a finite gap between the logit of the true class and each competing logit:

$$z_y - z_j = \log\frac{1-\epsilon+\epsilon/k}{\epsilon/k} = \log\Big(1 + \frac{k(1-\epsilon)}{\epsilon}\Big).$$

For $k = 10$ classes and $\epsilon = 0.1$, this optimal logit gap is $\ln 91 \approx 4.51$, and the loss at the optimum equals the entropy of the smoothed target, $H(\mathbf{p}^{\epsilon}) = 0.500$ nats.

It is instructive to see how quickly the logits drift without smoothing. Consider a single example whose logits start at zero, updated by gradient descent with step size 1. The gradient of the cross-entropy with respect to the logits is $\mathbf{q} - \mathbf{p}^{\epsilon}$. With a one-hot target, all wrong classes stay tied, and each step widens the gap g between the true class and the rest by $(1-q_y) + q_j = k/(e^g + k - 1)$, which is approximately ke^{-g} once g is large. Incrementing g by ke^{-g} multiplies e^g by $e^{ke^{-g}} \approx 1 + ke^{-g}$, thereby adding roughly k to e^g on each step. After t steps we therefore expect $e^g \approx kt$, or $g \approx \ln(kt)$. The gap never stops growing, though its growth is logarithmic: ten times as many steps add only $\ln 10 \approx 2.30$. After $10{,}000$ steps, we expect $g \approx \ln 10^5 \approx 11.51$. The next cell runs this gradient descent both without and with smoothing.

```
k, checkpoints = 10, (10, 100, 1000, 10000)
for eps in (0.0, 0.1):
    p_eps = np.full(k, eps / k)
    p_eps[0] += 1 - eps                  # the true class first
    z = np.zeros(k)                      # the logits of one example
    for step in range(1, checkpoints[-1] + 1):
        q = np.exp(z - z.max())
        z -= q / q.sum() - p_eps         # gradient step, step size 1
        if step in checkpoints:
            log_q = z - z.max() - np.log(np.exp(z - z.max()).sum())
            print(f'eps = {eps}  step {step:5d}  gap = {z[0] - z[1]:5.2f}  '
                  f'loss = {-p_eps @ log_q:.4f}')
print(f'optimum for eps = 0.1: gap = ln 91 = {math.log(91):.2f}, '
      f'loss = H(p_eps) = {entropy(p_eps):.4f}')
```

```
eps = 0.0  step    10  gap =  4.29  loss = 0.1158
eps = 0.0  step   100  gap =  6.87  loss = 0.0093
eps = 0.0  step  1000  gap =  9.21  loss = 0.0009
eps = 0.0  step 10000  gap = 11.51  loss = 0.0001
eps = 0.1  step    10  gap =  3.81  loss = 0.5243
eps = 0.1  step   100  gap =  4.51  loss = 0.5003
eps = 0.1  step  1000  gap =  4.51  loss = 0.5003
```

```
eps = 0.1  step 10000  gap =  4.51  loss = 0.5003
optimum for eps = 0.1: gap = ln 91 = 4.51, loss = H(p_eps) = 0.5003
```

With one-hot targets, the logit gap reaches 4.29, 6.87, 9.21, and 11.51. Our approximation $\ln(kt)$ predicts 4.61, 6.91, 9.21, and 11.51: the actual gap starts slightly below the estimate (where ke^{-g} is still rough) and agrees to two decimal places from $1{,}000$ steps onward. Meanwhile, the loss keeps inching downward without ever reaching zero. In a neural network large enough to fit its training set, this unbounded drift manifests as weights that keep growing and predicted probabilities near 1 on every training example, mislabeled ones included. With $\epsilon = 0.1$, by contrast, the gap settles at 4.51 by step 100 and stays there, while the loss comes to rest at 0.500.

Training now has a finite optimum, though a smoothed model must be interpreted with care. Even when it predicts the smoothed target without error, its training loss plateaus at $H(\mathbf{p}^{\epsilon})$ rather than zero. Moreover, its outputs are no longer the true conditional probabilities: no class can receive probability greater than $1-\epsilon+\epsilon/k$ (here 0.91). In practice, $\epsilon = 0.1$ is the standard choice, and the training recipe of Section 7.6 adopts it. Exercise 4 explores what label smoothing does to the logits of the incorrect classes.

37.1.3.4 Knowledge Distillation

Soft targets can also come from another trained model. *Knowledge distillation* (Hinton et al. 2015) trains a compact *student* network to imitate a larger *teacher* network, an approach applied to language models in Section 26.1. Instead of training on a one-hot label, the student trains on the teacher's full probability distribution over classes. The fact that a handwritten "2" resembles a "7" much more than it resembles a "0" is information a one-hot label discards; it lives in the teacher's logits for the wrong classes. (Exercise 4 asks why a teacher trained with label smoothing passes on less of this information.)

As with any fixed target distribution, the loss is minimized when the student reproduces the target itself, so the decomposition Equation 37.5 adds nothing new here. What does call for explanation is the scale of the loss, which carries a factor that looks arbitrary at first glance.

Because a well-trained teacher's softmax outputs are nearly one-hot, the probabilities it assigns to wrong classes are usually too tiny to exert much pull on the student. We therefore *soften* both distributions first by dividing the logits by a *temperature* $\tau > 0$ before applying the softmax:

$$p_j^{(\tau)} = \operatorname{softmax}(\mathbf{z}^{\text{tea}}/\tau)_j, \qquad q_j^{(\tau)} = \operatorname{softmax}(\mathbf{z}^{\text{stu}}/\tau)_j.$$

Setting $\tau > 1$ compresses the gaps between logits and shifts probability mass toward the wrong classes. The distillation loss is the KL divergence between these softened distributions, multiplied by τ^2:

$$\mathcal{L}_{\text{distill}} = \tau^2 \, D_{\text{KL}}\big(\mathbf{p}^{(\tau)} \,\|\, \mathbf{q}^{(\tau)}\big). \tag{37.9}$$

This loss vanishes if and only if the student matches the softened teacher. To see why the factor τ^2 is needed, let us differentiate the unscaled KL term with respect to a student logit. Only the cross-entropy part $-\sum_j p_j^{(\tau)} \log q_j^{(\tau)}$ depends on the student, and the derivative of a log-softmax at temperature τ is the familiar softmax-minus-target expression scaled by $1/\tau$:

$$\frac{\partial}{\partial z_j^{\text{stu}}} D_{\text{KL}}(\mathbf{p}^{(\tau)} \| \mathbf{q}^{(\tau)}) = \frac{1}{\tau}(q_j^{(\tau)} - p_j^{(\tau)}).$$

Each student logit is thus nudged by the difference between the student's and teacher's softened probabilities for that class, divided by τ.

At high temperatures, that probability difference shrinks too. Subtracting the mean logit $\bar{z}$ leaves the softmax unchanged, and for large τ we can expand $e^{(z_j - \bar{z})/\tau} \approx 1 + (z_j - \bar{z})/\tau$ to first order. Because the centered deviations $z_j - \bar{z}$ sum to zero, the softmax denominator is approximately k, giving $\text{softmax}(\mathbf{z}/\tau)_j \approx \frac{1}{k}(1 + (z_j - \bar{z})/\tau)$. Consequently, $q_j^{(\tau)} - p_j^{(\tau)}$ shrinks like $1/\tau$, and the unscaled gradient shrinks like $1/\tau^2$:

$$\frac{1}{\tau}(q_j^{(\tau)} - p_j^{(\tau)}) \;\approx\; \frac{1}{k\,\tau^2}\Big((z_j^{\text{stu}} - \bar{z}^{\text{stu}}) - (z_j^{\text{tea}} - \bar{z}^{\text{tea}})\Big).$$

Multiplying the loss by τ^2 cancels this decay and keeps the gradient magnitude roughly constant as τ varies. That stability is essential in practice because the distillation loss is rarely used in isolation: it is typically added, with a fixed weight, to the standard cross-entropy loss on the hard labels. Without the τ^2 factor, tuning τ would inadvertently change the relative weight of the two losses. With it, the temperature controls only how much of the teacher's small wrong-class probabilities the student sees.

Let us verify this scaling numerically using teacher logits $(5, 2, -1)$ and student logits $(3, 3, 0)$. Both vectors have mean 2, so τ^2 times the gradient should converge to $\frac{1}{3}((1, 1, -2) - (3, 0, -3)) = (-0.667, 0.333, 0.333)$, which has norm 0.816. In the cell below, we use automatic differentiation to compute the gradient of the unscaled KL term across five temperatures and compare its norm before and after multiplying by τ^2.

```
z_tea = torch.tensor([5.0, 2.0, -1.0])
z_stu = torch.tensor([3.0, 3.0, 0.0], requires_grad=True)
print(' tau  |grad|  |tau^2 grad|  tau^2 grad')
for tau in (1.0, 2.0, 5.0, 10.0, 20.0):
    p_tau = torch.softmax(z_tea / tau, 0)
    kl = (p_tau * (torch.log(p_tau) - torch.log_softmax(z_stu / tau, 0))).sum()
    g = torch.autograd.grad(kl, z_stu)[0].numpy()
    print(f'{tau:4.0f}  {np.linalg.norm(g):.4f}  '
          f'{np.linalg.norm(tau**2 * g):12.3f}  {np.round(tau**2 * g, 3)}')
limit = (z_stu - z_stu.mean()) - (z_tea - z_tea.mean())
limit = limit.detach().numpy() / len(limit)
print(f'limit {"":6s}  {np.linalg.norm(limit):12.3f}  {np.round(limit, 3)}')
```

```
 tau  |grad|  |tau^2 grad|  tau^2 grad
   1  0.6391         0.639  [-0.462  0.441  0.022]
   2  0.2190         0.876  [-0.672  0.549  0.123]
   5  0.0368         0.920  [-0.741  0.478  0.263]
  10  0.0088         0.884  [-0.719  0.413  0.306]
  20  0.0021         0.854  [-0.697  0.374  0.322]
limit                0.816  [-0.667  0.333  0.333]
```

Without the τ^2 factor, the gradient collapses: its norm drops from 0.64 at $\tau = 1$ to 0.0021 at $\tau = 20$, roughly three hundred times smaller, and the final doubling of τ divides it by about

four, just as $1/\tau^2$ predicts. With the factor included, the norm stays between 0.64 and 0.92. It slightly overshoots the asymptotic limit 0.816, peaks near $\tau = 5$, and then settles back slowly at a rate of order $1/\tau$ (still reading 0.854 at $\tau = 20$). Notice also that at $\tau = 1$ the loss reduces to the ordinary KL divergence, and for a one-hot teacher (whose entropy is zero) it becomes the ordinary cross-entropy loss, so standard supervised training is a special case.

37.1.4 Entropy and KL Divergence for Densities

So far every outcome has been discrete and countable. A regression target, by contrast, is a real number described by a probability density, so sums become integrals. The KL divergence survives this transition intact, whereas entropy does not: continuous entropy can become negative and depends on the units in which the variable is measured. Consequently, the negative log-likelihood loss of a continuous model can fall below zero, and its floor shifts whenever we rescale the data. What retains its absolute meaning is the excess above the floor, and with it the difference in loss between any two models evaluated on the same data. (On a first reading, you may skip these three subsections and resume at the final paragraph of the section.)

37.1.4.1 Differential Entropy

For a continuous random variable X with density p, replacing the sum in the discrete entropy with an integral gives the *differential entropy*:

$$h(X) = -\int p(x) \log p(x)\, dx. \tag{37.10}$$

This is the expected negative log-density, and density values are not probabilities: they carry units of $1/x$ and can easily exceed 1. We use a lowercase h to distinguish differential entropy from the discrete H, because the two behave quite differently.

The simplest example already shows why. The uniform density on $[0, a]$ equals $1/a$ throughout the interval, so $h = -\int_0^a \frac{1}{a} \log \frac{1}{a}\, dx = \log a$. When $a = \frac{1}{2}$, this gives $-\ln 2 \approx -0.693$ nats, a negative entropy that no discrete distribution can ever have. Worse yet, measuring the very same length in centimetres instead of metres multiplies a by 100 and adds $\ln 100 \approx 4.61$ nats to h, even though the underlying uncertainty has not changed at all.

The same kind of shift happens under any smooth change of variables. If $y = g(x)$ is invertible with derivative g', then the density of Y at $g(x)$ is $p_X(x)/|g'(x)|$, and therefore

$$h(Y) = h(X) + \mathbb{E}\big[\log |g'(X)|\big].$$

For a random vector $\mathbf{x} \in \mathbb{R}^n$, both the definition and this transformation rule keep the exact same form, with an integral over $\mathbb{R}^n$ and the Jacobian determinant $|\det \partial g / \partial \mathbf{x}|$ replacing $|g'|$. Stretching a variable increases its differential entropy, while compressing it drives the entropy down without bound. As we will derive in Section 37.1.4.3, a Gaussian with standard deviation σ has differential entropy $h = \frac{1}{2}\log(2\pi e \sigma^2)$, which turns negative as soon as $\sigma < (2\pi e)^{-1/2} \approx 0.24$.

Fortunately, the KL divergence does not share this sensitivity to units. If we apply the same change of variables to both the data density p and a model density q, both are divided by

the same Jacobian factor, which cancels inside the ratio p/q. The KL divergence is therefore invariant to invertible reparameterizations. For two uniform densities p_1 on $[0, a_1]$ and p_2 on $[0, a_2]$ with $a_1 \le a_2$, for example, $\int p_1 \log(p_1/p_2)\, dx = \log(a_2/a_1)$: for $a_1 = \frac{1}{2}$ and $a_2 = 1$, this is $\ln 2 \approx 0.693$ nats whether measured in metres or in centimetres. The cross-entropy $-\mathbb{E}_P[\log q]$ used to train a continuous model shifts by the same constant as h, and since that constant does not depend on the model, both the gradients and the minimizer remain unchanged. Whenever $h(p)$ is finite, the decomposition Equation 37.5 still holds with h in place of H, so maximum likelihood for densities minimizes $D_{\mathrm{KL}}(p\|q_\theta)$ at the population level.

Note also that differential entropy is not simply the limit of discrete entropy under fine binning. Partition the real line into cells of width Δ, and let X_Δ record which cell contains X. For a point x_i inside cell i, $P(X_\Delta = i) \approx p(x_i)\Delta$, and

$$H(X_\Delta) + \log \Delta \to h(X) \quad \text{as } \Delta \to 0$$

whenever p is regular enough for the Riemann sums to converge to the integrals (for instance, when p is continuous and both p and $p \log p$ are Riemann-integrable). The discrete entropy $H(X_\Delta)$ itself diverges like $\log(1/\Delta)$, because a finer grid always demands more yes/no questions, and $h(X)$ is the finite offset that remains after subtracting that growth. For the uniform density on $[0, 1]$ with $\Delta = \frac{1}{8}$, all eight cells are equally likely, so $H_2(X_\Delta) = 3$ bits and $\log_2 \Delta = -3$, which sum to $0 = h$. Exercise 6 proves this limit and tests it numerically on a Gaussian.

37.1.4.2 KL Divergence Between Gaussians

When a regression model predicts a real number along with its uncertainty, it most often outputs a Gaussian: a mean and a variance. Comparing such a model with the data, or comparing two Gaussian models against each other, requires the KL divergence between two Gaussians, which has a clean closed form. For $P = \mathcal{N}(\mu_1, \sigma_1^2)$ and $Q = \mathcal{N}(\mu_2, \sigma_2^2)$,

$$D_{\mathrm{KL}}(\mathcal{N}(\mu_1, \sigma_1^2) \,\|\, \mathcal{N}(\mu_2, \sigma_2^2)) = \log \frac{\sigma_2}{\sigma_1} + \frac{\sigma_1^2 + (\mu_1 - \mu_2)^2}{2\sigma_2^2} - \frac{1}{2}. \tag{37.11}$$

The formula combines a log ratio of standard deviations with the variance and squared mean error of P, both measured relative to the model's variance σ_2^2. As Gibbs' inequality guarantees, it is non-negative and vanishes only when $\mu_1 = \mu_2$ and $\sigma_1 = \sigma_2$. It is also visibly asymmetric, because σ_1 and σ_2 enter in different ways. Exercise 7 derives this expression directly from the definition.

Start with the simplest case: a unit mean shift with equal variances, $P = \mathcal{N}(0, 1)$ and $Q = \mathcal{N}(1, 1)$. Here the formula reduces to $(\mu_1 - \mu_2)^2/(2\sigma^2) = \frac{1}{2}$ nat in either direction. Now let the variances differ instead: take $P = \mathcal{N}(0, 1)$ and $Q = \mathcal{N}(0, 4)$, so $\sigma_2 = 2$. Plugging in gives $D_{\mathrm{KL}}(P\|Q) = \ln 2 + \frac{1}{8} - \frac{1}{2} \approx 0.318$ nats, whereas the reverse direction gives $D_{\mathrm{KL}}(Q\|P) = -\ln 2 + 2 - \frac{1}{2} \approx 0.807$ nats, more than 2.5 times larger.

Figure 37.4 reveals where each divergence comes from. In both directions, the integrand changes sign at $|x| = (8 \ln 2/3)^{1/2} \approx 1.36$, where the two densities cross. In panel (a), the wide model explains narrow data: it is too surprised in the core ($|x| < 1.36$, contributing $+0.424$) and slightly less surprised than the truth in the tails (contributing -0.106). In

panel (b), the narrow model must explain wide data: the core contributes -0.240, while the tails, where the narrow model assigns tiny probability to outcomes that actually occur, contribute a hefty $+1.046$.

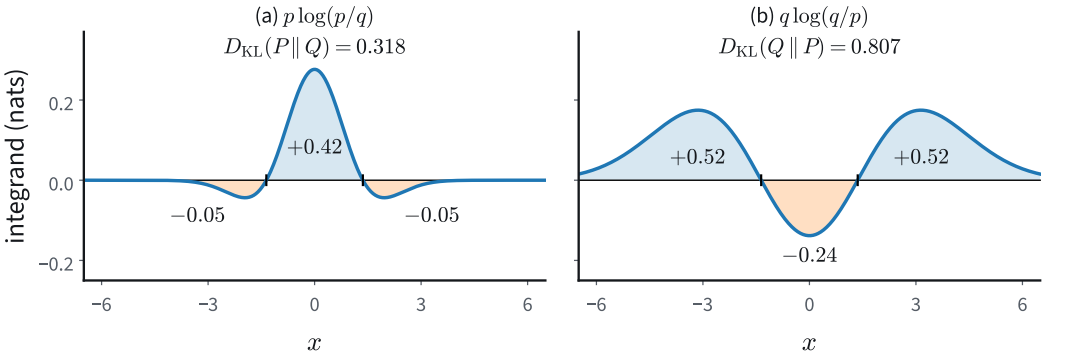


Figure 37.4: Where the KL divergence between $\mathcal{N}(0,1)$ and $\mathcal{N}(0,4)$ comes from. (a) The narrow distribution explained by the wide model: a positive core and slightly negative tails sum to 0.318 nats. (b) The wide distribution explained by the narrow model: the tails, where the narrow model assigns little probability, contribute 1.05 nats against a core of -0.24, for a total of 0.807. The sign changes at $|x| = 1.36$ in both directions.

A narrow model is far more surprised by wide data than a wide model is by narrow data. In general, KL divergence penalizes a model severely whenever outcomes occur that it had nearly ruled out, but stays mild when the model merely spreads extra probability over regions where no data appear. What this asymmetry means for fitting generative models in one direction versus the other is the subject of Section 37.3.2.

The definition also suggests a general way to estimate a KL divergence when no closed form exists: draw samples from P and average $\log p(x) - \log q(x)$. Because we will rely on sample-based estimates later, it is worth checking how accurate they are on this Gaussian pair. The next cell implements both the closed form and the Monte Carlo estimator together with its standard error.

```
def gaussian_kl(mu1, sigma1, mu2, sigma2):
    """Closed-form KL(N(mu1, sigma1^2) || N(mu2, sigma2^2)), in nats."""
    return (math.log(sigma2 / sigma1)
            + (sigma1 ** 2 + (mu1 - mu2) ** 2) / (2 * sigma2 ** 2)
            - 0.5)

def gaussian_log_pdf(x, mu, sigma):
    """Log-density of N(mu, sigma^2) at x."""
    return (-0.5 * math.log(2 * math.pi * sigma ** 2)
            - (x - mu) ** 2 / (2 * sigma ** 2))

def mc_kl(mu1, sigma1, mu2, sigma2, n=200000):
    """Monte Carlo estimate of the same KL from n samples x ~ P, with its
    standard error."""
    x = np.random.default_rng().normal(loc=mu1, scale=sigma1, size=n)
    r = gaussian_log_pdf(x, mu1, sigma1) - gaussian_log_pdf(x, mu2, sigma2)
```

```
    return float(r.mean()), float(r.std() / math.sqrt(n))
```

For the mean shift, we can predict the estimator's accuracy in advance. The log-ratio is $\log p(x) - \log q(x) = \frac{1}{2} - x$, which has variance 1 under P, so the average of $n = 2 \cdot 10^5$ samples has standard error $1/\sqrt{n} = 0.0022$.

For the pair with different variances, however, the two directions are not equally easy to estimate. Up to sign, both average the same function, $\ln 2 - \frac{3}{8}x^2$, but one samples from the narrow Gaussian while the other samples from the wide one, where x^2 fluctuates four times as much. Let us estimate all three divergences and check.

```
pairs = {'N(0,1) || N(1,1)': (0.0, 1.0, 1.0, 1.0),
         'N(0,1) || N(0,4)': (0.0, 1.0, 0.0, 2.0),
         'N(0,4) || N(0,1)': (0.0, 2.0, 0.0, 1.0)}
print(f'{"P || Q":18s} {"closed":>7s} {"sampled":>8s} {"std err":>8s}')
for name, args in pairs.items():
    estimate, std_err = mc_kl(*args)
    print(f'{name:18s} {gaussian_kl(*args):7.4f} {estimate:8.4f} {std_err:8.4f}')
```

```
P || Q             closed  sampled  std err
N(0,1) || N(1,1)   0.5000   0.5010   0.0022
N(0,1) || N(0,4)   0.3181   0.3169   0.0012
N(0,4) || N(0,1)   0.8069   0.8017   0.0047
```

The standard errors come out to 0.0022, 0.0012, and 0.0047, and every estimate lands within a few standard errors of the closed form. Notice that the direction sampling from the wide Gaussian is four times noisier than its reverse, meaning it requires sixteen times as many samples to reach the same accuracy. Panel (b) of Figure 37.4 explains why: most of this divergence comes from the tails, where the log-ratio is large and samples are sparse. We will encounter this same sampling difficulty at a much larger scale in Section 37.7.

37.1.4.3 Maximum Entropy and the Gaussian

On a finite set, the uniform distribution maximizes uncertainty. Which density on the real line plays the same role? Without any constraint, none does: no density can be uniform on all of $\mathbb{R}$, and stretching a density raises its differential entropy without limit. Suppose instead that we fix the two moments a Gaussian model reports, the mean μ and variance σ^2. Among all densities with those two moments, the Gaussian has the largest differential entropy. In this precise sense, a Gaussian noise model makes the fewest assumptions: it commits to a mean and a variance, and to nothing else.

The proof mirrors our one-line argument for the uniform bound, with a single twist. In the discrete case, $-\log u(x) = \log k$ was constant across all outcomes, so every distribution P had the same cross-entropy $\log k$ to the uniform model. Here, let φ denote the Gaussian density with mean μ and variance σ^2. Its surprisal $-\log \varphi(x) = \frac{1}{2}\log(2\pi\sigma^2) + \frac{(x-\mu)^2}{2\sigma^2}$ is no longer constant, but it is a quadratic function of x, and the expectation of a quadratic depends on P solely through the two moments we fixed. For every density P with mean μ and variance σ^2,

$$\mathrm{CE}(P,\varphi) = \tfrac{1}{2}\log(2\pi\sigma^2) + \frac{\mathbb{E}_P\left[(x-\mu)^2\right]}{2\sigma^2} = \tfrac{1}{2}\log(2\pi\sigma^2) + \tfrac{1}{2} = \tfrac{1}{2}\log(2\pi e\sigma^2).$$

Remarkably, the cross-entropy to φ does not depend on which such P generated the data. Setting $P = \varphi$ reduces cross-entropy to entropy, so this constant is the Gaussian's own differential entropy, the closed form we quoted earlier. For any other density P with the same moments, the decomposition Equation 37.5 (with h in place of H) and Gibbs' inequality yield

$$h(P) = \mathrm{CE}(P,\varphi) - D_{\mathrm{KL}}(P\|\varphi) \le \tfrac{1}{2}\log(2\pi e\sigma^2) = h(\mathcal{N}(\mu,\sigma^2)), \tag{37.12}$$

with equality if and only if $P = \mathcal{N}(\mu,\sigma^2)$. (If $h(P) = -\infty$, the inequality holds trivially.) The gap in this bound, $D_{\mathrm{KL}}(P\|\varphi)$, also has a direct interpretation: it is the excess loss incurred by the best possible Gaussian model when the data are not Gaussian (Exercise 8).

The uniform and Gaussian distributions are two instances of a single principle. Whenever a candidate density's surprisal averages to the same constant under every distribution satisfying the constraints, that candidate maximizes entropy among all of them. Section 36.4.2 establishes this result in full generality for constraints on the expectations of arbitrary statistics, where the maximizing distributions form an exponential family. The geometric distribution on the non-negative integers $\{0, 1, 2, \ldots\}$ is a third example: its surprisal is affine in the outcome, so it maximizes entropy among all distributions on the non-negative integers with a given mean.

Let us step back and see what remains open. Earlier, the discrete model Q' assigned zero probability to an outcome that actually occurred, making its KL divergence infinite; the narrow Gaussian of Figure 37.4 came close, accumulating most of its divergence in tails it had nearly ruled out. A generative model that maps a few random numbers into an image places all of its probability mass on a thin surface inside pixel space. If real data do not lie on that exact surface, the KL divergence is infinite regardless of whether the surface passes right next to the data or miles away. Section 37.3 shows that every divergence built from the density ratio p/q shares this blindness, and Section 37.4 develops comparison measures that can still tell near from far. A second question lies even closer at hand. Earlier we counted yes/no questions and read them as binary codewords, but only for a dyadic source whose surprisals were whole numbers of bits. Section 37.2 constructs codes for arbitrary sources and asks how close a real compressor can get to the entropy.

37.1.5 Summary

A cross-entropy loss measures a model's average surprisal on the data. It decomposes into the entropy of the data, which no model can lower, plus the KL divergence from the data to the model, which by Gibbs' inequality is never negative and vanishes only for the true distribution. Any loss value must therefore be judged against its irreducible floor, the conditional entropy of the label given the input, which on fresh data only the true conditional probabilities can reach. That floor is zero when the input uniquely determines the label, and $H_b(\rho) + \rho\log(k-1)$ under symmetric label noise at rate ρ.

In bits, the entropy of a finite source equals the average number of yes/no questions needed to identify an outcome (to within one question) and is at most $\log_2 k$ for k outcomes. Softening the targets shifts the optimum: label smoothing creates a finite optimal logit gap with a

positive loss floor, while knowledge distillation matches a softened teacher, with the factor τ^2 keeping the gradient scale steady as the temperature changes. For continuous densities, differential entropy can be negative and shifts with the units of measurement, whereas both the KL divergence and the minimizer of the cross-entropy remain invariant.

37.1.6 Exercises

1. **Uniqueness of self-information.** ● Show that self-information is unique up to scale: if a strictly decreasing function $I(p)$ on $(0, 1]$ satisfies $I(1) = 0$ and $I(p_1 p_2) = I(p_1) + I(p_2)$ for all $p_1, p_2 \in (0, 1]$, then $I(p) = -c \log p$ for some constant $c > 0$. Work with the function $g(t) = I(e^{-t})$ for $t \geq 0$.

2. code **The label-noise floor.** Labels on k classes are correct with probability $1 - \rho$ and otherwise uniform over the other $k - 1$ classes, independently of the input.

    1. Assume that the clean label is a function of the input. Show that the population cross-entropy of any classifier is at least $H_b(\rho) + \rho \log(k - 1)$, that the optimal prediction is the noisy conditional distribution itself, and that the optimal accuracy is $1 - \rho$ whenever $1 - \rho > \rho/(k - 1)$.
    2. Evaluate the floor for $k = 10$ and $\rho = 0.05, 0.1, 0.2$.
    3. A colleague reports a validation loss of 0.40 nats on a 10-class data set known to have 10% such noise in every split. List possible explanations, and say which of them the floor rules out and which it does not.
    4. Train the softmax regression of Section 3.3 on Fashion-MNIST with 10% of the training and test labels flipped in this way. Compare the test loss with 0.545 and the test accuracy with 90%. The gap is large. Which assumption of (a) does Fashion-MNIST violate, what does the linear model's own error add, and how could you estimate the floor that does apply?

3. **Other scoring rules.** A scoring rule $S(\mathbf{q}, y)$ is strictly proper if the expected penalty $\mathbb{E}_{y \sim \mathbf{p}}[S(\mathbf{q}, y)]$ is uniquely minimized at $\mathbf{q} = \mathbf{p}$. For the log score, this property is Gibbs' inequality.

    1. Show that the expected Brier score, with $S(\mathbf{q}, y) = \sum_j (q_j - \mathbf{1}(j = y))^2$, exceeds the expected Brier score of the truthful report by $\|\mathbf{q} - \mathbf{p}\|^2$, so that the Brier score is strictly proper (Gneiting and Raftery 2007).
    2. The linear score $S(\mathbf{q}, y) = -q_y$ rewards the probability placed on the observed class. Which report minimizes its expected penalty? Is the linear score proper? What does a classifier trained with it report for an input whose label is class 1 with probability 0.6 and class 2 with probability 0.4?

4. **Label smoothing on ambiguous inputs.** ● Label smoothing trains with the target Equation 37.8.

    1. In the training distribution, an input $\mathbf{x}$ carries the label y with probability $p(y \mid \mathbf{x})$. Show that the expected smoothed loss for this input is minimized by $\mathbf{q}^*(\cdot \mid \mathbf{x}) = (1 - \epsilon)\, p(\cdot \mid \mathbf{x}) + \frac{\epsilon}{k}\mathbf{1}$. How can $p(\cdot \mid \mathbf{x})$ be recovered from $\mathbf{q}^*$? What is the largest probability that the minimizer can assign to a class?
    2. For one example with logits $\mathbf{z}$ and $\mathbf{q} = \mathrm{softmax}(\mathbf{z})$, the gradient of $\mathrm{CE}(\mathbf{p}^\epsilon, \mathbf{q})$ with respect to z_j is $q_j - p_j^\epsilon$. For $\epsilon > 0$, show that all wrong-class logits are equal at every stationary point.
    3. With one-hot targets ($\epsilon = 0$), the gradient with respect to a wrong-class logit is q_j, and the loss has no stationary point. Show that gradient descent on the logits

of one example still shrinks the gap between two wrong-class logits, and explain why this shrinkage stalls as $q_y \to 1$. In a network, the logits of all examples are computed from shared features. Which differences between wrong-class logits can therefore survive training with $\epsilon = 0$? Why does training with $\epsilon > 0$ remove them?

4. Müller et al. (2019) observe that teachers trained with label smoothing produce worse students under the distillation loss Equation 37.9. Explain this observation with the results of (b) and (c).

5. code **Distillation at high temperature.** ● Hinton et al. (2015) state in their Section 2.1 that, at high temperature, distillation reduces to matching zero-mean logits by squared error.

   1. Write $q_j = p_j(1 + u_j)$, so that $\sum_j p_j u_j = 0$, and expand $D_{\mathrm{KL}}(\mathbf{p}\|\mathbf{q}) = -\sum_j p_j \log(1 + u_j)$ to second order to obtain

$$D_{\mathrm{KL}}(\mathbf{p}\|\mathbf{q}) = \frac{1}{2}\sum_j \frac{(q_j - p_j)^2}{p_j} + O(\|\mathbf{q} - \mathbf{p}\|^3).$$

   Combine this expansion with the large-τ approximation of $\mathrm{softmax}(\mathbf{z}/\tau)$ in Section 37.1.3.4 to show that, as $\tau \to \infty$,

$$\tau^2 D_{\mathrm{KL}}(\mathbf{p}^{(\tau)} \,\|\, \mathbf{q}^{(\tau)}) \to \frac{1}{2k}\sum_{j=1}^{k}\Big((z_j^{\mathrm{stu}} - \bar{z}^{\mathrm{stu}}) - (z_j^{\mathrm{tea}} - \bar{z}^{\mathrm{tea}})\Big)^2.$$

   2. Evaluate $\tau^2 D_{\mathrm{KL}}(\mathbf{p}^{(\tau)}\|\mathbf{q}^{(\tau)})$ for $\tau = 1, 2, 5, 10, 100$ with the logits $\texttt{z_tea} = (5, 2, -1)$ and $\texttt{z_stu} = (3, 3, 0)$ of Section 37.1.3.4, and compare it with the limit.

   3. At low temperature, which teacher logits have almost no influence on the loss? Hinton et al. (2015) argue that this can be an advantage and also a drawback. Give one argument for each side, based on what the teacher's own cross-entropy training does to these logits.

6. code **Quantized entropy against differential entropy.** Let X have density p, and let X_Δ record which cell $[i\Delta, (i+1)\Delta)$ contains X.

   1. Show that $H(X_\Delta) + \log\Delta \to h(X)$ as $\Delta \to 0$ when p is continuous and p and $p\log p$ are Riemann-integrable. Write $P(X_\Delta = i) = p(x_i)\Delta$ for some x_i in the cell, by the mean value theorem for integrals.
   2. Check the limit by hand for the uniform density on $[0, 1]$ with $\Delta = \frac{1}{8}$. What happens for a cell width that does not divide 1, such as $\Delta = 0.3$?
   3. Compute $H(X_\Delta) + \log\Delta$ numerically for $X \sim \mathcal{N}(0, 1)$ with $\Delta = 0.5, 0.1, 0.01$, using the Gaussian distribution function for the cell probabilities, and compare it with $h = \frac{1}{2}\log(2\pi e) \approx 1.4189$ nats. (At $\Delta = 0.5$ you should find 1.4292.)
   4. Why does $H(X_\Delta)$ itself diverge as $\Delta \to 0$? A regression target is binned into cells of width Δ and predicted with a softmax classifier. Which quantity does its loss approach at best, and what must be added to it before it can be compared with the negative log-density of a continuous model?

7. **Gaussian KL divergence.** Let $P = \mathcal{N}(\mu_1, \sigma_1^2)$.

1. Derive the closed form Equation 37.11 by writing out $\mathbb{E}_{x\sim P}[\log p(x) - \log q(x)]$ and using $\mathbb{E}_{x\sim P}[x] = \mu_1$ and $\mathbb{E}_{x\sim P}[(x-\mu_1)^2] = \sigma_1^2$.
2. Let Q have mean $\mu_1 + \delta$ and standard deviation $\sigma_1 e^{\eta}$. Expand both $D_{\mathrm{KL}}(P\|Q)$ and $D_{\mathrm{KL}}(Q\|P)$ to second order in δ and η. At which order do the two directions first differ?
3. Evaluate the second-order approximation for $P = \mathcal{N}(0,1)$ and $Q = \mathcal{N}(0,4)$, and compare it with the exact values 0.318 and 0.807 nats of the two directions.

8. **The excess loss of a Gaussian model.** A regression model reports a Gaussian $\mathcal{N}(m, s^2)$ for a target whose density P has mean μ, variance σ^2 and a finite differential entropy.

   1. Show that $\mathrm{CE}(P, \mathcal{N}(m, s^2))$ is smallest at $m = \mu$ and $s^2 = \sigma^2$, and that this best Gaussian model exceeds the floor $h(P)$ by $D_{\mathrm{KL}}(P\|\varphi) = h(\varphi) - h(P)$, where φ is the density of $\mathcal{N}(\mu, \sigma^2)$.
   2. Evaluate the excess for the uniform density on $[0, a]$, which has variance $a^2/12$, and for the Laplace density $\frac{1}{2b}e^{-|x|/b}$, which has variance $2b^2$ and differential entropy $1 + \log(2b)$. (You should find about 0.176 and 0.072 nats.)
   3. Neither answer depends on the scale a or b. Explain why without computing, from what Section 37.1.4.1 says about a change of units.
   4. The Gaussian has the largest entropy for a given variance, so no density has a higher floor. Is there also a largest excess? Consider densities that concentrate near the two points $\mu \pm \sigma$. What does your answer say about a Gaussian model for a target with two well-separated modes?

37.2 Codes, Compression, and Perplexity

In Section 37.1, we interpreted entropy and cross-entropy as average codeword lengths, suggesting that a model reaching a loss of x bits per symbol should be able to compress a text to roughly x bits per symbol. Yet we only verified that claim for one convenient source whose surprisals happened to be whole numbers of bits; we never constructed a code for any other distribution. Until we actually build such a code, the connection between log-loss and file size remains an analogy. This section closes that gap by asking two questions: can we construct a code whose length matches a model's log-loss, and how close do practical compressors come to that limit?

Everyday compressors already hint at the answer. Take *The Time Machine*, the novel from Section 8.2, cleaned down to 173,428 characters over a 27-symbol alphabet (the 26 lowercase letters plus space). The general-purpose compressor `zlib`, which replaces repeated strings with back-references and then codes symbols individually, stores the text in 2.84 bits per character. By contrast, a tiny counting model that predicts each character from the three characters before it, updating its counts as it reads, compresses the same text to 2.55 bits per character. Better prediction yields a shorter file, and this section explains why. Once a code turns log-loss into file size, a loss such as the 1.2 nats per character from the chapter introduction becomes a concrete storage rate, and comparing two models becomes as direct as comparing the lengths of the files they produce.

We build toward that result in four steps. First, we ask how short codewords can be when a receiver must parse a binary stream with no separators between symbols. Second, we measure the penalty of forcing every codeword to use a whole number of bits, and we build a stream code that avoids paying that penalty on every symbol. Third, we turn to real text,

where each character depends on the past, to see how short a file can possibly get and how the cost of learning a model enters the total length. Finally, we connect these code lengths to perplexity, the metric language models report on held-out text. Throughout the section we work in bits rather than nats because codewords are binary strings; recall that x bits equal $x \ln 2$ nats.

Alongside the book's standard imports, we need `re` for text preprocessing and `zlib` as our baseline compressor.

```
import re
import zlib
```

37.2.1 Symbol Codes

Let us start with the simplest communication problem. A sender observes a stream of symbols drawn independently from a distribution P over k outcomes and must transmit them as a single binary string with no separators between symbols, while the receiver must recover the original sequence unambiguously.

37.2.1.1 Prefix Codes and the Kraft Inequality

In a symbol code, each outcome i is assigned a binary string, its *codeword*, of length ℓ_i. How can the receiver tell where one codeword ends and the next begins without a delimiter? The simplest guarantee is that no codeword is a prefix of any other; such a code is called *prefix-free*, or simply a *prefix code*. Telephone numbers follow the same rule: once the digits you have dialed match a valid number, no longer number starts with that same sequence, so the exchange immediately knows the number is complete. For four symbols, the code $0, 10, 110, 111$ is prefix-free, and the incoming stream `110010111` can be parsed in only one way, as `110|0|10|111`.

Visualizing a prefix code as a binary tree (Figure 37.5) immediately reveals the constraint on codeword lengths. A codeword of length ℓ ends at a leaf of depth ℓ and rules out every descendant below it, claiming a fraction $2^{-\ell}$ of the full tree. Because different codewords cannot overlap, the fractions they claim must fit inside a single tree, a total budget of 1, which makes short codewords expensive. The lengths $(1, 2, 3, 3)$ in the figure spend $\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{8} = 1$, using up the entire tree. By contrast, four codewords of length 1 would require a budget of 2, which is impossible since only two binary strings of length 1 exist. The lengths $(1, 2, 4)$ spend $\frac{1}{2} + \frac{1}{4} + \frac{1}{16} = 0.8125$ and fit comfortably with room to spare, for example as $0, 10, 1100$.

The same tree also describes a strategy for the yes/no questions game of Section 37.1. Each internal node poses a binary question, and each codeword records the sequence of answers leading to a symbol. Taking 0 for yes, the code $0, 10, 110, 111$ asks "Is it the first symbol?", then "Is it the second?", then "Is it the third?". The previous section used 1 for yes and obtained the mirror image, $1, 01, 001, 000$.

Proposition (Kraft inequality). *A prefix-free binary code with codeword lengths $\ell_1, \dots, \ell_k$ exists if and only if*

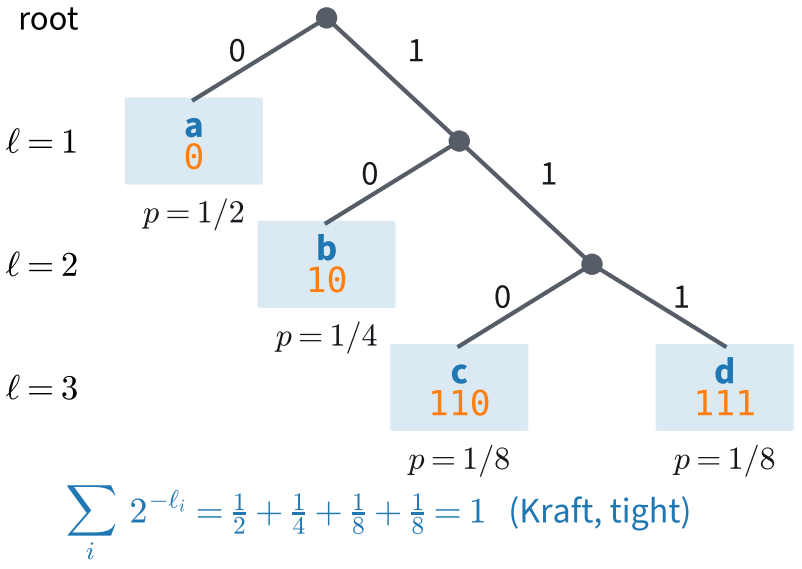


Figure 37.5: The prefix-free code $0, 10, 110, 111$ as a binary tree. Each codeword is the path from the root to a leaf (left edge 0, right edge 1), and no codeword passes through another. A leaf at depth ℓ claims a fraction $2^{-\ell}$ of the tree; here the four leaves exhaust the entire budget, $2^{-1} + 2^{-2} + 2^{-3} + 2^{-3} = 1$. The symbols a, b, c, d and their probabilities belong to the dyadic source from the previous section.

$$\sum_{i=1}^{k} 2^{-\ell_i} \le 1. \tag{37.13}$$

Proof. Associate each binary codeword c of length ℓ with the half-open interval $[0.c,\ 0.c + 2^{-\ell})$, consisting of all real numbers in $[0, 1)$ whose binary expansion begins with c. For instance, the codeword 10 corresponds to $[0.10_2, 0.11_2) = [\frac{1}{2}, \frac{3}{4})$, which has width 2^{-2}. In Figure 37.5, this interval represents the leaf for b together with the entire subtree removed beneath it. Any two such *dyadic* intervals are either nested or disjoint: if c is a prefix of c', every number starting with c' also starts with c, so the intervals are nested; otherwise c and c' disagree at some position within the shorter length, so no number can start with both and the intervals are disjoint. A prefix-free code therefore corresponds to disjoint intervals inside $[0, 1)$, whose total width $\sum_i 2^{-\ell_i}$ can be at most 1.

Conversely, suppose the lengths satisfy Equation 37.13. Sort them in nondecreasing order, $\ell_1 \le \cdots \le \ell_k$, and place intervals of width $2^{-\ell_i}$ side by side starting from 0. The i-th interval begins at $s_i = \sum_{m<i} 2^{-\ell_m}$ and ends at $s_i + 2^{-\ell_i} \le 1$. Because every preceding length satisfies $\ell_m \le \ell_i$, every term $2^{-\ell_m}$ in s_i is an integer multiple of $2^{-\ell_i}$, and hence s_i itself is an integer multiple of $2^{-\ell_i}$. The i-th interval is therefore the dyadic interval of the codeword given by the first ℓ_i binary digits of s_i. Since these intervals are disjoint, the first part of the proof guarantees that no codeword is a prefix of another. $\blacksquare$

Notably, Kraft's budget is not an artifact of insisting on prefix-free codes. A code is called *uniquely decodable* if every concatenation of codewords parses in only one way, even if some codewords are prefixes of others. McMillan proved that every uniquely decodable code must still satisfy Equation 37.13 (Exercise 1). Thus any set of codeword lengths achievable by a uniquely decodable code is already achievable by a prefix code, so restricting our attention to prefix codes loses nothing.

37.2.1.2 Shannon Lengths and the Entropy Bound

Knowing the budget, we can now ask which codeword lengths minimize the expected length $\mathbb{E}[\ell] = \sum_i p_i \ell_i$. Frequent symbols clearly deserve shorter codewords, yet assigning a length-1 codeword already consumes half of the entire Kraft budget.

To find the ideal balance, let us temporarily relax the integer constraint and treat the lengths ℓ_i as real numbers. Leaving budget unspent can never help, since shortening any codeword strictly reduces $\mathbb{E}[\ell]$, so the optimum must satisfy $\sum_i 2^{-\ell_i} = 1$. By Lagrange's method for equality-constrained optimization (Section 34.2.4), the gradient of the objective must be proportional to the gradient of the constraint at the minimum. The derivative of $\mathbb{E}[\ell]$ with respect to ℓ_i is p_i, while the derivative of $\sum_i 2^{-\ell_i}$ is $-2^{-\ell_i} \ln 2$. Hence $2^{-\ell_i} = c\, p_i$ for a single constant c across all i. Summing both sides over i forces $c = 1$, giving the optimal real lengths and the resulting expected length:

$$\ell_i = \log_2 \frac{1}{p_i}, \qquad \mathbb{E}[\ell] = \sum_i p_i \log_2 \frac{1}{p_i} = H_2(P).$$

Each symbol's ideal length is simply its *surprisal* $\log_2(1/p_i)$, the self-information of Section 37.1 measured in bits, and the resulting expected length is the binary entropy $H_2(P)$. Of course, actual codewords must have integer lengths. For the dyadic source $P =$

$(\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{8})$, the surprisals $1, 2, 3, 3$ are already integers, so the code of Figure 37.5 achieves $\mathbb{E}[\ell] = 1.75$ bits $= H_2(P)$ with zero waste. For $P = (0.6, 0.3, 0.1)$, the first of this chapter's running pair of distributions, the surprisals are 0.74, 1.74, and 3.32 bits. Rounding each up to the next integer produces the *Shannon lengths* $1, 2, 4$, the same profile that fit the binary tree with room to spare. Its Kraft sum is 0.8125, and its expected length is $\mathbb{E}[\ell] = 0.6 + 0.6 + 0.4 = 1.6$ bits, compared with $H_2(P) = 1.295$.

Even with carefully chosen integer lengths, no prefix code can beat the entropy, and Shannon lengths never exceed it by more than one bit. Both bounds follow from a single identity from Section 37.1, the cross-entropy decomposition, restated here in bits (where the subscript 2 and superscript (2) denote base-2 logarithms). The expected log-loss of a model Q on data drawn from P splits into the entropy of P and the Kullback–Leibler divergence from P to Q:

$$\mathrm{CE}_2(P, Q) = -\sum_i p_i \log_2 q_i = H_2(P) + D_{\mathrm{KL}}^{(2)}(P\|Q), \qquad D_{\mathrm{KL}}^{(2)}(P\|Q) \geq 0. \tag{37.14}$$

Here the nonnegativity of the KL divergence is Gibbs' inequality: no model can achieve a lower expected log-loss than the true entropy of the source, and equality holds only when $Q = P$. For our running pair, $P = (0.6, 0.3, 0.1)$ and $Q = (0.2, 0.5, 0.3)$, the three quantities are $1.867 = 1.295 + 0.571$ bits (up to rounding in the last decimal place).

Proposition (entropy bounds the code length). *Let P be a distribution with $p_i > 0$ for all i (symbols of zero probability need no codeword). Every prefix-free binary code satisfies $\mathbb{E}[\ell] \geq H_2(P)$, with equality if and only if $p_i = 2^{-\ell_i}$ for every i. Conversely, the* Shannon code *with lengths $\ell_i = \lceil \log_2(1/p_i) \rceil$ is prefix-free and satisfies $\mathbb{E}[\ell] < H_2(P) + 1$.*

Proof. For the lower bound, we compare the code against the probability distribution it implicitly defines. Let $Z = \sum_i 2^{-\ell_i}$, which satisfies $Z \leq 1$ by the Kraft inequality, and define the normalized distribution $q_i = 2^{-\ell_i}/Z$. Solving for ℓ_i gives $\ell_i = -\log_2 q_i - \log_2 Z$, and taking the expectation under P yields

$$\mathbb{E}[\ell] = \sum_i p_i \ell_i = -\sum_i p_i \log_2 q_i - \log_2 Z = H_2(P) + D_{\mathrm{KL}}^{(2)}(P\|Q) - \log_2 Z \geq H_2(P). \tag{37.15}$$

Both excess terms are nonnegative: $D_{\mathrm{KL}}^{(2)}(P\|Q) \geq 0$ by Gibbs' inequality, and $-\log_2 Z \geq 0$ because $Z \leq 1$. Equality requires both to vanish, meaning $Q = P$ and $Z = 1$, or equivalently $p_i = 2^{-\ell_i}$ for all i. For the upper bound, the Shannon lengths satisfy $2^{-\ell_i} \leq p_i$, so $\sum_i 2^{-\ell_i} \leq \sum_i p_i = 1$; they therefore pass Kraft's test and define a valid prefix code. Since each ceiling adds strictly less than one bit, $\ell_i < \log_2(1/p_i) + 1$, and averaging under P gives $\mathbb{E}[\ell] < H_2(P) + 1$. $\blacksquare$

The identity Equation 37.15 is even more informative than the bound it establishes: it shows that a prefix code can waste bits in only two ways: by implying a distribution Q that differs from P, or by leaving part of its Kraft budget unspent ($Z < 1$). For our running distribution P, the Shannon code wastes $1.6 - 1.295 = 0.305$ bits per symbol, of which only 0.005 bits come from mismatch while 0.300 bits come from unspent budget.

Together, the two halves of the proposition sandwich the optimal expected length of any symbol code between $H_2(P)$ and $H_2(P) + 1$ bits. This is precisely the bound on the expected number of yes/no questions promised in Section 37.1. The entropy floor is attained only when every probability is an exact power of $\frac{1}{2}$.

How much of that one-bit overhead can a smarter symbol code recover? Huffman's algorithm (Huffman 1952) constructs an optimal prefix code by repeatedly merging the two least probable symbols into a single combined symbol (Exercise 2), and it is the symbol coder used inside `zlib`. On our running P, Huffman's algorithm chooses the lengths $1, 2, 2$ and averages 1.4 bits, just 0.1 bits above the entropy, eliminating two-thirds of the Shannon code's waste. Yet even an optimal symbol code faces an unavoidable wall: no codeword can be shorter than 1 bit, and on a highly skewed source that single-bit floor swallows almost the entire savings.

Consider a source that emits `a` with probability 0.9 and `b` with probability 0.1. Its entropy is 0.47 bits per symbol, and each `a` carries only $\log_2(1/0.9) = 0.15$ bits of surprisal. Still, every symbol code must spend at least one full bit on every `a`, more than six times its information content. If `a` occurs with probability 0.999, the entropy drops to 0.011 bits per symbol, yet any symbol code still spends 1 bit. That "$+1$" penalty is the cost of rounding once per symbol. Coding blocks of n symbols at a time amortizes it: treating an n-symbol block as a single symbol from a larger alphabet rounds up only once per block, driving the per-symbol overhead below $1/n$. Even coding *pairs* of `a`/`b` symbols with Huffman's algorithm immediately lowers the rate from 1 bit down to 0.645 bits per original symbol.

37.2.1.3 Mismatched Codes and Cross-Entropy

In practice, we rarely know the true data distribution P; instead, we build our code from a model Q. The Shannon code for Q assigns lengths $\ell_i = \lceil \log_2(1/q_i) \rceil$, which requires $q_i > 0$ for every symbol that P can emit. When this code is used on data actually generated by P, its expected length satisfies

$$\mathrm{CE}_2(P, Q) \;\le\; \mathbb{E}_P[\ell] \;<\; \mathrm{CE}_2(P, Q) + 1, \tag{37.16}$$

because $\log_2(1/q_i) \le \ell_i < \log_2(1/q_i) + 1$ holds for every symbol i, and averaging under P preserves both inequalities. Cross-entropy is therefore the expected length of a code designed for the model Q, up to a single bit of rounding, and by Equation 37.14 the KL divergence is the exact penalty paid for using Q instead of the true distribution P.

Let us verify these bounds numerically. The cell below constructs the Shannon code for a model `q`, evaluates its expected length under a data distribution `p`, and reports the Kraft sum alongside the cross-entropy. We test both sources from this section under matched and mismatched codes: the dyadic source paired with a uniform model, and our running P paired with $Q = (0.2, 0.5, 0.3)$.

```
def shannon_lengths(q):
    return [math.ceil(math.log2(1 / q_i)) for q_i in q]

def code_report(name, p, q):
    """The Shannon code built from q, used on data from p (all in bits)."""
    ells = shannon_lengths(q)
    kraft = sum(2 ** -ell for ell in ells)
```

```
        avg = sum(p_i * ell for p_i, ell in zip(p, ells))
        ce2 = -sum(p_i * math.log2(q_i) for p_i, q_i in zip(p, q))
        print(f'{name:<16} lengths {ells}  Kraft {kraft:.4f}  '
              f'E[ell] {avg:.3f}  CE2 {ce2:.3f}')

    dyadic, uniform = [1/2, 1/4, 1/8, 1/8], [1/4] * 4
    p, q = [0.6, 0.3, 0.1], [0.2, 0.5, 0.3]
    code_report('dyadic, own', dyadic, dyadic)
    code_report('dyadic, uniform', dyadic, uniform)
    code_report('P, own', p, p)
    code_report('P, code from Q', p, q)
```

```
dyadic, own      lengths [1, 2, 3, 3]  Kraft 1.0000  E[ell] 1.750  CE2 1.750
dyadic, uniform  lengths [2, 2, 2, 2]  Kraft 1.0000  E[ell] 2.000  CE2 2.000
P, own           lengths [1, 2, 4]  Kraft 0.8125  E[ell] 1.600  CE2 1.295
P, code from Q   lengths [3, 1, 2]  Kraft 0.8750  E[ell] 2.300  CE2 1.867
```

The two matched rows reproduce our hand calculations: here `CE2` equals the entropy, and the expected lengths are 1.75 and 1.6 bits. The mismatched rows reveal the cost of using the wrong model. Building a Shannon code from Q gives lengths $3, 1, 2$, inadvertently assigning the longest codeword to the most frequent symbol of P. On data from P, it averages 2.3 bits per symbol, landing squarely inside the interval $1.867 \leq 2.3 < 2.867$ predicted by Equation 37.16. The uniform code assigns two bits to every symbol of the dyadic source; nothing is rounded, so the lower bound holds with equality, and $2.0 = 1.75 + 0.25$ bits splits cleanly into the entropy plus a KL divergence of 0.25 bits. Finally, the Kraft column shows how much budget each code leaves unused: none for the dyadic source, 0.1875 for the Shannon code of P, and 0.125 for the code built from Q.

Notice one subtlety: the bound Equation 37.16 applies specifically to the Shannon code of Q, not to every code one might build from Q. Huffman's code for Q has lengths $2, 1, 2$, spends the full Kraft budget, and averages only $1.2 + 0.3 + 0.2 = 1.7 < 1.867$ bits on data from P. There is no contradiction: its implied distribution is $(\frac{1}{4}, \frac{1}{2}, \frac{1}{4})$ rather than Q, and that dyadic distribution happens to sit closer to P. As Equation 37.15 showed, a code's excess length depends only on the distribution it implies and its unspent budget; Q matters only through the codeword lengths it produces.

37.2.2 Stream Codes and Description Length

A symbol code pays a rounding penalty on every symbol. Next we build a code that rounds only once per message, connect it directly to a language model, and then ask how to account for the cost of describing the model itself.

37.2.2.1 Arithmetic Coding

Coding blocks of symbols eliminates the per-symbol rounding loss, but it runs straight into a combinatorial wall: a codebook for blocks of 100 binary symbols would require 2^{100} entries, far too many to store. What we really want is the efficiency of a block code without its lookup table: a single codeword for the entire message, computed on the fly from the message itself. Codes of this kind are called *stream codes*, and *arithmetic coding* (Rissanen 1976; Witten et al. 1987) is the classic construction. Rather than looking up a codeword, arithmetic coding represents the message as a subinterval of $[0, 1)$. Figure 37.6 illustrates the idea on the ten-

symbol message aaaaaaaaba, generated by our skewed source with $q(\mathtt{a}) = 0.9$ and encoded under the true model.

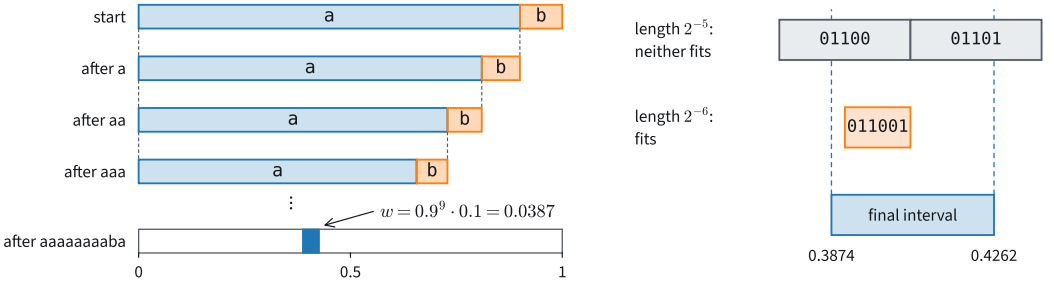


Figure 37.6: Arithmetic coding of the message aaaaaaaaba under the model $q(\mathtt{a}) = 0.9$. (a) Each symbol narrows the current interval to the subinterval allocated to it by the model; after ten symbols the remaining width is $w = 0.9^9 \cdot 0.1 = 0.0387$, the probability of the message. (b) The final interval is encoded by a dyadic interval lying wholly inside it. Neither dyadic interval of length 2^{-5} that overlaps it fits completely inside, whereas the interval of length 2^{-6} from 0.390625 to 0.40625 does. Its binary address 011001 is the codeword: six bits for a message whose surprisal is 4.69 bits.

The encoder maintains a current interval, initially $[0, 1)$. To encode the next symbol x_i, it partitions the current interval into one slice per possible symbol, with widths proportional to the model's conditional probabilities $q(\cdot \mid x_{<i})$, and keeps only the slice corresponding to the symbol that actually occurred. Each step multiplies the interval width by the probability of the observed symbol, so after processing $x_1, \dots, x_n$ the final interval has width

$$w = \prod_{i=1}^{n} q(x_i \mid x_{<i}),$$

which is exactly the probability the model assigns to the whole message. Because distinct messages of length n land in disjoint intervals, any point inside the final interval uniquely identifies the message: a decoder that replays the same model checks which slice contains the transmitted number, outputs the corresponding symbol, and narrows its own interval in lockstep. The decoder also needs to know when to stop, either by knowing n in advance or by including a dedicated end-of-message symbol, and we set that framing cost aside here. All that remains is to specify a point inside the final interval in binary, and the dyadic intervals from the Kraft proof do this cleanly.

Any interval of width w is guaranteed to contain a dyadic interval of length $2^{-\ell}$ once we choose $\ell = \lceil \log_2(1/w) \rceil + 1$. Why? Because this choice ensures $2^{-\ell} \le w/2$, so the first multiple of $2^{-\ell}$ at or above the interval's left endpoint falls within its left half, and the dyadic interval of length $2^{-\ell}$ starting there cannot overshoot the right endpoint. Panel (b) of Figure 37.6 shows both cases: the dyadic intervals of length 2^{-5} protrude past the edges of the final interval, whereas one of length 2^{-6} fits completely inside. Since $\ell < \log_2(1/w) + 2$, the resulting codeword length satisfies

$$\ell < \sum_{i=1}^{n} \log_2 \frac{1}{q(x_i \mid x_{<i})} + 2, \tag{37.17}$$

meaning the message is encoded in its total surprisal under the model plus at most two extra bits (for a known message length n). Rounding now happens once for the entire message rather than once per symbol. Moreover, codewords for different length-n messages identify dyadic intervals inside disjoint message intervals, so by the first half of the Kraft proof they form a prefix code over messages. The entropy lower bound therefore applies to whole messages just as it did to single symbols: no matter what model we choose, the expected code length can never fall below the entropy of the message.

Let us run this interval-narrowing procedure on `aaaaaaaaba` and compare its length against the symbol-by-symbol Shannon code. The cell below tracks the interval, extracts the binary codeword, verifies that its dyadic interval fits inside the final interval, and prints both bit counts.

```
msg = 'aaaaaaaaba'                    # nine a's, one b
q = {'a': 0.9, 'b': 0.1}              # an i.i.d. model of the source

lo, hi = 0.0, 1.0
for ch in msg:                        # keep the observed symbol's slice
    width = hi - lo
    if ch == 'a':                     # 'a' owns the left 90% of the interval
        hi = lo + width * q['a']
    else:                             # 'b' owns the right 10%
        lo = lo + width * q['a']

ell = math.ceil(-math.log2(hi - lo)) + 1      # codeword length
m = math.ceil(lo * 2 ** ell)                  # first multiple of 2^-ell >= lo
assert lo <= m / 2 ** ell and (m + 1) / 2 ** ell <= hi
surprisal = -math.log2(math.prod(q[ch] for ch in msg))
shannon_bits = sum(math.ceil(math.log2(1 / q[ch])) for ch in msg)
print(f'final interval [{lo:.4f}, {hi:.4f}), width {hi - lo:.4f}')
print(f'surprisal {surprisal:.2f} bits, codeword {m:0{ell}b} ({ell} bits)')
print(f'symbol-by-symbol Shannon code: {shannon_bits} bits')
```

```
final interval [0.3874, 0.4262), width 0.0387
surprisal 4.69 bits, codeword 011001 (6 bits)
symbol-by-symbol Shannon code: 13 bits
```

The final width 0.0387 matches the product $0.9^9 \cdot 0.1$ as expected. The message has 4.69 bits of surprisal, and arithmetic coding stores it in 6 bits (`011001`), whereas the symbol-by-symbol Shannon code needs 13 bits because it spends a full bit on each of the nine cheap `a`'s. Because this particular string has the source's average composition (nine `a`'s and one `b`), its surprisal equals $10\, H_2(0.9, 0.1) = 4.69$ bits; a message with two `b`'s would require more bits.

Naive floating-point narrowing only works for short messages. Since the width shrinks by a factor q at every step, after roughly a hundred symbols the two endpoints become indistinguishable within the 53-bit significand of a double-precision float (Exercise 3). Practical arithmetic coders solve this using fixed-width integer arithmetic and emit leading bits as soon as both endpoints agree on them, keeping the working interval rescaled to full precision (Witten et al. 1987). With rescaling, a message can be arbitrarily long and the two-bit overhead vanishes per symbol: a text of n symbols is stored in n times the model's average log-loss in bits.

37.2.2.2 A Language Model as a Compressor

Nothing in arithmetic coding assumed that symbols are independent. Any conditional model $q(\cdot \mid x_{<i})$ can drive the interval narrowing, provided encoder and decoder compute the same next-symbol probabilities bit for bit in finite precision. Pair an autoregressive language model with an arithmetic coder, and you have a lossless compressor: given a shared tokenizer, model, and coder, any document compresses to its total surprisal under the model plus at most two bits. (A practical file format adds minor framing overhead, and if the model itself is fixed in advance, both sides must already possess it.)

The simplest model that learns on the fly is a counting model. An *adaptive order-m model* predicts each character from the m characters immediately preceding it, its *context* s, using

$$q(x \mid s) = \frac{c(s,x) + \frac{1}{2}}{c(s) + \frac{k}{2}},$$

where $c(s,x)$ counts how many times x has followed s so far, $c(s) = \sum_x c(s,x)$ is the total count for context s, and k is the alphabet size. Adding $\frac{1}{2}$ to every count guarantees that no character ever receives zero probability (which would cost infinitely many bits). Crucially, the model updates its counts *after* coding each character. Because the decoder performs the same updates on the characters it has already recovered, encoder and decoder stay in lockstep and no model parameters ever need to be transmitted.

Let us test this compressor on *The Time Machine*. The first cell downloads and cleans the text down to lowercase letters and spaces, then defines a function that computes the adaptive code length in bits per character.

```
fname = d2l.download(d2l.DATA_URL + 'timemachine.txt',
                     sha1_hash='090b5e7e70c295757f55df93cb0a180b9691891a')
with open(fname) as f:
    text = re.sub('[^A-Za-z]+', ' ', f.read()).lower()   # 26 letters, space

def adaptive_bits(text, m, k):
    """Bits per character of the adaptive order-m model with half counts."""
    counts, totals, bits = {}, {}, 0.0
    for i, ch in enumerate(text):
        s = text[max(0, i - m):i]                # the context
        c = counts.setdefault(s, {})
        bits -= math.log2((c.get(ch, 0) + 0.5) / (totals.get(s, 0) + k / 2))
        c[ch] = c.get(ch, 0) + 1                 # learn after coding
        totals[s] = totals.get(s, 0) + 1
    return bits / len(text)
```

Next we evaluate contexts of length zero through four and compare them against `zlib` at its highest compression level. Longer contexts predict better once their counts are populated, yet they also have far more counts to learn; the output shows where that tradeoff turns.

```
k, n = len(set(text)), len(text)
freq = [text.count(ch) / n for ch in set(text)]
h_freq = -sum(f * math.log2(f) for f in freq)
print(f'{n} characters over {k} symbols, letter-frequency entropy '
```

```
        f'{h_freq:.3f} bits')
bits = [adaptive_bits(text, m, k) for m in range(5)]
for m, b in enumerate(bits):
    print(f'order {m}: {b:.2f} bits/char = {b * math.log(2):.2f} '
          f'nats/char, perplexity 2^bits = {2 ** b:.2f}')
print(f'zlib -9: {8 * len(zlib.compress(text.encode(), 9)) / n:.2f} bits/char')
print(f'order 0 minus the letter-frequency entropy: {bits[0] - h_freq:.4f} '
      f'bits/char, {(bits[0] - h_freq) * n:.0f} bits in all')
```

```
173428 characters over 27 symbols, letter-frequency entropy 4.078 bits
order 0: 4.08 bits/char = 2.83 nats/char, perplexity 2^bits = 16.91
order 1: 3.29 bits/char = 2.28 nats/char, perplexity 2^bits = 9.81
order 2: 2.75 bits/char = 1.90 nats/char, perplexity 2^bits = 6.70
order 3: 2.55 bits/char = 1.77 nats/char, perplexity 2^bits = 5.84
order 4: 2.78 bits/char = 1.92 nats/char, perplexity 2^bits = 6.85
zlib -9: 2.84 bits/char
order 0 minus the letter-frequency entropy: 0.0011 bits/char, 183 bits in all
```

Adding context drops the code length from 4.08 bits per character with no context down to 2.55 bits per character with three characters of context, beating `zlib`'s 2.84. Adding a fourth character, however, reverses the gain: the order-4 model climbs back up to 2.78 bits per character. Most of its $27^4 \approx 530{,}000$ possible contexts appear too rarely in the novel for the model to learn their next-character probabilities, and every extra bit spent while those counts warm up goes directly into the file.

By contrast, with no context at all there is very little to learn. The first line of output reports the entropy of the novel's empirical letter frequencies, 4.078 bits per character, which is the rate a context-free coder would achieve if it knew those frequencies in advance. The order-0 model starts from scratch, and the final line measures the exact cost of learning them: just 0.0011 bits per character, or 183 bits across the entire book. Learning 26 letter frequencies is cheap; learning half a million contexts is not. (We will return to the nats and perplexity columns in Section 37.2.4.1.)

An adaptive model avoids sending parameters, at the cost of predicting poorly early in the file while it learns. A pretrained model takes the opposite trade-off: it compresses every file well from the very first byte, provided its weights have already been shared once. Delétang et al. (2023) demonstrate this trade-off with a 70-billion-parameter language model. Driving an arithmetic coder with its next-token probabilities compresses ImageNet image patches to 43.4% of their raw size (compared with 58.5% for PNG) and LibriSpeech audio to 16.4% (compared with 30.3% for FLAC). Those figures exclude the model itself: at two bytes per parameter, the weights occupy 140 GB against 1 GB for each dataset, and once the model size is counted it no longer beats the standard codecs.

37.2.2.3 Minimum Description Length

In our *The Time Machine* experiment, each adaptive model paid for learning its counts as it went. Remove that charge and the ranking flips completely: if we first gather counts from the entire novel and then score the same novel with those counts, order 4 beats order 3, order 5 beats order 4, and so on, because a larger lookup table can simply memorize the text (Exercise 4). Judged solely by how well it fits the data in hindsight, a richer model never loses. Yet a receiver who does not already possess the model must first be sent its description. The *minimum description length* (MDL) principle resolves this by choosing the

model $\mathcal{M}$ that minimizes the total code length of the model and the data together:

$$\hat{\mathcal{M}} = \operatorname*{argmin}_{\mathcal{M}} \big[L(\mathcal{M}) + L(\mathcal{D} \mid \mathcal{M}) \big], \tag{37.18}$$

where $L(\mathcal{M})$ is the number of bits needed to specify the model and $L(\mathcal{D} \mid \mathcal{M})$ is the log-loss of the data $\mathcal{D}$ under that model, which is its code length in bits by the previous subsections. Under MDL, a more elaborate model is preferred only if it saves more bits on the data than it costs to describe. In this sense MDL is Occam's razor quantified in bits. Moreover, when the model description comes from a prior distribution via $L(\mathcal{M}) = -\log_2 P(\mathcal{M})$, minimizing the sum is minimizing minus the log prior minus the log likelihood, so MDL coincides with maximum a posteriori estimation (Section 36.6.4).

A simple coin-flipping example makes both terms concrete. Suppose $n = 1{,}000$ flips produce 600 heads. The fair-coin hypothesis $\mathcal{M}_0$ has no parameters to transmit and codes each flip in 1 bit, giving $L(\mathcal{D} \mid \mathcal{M}_0) = 1{,}000$ bits. The biased-coin hypothesis $\mathcal{M}_1$ must first transmit its bias θ, which requires choosing a finite precision. Because the sample proportion of heads in n flips fluctuates on the scale of $1/\sqrt{n}$, sending digits of θ finer than $1/\sqrt{n}$ wastes bits describing sampling noise (Exercise 5). A grid of spacing $1/\sqrt{n}$ on $(0, 1)$ has $\sqrt{n}$ points, so naming the closest grid point costs $L(\mathcal{M}_1) = \log_2 \sqrt{n} = \frac{1}{2} \log_2 1{,}000 \approx 5$ bits. Given θ, the data take $L(\mathcal{D} \mid \mathcal{M}_1) \approx 1{,}000\, H_2(0.6, 0.4) = 971$ bits, for a total of $976 < 1{,}000$ bits. MDL therefore selects the biased coin: spending 5 bits on the parameter saves 29 bits on the data. Had we observed only 520 heads, the data savings would be just 1.2 bits, less than the 5-bit parameter cost, and MDL would keep the fair coin. Charging $\frac{1}{2} \log_2 n$ bits per real parameter in this fashion is known as the *Bayesian information criterion*.

In fact, our *The Time Machine* experiment was already applying MDL. An adaptive code length is a valid complete description of the data, with the cost of learning the model folded directly into the bit stream. For the order-0 model, that learning cost is tiny: charging $\frac{1}{2} \log_2 n \approx 8.7$ bits for each of the 26 free letter frequencies predicts about 226 bits, reasonably close to the 183 bits we actually observed (the criterion retains only the leading asymptotic term). For the order-4 model, learning a separate distribution for every observed context cost more bits than the fourth character of context saved, so the adaptive code length automatically favored order 3.

Notice that a naive parameter count would have missed the mark badly here. The order-4 model formally has $27^4 \cdot 26 \approx 14$ million free frequencies; at 8.7 bits each, the Bayesian information criterion would assess a penalty of 120 million bits for a novel that takes only 1.4 million bits uncompressed at eight bits per character. Parameter-counting asymptotics assume far more data than parameters. In reality, most of those four-character contexts never appear, and the adaptive code charges nothing for them. Whenever a model has more parameters than data, including modern neural networks, the adaptive code length is the description length that remains meaningful.

37.2.3 Entropy Rate and Typical Sequences

Up to now we have taken a model as given and measured the length of its code. Now we turn the question around: for a source with memory, how short can the code of the *best possible* model be?

37.2.3.1 The Entropy Rate

For independent symbols, the fundamental compression floor is the single-symbol entropy. Real text, however, is full of dependence: after `q` comes `u`, and after `th` comes `e` or `a` far more often than `z`. To analyze sources with memory, suppose the sequence $X_1, X_2, \ldots$ is *stationary*, meaning that the joint distribution of any block of symbols is unchanged when shifted in time.

We draw on the joint and conditional entropies introduced in Section 37.1.1.3. For the remainder of this section, H denotes entropy in bits for random variables, while $H_2(p_1, \ldots, p_k)$ continues to denote the binary entropy of an explicit probability vector. The joint entropy $H(X_1, \ldots, X_n)$ treats an entire n-symbol block as a single random variable, while the conditional entropy $H(X_n \mid X_{<n}) = \mathbb{E}\big[-\log_2 p(X_n \mid X_{<n})\big]$ measures the expected log-loss of the best possible next-symbol predictor, namely the true source itself. Applying the chain rule $H(X, Y) = H(X) + H(Y \mid X)$ repeatedly breaks the block entropy into a sum of conditional entropies, $H(X_1, \ldots, X_n) = \sum_{i=1}^{n} H(X_i \mid X_{<i})$. The *entropy rate* of the source is the limiting conditional entropy, or equivalently the limiting entropy per symbol of long blocks:

$$H_{\text{rate}} = \lim_{n\to\infty} \frac{1}{n} H(X_1, \ldots, X_n) = \lim_{n\to\infty} H(X_n \mid X_{<n}). \tag{37.19}$$

Because the block entropy is the sum of n conditional entropies, the first expression is the running average of the terms whose limit appears in the second. For any stationary source, both limits exist and coincide, and $\frac{1}{n} H(X_1, \ldots, X_n)$ decreases monotonically toward H_{rate} (Cover and Thomas 1999), facts we take as given. The intuition is straightforward: conditioning never increases entropy (Section 37.1), so each additional symbol of context can only reduce next-symbol uncertainty, and the running average of a nonincreasing sequence is itself nonincreasing. When symbols are independent and identically distributed, both limits reduce immediately to $H(X_1)$.

A two-state Markov chain shows how strongly memory lowers the floor. Suppose the chain emits `a` or `b`, staying at its current symbol with probability 0.9 and switching with probability 0.1. By symmetry, both symbols occur equally often, so a single symbol viewed in isolation has entropy $H(X_1) = 1$ bit. Given the previous symbol, however, the next symbol behaves like a biased coin with probability 0.1 of switching, so $H(X_n \mid X_{n-1}) = H_2(0.9, 0.1) = 0.47$ bits, and looking further back adds no new information. Memory alone cuts the fundamental floor from 1 bit per symbol down to 0.47 (Exercise 6 analyzes the general asymmetric two-state chain).

For any stationary source, no uniquely decodable code can average fewer than H_{rate} bits per symbol. Indeed, any code for blocks of n symbols is a symbol code over the alphabet of n-symbol blocks; by the entropy bound (extended to uniquely decodable codes via McMillan's theorem), it must spend at least $H(X_1, \ldots, X_n)$ bits per block, and monotonicity guarantees $H(X_1, \ldots, X_n) \geq n H_{\text{rate}}$.

Just as importantly, this floor is achievable. An arithmetic coder driven by the source's true conditional probabilities spends strictly less than the message's surprisal plus two bits by Equation 37.17. Since the expected surprisal of an n-symbol block under its own distribution is $H(X_1, \ldots, X_n)$, the coder averages less than $\frac{1}{n} H(X_1, \ldots, X_n) + \frac{2}{n}$ bits per symbol, which converges to H_{rate} as n grows.

In a classic experiment, Shannon asked people to guess the next letter of printed English and concluded that its entropy rate lies between roughly 0.6 and 1.3 bits per character (Shannon 1951), far below the $\log_2 27 = 4.75$ bits required for uniformly random letters and spaces. Notice that any working compressor establishes an *upper* bound on the entropy rate: our order-3 counting model's 2.55 bits per character is one such upper bound, whereas no compressor can ever prove a lower bound. Returning to the chapter introduction, a language model with a loss of 1.2 nats per character achieves 1.73 bits per character, well below our counting model's 2.55, yet still above the top of Shannon's range.

The best upper bounds today come from large language models. Driving an arithmetic coder, the model of Delétang et al. (2023) compresses Wikipedia text to less than one bit per byte, inside Shannon's range. Three caveats apply before we read this as the entropy rate of English: the size of the model is not counted, some of the test text may have been seen in training, and a byte of Wikipedia markup is not a symbol of Shannon's 27-letter alphabet. Where the true rate lies below these bounds, nobody knows.

37.2.3.2 Typical Sequences

So far we have bounded *average* code lengths. What does a single long message look like, and how many likely messages actually exist? Answering this illuminates entropy from a second angle, as an exponent counting the set of likely messages. Nothing in the rest of the section relies on this subsection, so you may skip it on a first reading. To keep the numbers concrete, consider independent binary symbols that equal 1 with probability 0.3, in messages of length $n = 200$.

The single most probable message is all zeros, with probability $0.7^{200} = 2^{-103}$. A message with 60 ones, the expected count, has probability $0.3^{60}\,0.7^{140} = 2^{-176.3}$, which equals 2^{-nH} where $H = H_2(0.3, 0.7) = 0.881$ bits; there are $\binom{200}{60} = 2^{172.2}$ such messages. Meanwhile, the total number of strings peaks at 100 ones, with $2^{195.9}$ strings, but each one has probability 2^{-225}, so all of them combined carry only 2^{-29} of the total probability mass. Most messages are unlikely, and the most likely message is unrepresentative.

Write $x_{1:n}$ for the sequence $x_1, \dots, x_n$. Because the symbols are independent, the surprisal per symbol is a sample mean of i.i.d. random variables, each with expectation $H = H_2(P)$. By the law of large numbers, that sample mean converges in probability to its expectation:

$$-\frac{1}{n}\log_2 P(X_{1:n}) = \frac{1}{n}\sum_{i=1}^{n} -\log_2 p(X_i) \xrightarrow{P} H. \tag{37.20}$$

For any fixed tolerance $\delta > 0$, a long message therefore has surprisal within $n\delta$ of nH bits with probability approaching one. This phenomenon is the *asymptotic equipartition property* (AEP), and the messages it captures form the *typical set* at tolerance δ:

$$A_\delta = \left\{ x_{1:n} : \left| -\frac{1}{n}\log_2 P(x_{1:n}) - H \right| < \delta \right\}. \tag{37.21}$$

Proposition (AEP). *Let $X_1, \dots, X_n$ be independent draws from a distribution P on a finite alphabet with binary entropy H, and let $\delta > 0$. Then: (i) $P(A_\delta) \to 1$ as $n \to \infty$; (ii) every $x_{1:n} \in A_\delta$ satisfies $2^{-n(H+\delta)} < P(x_{1:n}) < 2^{-n(H-\delta)}$; and (iii) $|A_\delta| \le 2^{n(H+\delta)}$, while $|A_\delta| \ge (1-\eta)\,2^{n(H-\delta)}$ whenever $P(A_\delta) \ge 1-\eta$.*

Proof. Part (i) is immediate from Equation 37.20, and part (ii) simply exponentiates the defining inequality of A_δ. For part (iii), summing the bounds in (ii) over all members of A_δ yields $1 \geq P(A_\delta) > |A_\delta|\, 2^{-n(H+\delta)}$ and $1 - \eta \leq P(A_\delta) < |A_\delta|\, 2^{-n(H-\delta)}$, from which both size bounds follow directly. ■

On the exponential scale, then, almost all probability mass concentrates on roughly 2^{nH} typical sequences, each occurring with probability roughly 2^{-nH}. Keep in mind that "roughly" hides a factor of $2^{\pm n\delta}$, which is substantial at finite lengths: for $n = 200$ and $\delta = 0.1$, that slack is 2^{20}.

The AEP leads directly to Shannon's source-coding theorem (Shannon 1948; Cover and Thomas 1999). To compress, assign each sequence in A_δ a fixed-length index of $\lceil n(H+\delta) \rceil$ bits and declare failure on any atypical message. The number of bits spent per source symbol is called the *rate* of the code; here the rate is $H + \delta$ (up to rounding), and the failure probability $P(A_\delta^c) \to 0$. Since $\delta > 0$ can be made arbitrarily small, every rate above H is achievable. (Prepending a single flag bit and falling back to an uncompressed copy for atypical messages makes the code strictly lossless at the same asymptotic rate.) Conversely, any fixed-length code of rate $R < H$ has only 2^{nR} codewords, so it can represent a set B of at most 2^{nR} messages. Choosing $\delta < H - R$, we find

$$P(B) \leq P(B \cap A_\delta) + P(A_\delta^c) < 2^{nR}\, 2^{-n(H-\delta)} + P(A_\delta^c) \to 0.$$

Thus every fixed-length code operating at a rate below H fails with probability tending to one. The same theorem holds for stationary ergodic sources with H_{rate} in place of H, a generalization we grant without proof (Cover and Thomas 1999).

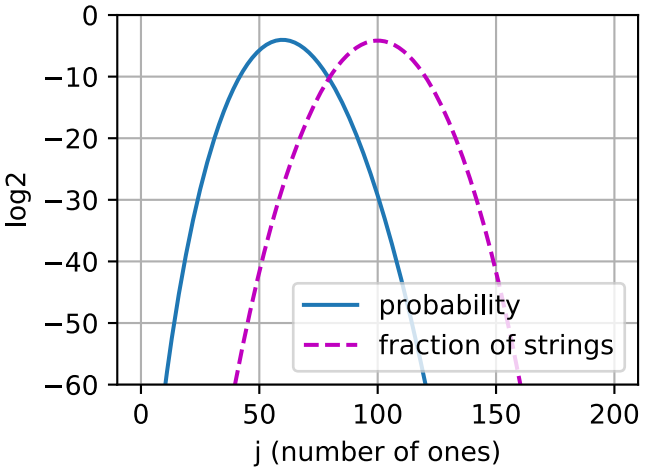
For our Bernoulli source we can count the typical set *exactly*, since all messages with the same number j of ones share the same probability. The cell below groups all 2^{200} binary strings by j, sums the counts and probabilities of the groups that satisfy the typicality condition at $\delta = 0.1$, and compares the resulting size against the benchmark $2^{nH} = 2^{176.3}$.

```
n, p, delta = 200, 0.3, 0.1
j = np.arange(n + 1)                                  # number of ones
log_fact = np.concatenate([[0.0], np.cumsum(np.log(np.arange(1, n + 1)))])
log2_count = (log_fact[n] - log_fact[j] - log_fact[n - j]) / np.log(2)
bits_per_symbol = -(j * np.log2(p) + (n - j) * np.log2(1 - p)) / n
log2_prob_j = log2_count - n * bits_per_symbol        # log2 P(j ones)
h = -(p * np.log2(p) + (1 - p) * np.log2(1 - p))
typical = np.abs(bits_per_symbol - h) < delta
log2_size = np.logaddexp2.reduce(log2_count[typical])
prob = (2 ** log2_prob_j[typical]).sum()
print(f'H = {h:.3f} bits/symbol, nH = {n * h:.1f} bits')
print(f'typical: 2^{log2_size:.1f} of 2^{n} strings, '
      f'{j[typical].min()} <= j <= {j[typical].max()}')
print(f'probability they carry: {prob:.4f}')
print(f'bounds: {prob:.3f} * 2^{n * (h - delta):.1f} <= |A| '
      f'<= 2^{n * (h + delta):.1f}')
```

```
H = 0.881 bits/symbol, nH = 176.3 bits
typical: 2^188.8 of 2^200 strings, 44 <= j <= 76
probability they carry: 0.9893
bounds: 0.989 * 2^156.3 <= |A| <= 2^196.3
```

The $2^{188.8}$ typical strings, those containing between 44 and 76 ones, make up only one in 2^{11} of all binary strings, yet together they carry probability 0.989. Their count exceeds 2^{nH} by a factor of several thousand because $\delta = 0.1$ admits strings with up to 76 ones, which are far more numerous than those near 60; still, the size stays comfortably within the proposition's bounds of $0.989 \cdot 2^{156.3}$ and $2^{196.3}$. Let us plot both the probability and the fraction of strings as a function of j to see how cleanly they separate.

```
d2l.plot(j, [log2_prob_j, log2_count - n], 'j (number of ones)', 'log2',
         legend=['probability', 'fraction of strings'],
         ylim=[-60, 0])
```



The two curves peak far apart. Probability mass peaks at $j = 60$, where strings account for only a 2^{-28} fraction of all possibilities; the count of strings peaks at $j = 100$, where their combined probability is just 2^{-29}. The single most probable string ($j = 0$, all zeros) has probability 2^{-103} and drops completely off the chart. Typical sequences are neither the most probable strings nor the most numerous ones; they live in the narrow band where the product of count and individual probability is largest.

The typical set is a powerful tool for proofs, though a clumsy practical coder. Indexing it directly costs 189 bits for every typical string, 13 bits above $nH = 176.3$, and fails on roughly one message in a hundred. By contrast, arithmetic coding with the true model is strictly lossless and averages under $nH + 2 = 178.3$ bits by Equation 37.17. What the typical set gives us is geometric intuition: a long message almost surely reflects the source's true composition, and roughly 2^{nH} messages share that composition.

37.2.4 Perplexity

In practice, language models rarely report their performance as a file size. Instead, they report a single number on held-out text, *perplexity*, which is simply the code length of this section expressed in a different unit.

37.2.4.1 Perplexity as a Branching Factor

Given a text of n tokens and a model assigning next-token probabilities $q(\cdot \mid x_{<i})$, the average log-loss per token in nats and its exponential, the *perplexity* (Jelinek et al. 1977), are defined as

$$\mathrm{CE} = -\frac{1}{n}\sum_{i=1}^{n} \log q(x_i \mid x_{<i}), \qquad \mathrm{PPL} = \exp(\mathrm{CE}) = \Big(\prod_{i=1}^{n} q(x_i \mid x_{<i})\Big)^{-1/n}. \tag{37.22}$$

Written without distribution arguments, CE denotes the model's empirical loss on this particular text; its expectation over texts drawn from the source is the cross-entropy between source and model. The second expression shows that perplexity is the inverse geometric mean of the probabilities assigned to the observed tokens. As introduced in Section 8.3.2, perplexity has an immediate interpretation as an *effective branching factor*: a model that spreads its probability uniformly over k tokens at every step has perplexity k, whereas a perfect predictor on a deterministic sequence has perplexity 1.

The *The Time Machine* results in Section 37.2.2.2 make this concrete on character-level text. A completely uninformed model would guess uniformly among all 27 symbols, giving a perplexity of 27. Learning letter frequencies alone cuts the perplexity to 16.91 (roughly 17 equally likely choices per character), and using three characters of context brings it down to 5.84: on average, the model is about as uncertain about the next character as you are about the roll of a six-sided die. Note that our code computed perplexity as 2 raised to the bits per character, whereas Equation 37.22 uses e raised to the nats per character; both yield the same number because $\mathrm{PPL} = b^{\mathrm{CE}_b}$ for any logarithm base b (Exercise 8).

37.2.4.2 The Perplexity Floor

Just as code length cannot drop below the entropy rate, perplexity has a fundamental floor: for a stationary source, the expected perplexity is at least $2^{H_{\text{rate}}}$. Because CE is measured in nats while H_{rate} in Equation 37.19 is measured in bits, a factor of $\ln 2$ converts between the two.

Proposition (perplexity floor). *Let $X_1, \dots, X_n$ be drawn from a stationary source, let q be any model, and let CE and PPL be the empirical loss and perplexity Equation 37.22 on this text under q. Then*

$$\mathbb{E}[\mathrm{CE}] \geq H_{\text{rate}} \ln 2, \qquad \mathbb{E}[\mathrm{PPL}] = \mathbb{E}\big[e^{\mathrm{CE}}\big] \geq 2^{H_{\text{rate}}}. \tag{37.23}$$

Proof. Fix any prefix $x_{<i}$. The model's expected next-token loss given $x_{<i}$ is the cross-entropy between the true conditional distribution $p(\cdot \mid x_{<i})$ and $q(\cdot \mid x_{<i})$, which by Gibbs' inequality is at least the conditional entropy of the source. Averaging over prefixes gives $\mathbb{E}[-\log q(X_i \mid X_{<i})] \geq \mathbb{E}[-\log p(X_i \mid X_{<i})]$. Because the log-probability of a sequence decomposes as $\log p(x_{1:n}) = \sum_i \log p(x_i \mid x_{<i})$, summing over i and dividing by n yields $\mathbb{E}[\mathrm{CE}] \geq \frac{\ln 2}{n} H(X_1, \dots, X_n) \geq H_{\text{rate}} \ln 2$, where the second inequality follows from the monotone decrease of the block entropy per symbol. Finally, since e^x is convex, Jensen's inequality gives $\mathbb{E}[e^{\mathrm{CE}}] \geq e^{\mathbb{E}[\mathrm{CE}]} \geq 2^{H_{\text{rate}}}$. ■

In short, no model can be less perplexed on average than the intrinsic randomness of the source permits. To get a sense of scale, suppose a text had an entropy rate of one bit per character, a value inside Shannon's range for English. Then no character-level model could beat an expected perplexity of 2, one fair coin flip per letter, and our order-3 counting model's 5.84 would sit far above the floor. For real text the floor is as uncertain as the entropy rate itself, which we know only through upper bounds.

Notice carefully that the proposition bounds an *expectation*; the perplexity of a single short text can easily dip below the floor. Consider again our source that emits `a` with probability 0.9, whose floor is $2^H = 1.38$. Even under the true model, a ten-symbol string scores below 1.38 whenever it contains no `b` at all, which occurs with probability $0.9^{10} = 35\%$. Even for a long text from this source, the perplexity still falls below 2^H nearly half the time, though by a margin that vanishes as length grows. More generally, for an ergodic source scored by the true model, the perplexity of a single long text converges to $2^{H_{\text{rate}}}$ as $n \to \infty$, a fact we grant.

37.2.4.3 Perplexity across Tokenizers

Because perplexity is averaged per token, its value depends directly on the tokenizer, so two models can be compared by raw perplexity only when they share the same tokenizer. Section 8.3.2.4 demonstrates this pitfall across three tokenizers and resolves it by dividing total surprisal by the number of bytes in the text. In the language of this section, that fix simply converts perplexity back into a code length per unit of raw text. A text of n_{char} characters split into n_{tok} tokens has total code length $n_{\text{tok}} \log_2 \text{PPL}$ bits, giving

$$\text{bpc} = \frac{n_{\text{tok}}}{n_{\text{char}}} \log_2 \text{PPL}, \tag{37.24}$$

where the denominator depends solely on the text itself. When every character is one byte, as in our 27-symbol text, bits per character are identical to the bits per byte used in the main text.

Every quantity in this section, from code length to perplexity, relies on one crucial requirement: both encoder and decoder must be able to evaluate the model's probability $q(x_i \mid x_{<i})$ for each symbol. When two distributions are accessible only through samples, as when comparing a generative model's outputs against real data, no such code exists. Section 37.3 and Section 37.4 develop ways to compare distributions when we have only samples to work with.

37.2.5 Summary

A model's log-loss is a file size once we pair it with the right code. Any symbol-by-symbol prefix code must round each codeword to a whole number of bits, placing its optimal expected length between $H_2(P)$ and $H_2(P) + 1$ bits. Arithmetic coding removes that per-symbol penalty by rounding only once per message: driven by any next-symbol model, including an autoregressive language model, it compresses a sequence into its total surprisal plus at most two bits. Under a mismatched model Q, the expected code length is the cross-entropy $\text{CE}_2(P, Q)$, and the KL divergence measures the overhead of using Q instead of P. On *The Time Machine*, an adaptive order-3 counting model compresses the text to 2.55 bits per character and beats `zlib`, whereas order 4 does worse because the bits spent learning its

rarer contexts outweigh what the extra character saves, illustrating the minimum description length principle. For stationary sources with memory, no uniquely decodable code averages fewer bits per symbol than the entropy rate H_{rate}. Finally, perplexity is simply this code length exponentiated: it acts as an effective branching factor, obeys the expected lower bound $2^{H_{\text{rate}}}$, and can be compared across different tokenizers only after converting back to bits per character or bits per byte.

37.2.6 Exercises

1. **McMillan's theorem.** A code is uniquely decodable if every concatenation of codewords parses in only one way.
   1. Show that the code $\{0, 01, 011\}$ is uniquely decodable but not prefix-free, and check that it satisfies the Kraft inequality.
   2. Prove that every uniquely decodable code satisfies the Kraft inequality Equation 37.13. Hint: expand $\left(\sum_i 2^{-\ell_i}\right)^m$ as a sum over all strings of m codewords of $2^{-(\text{total length})}$, and group the strings by their total length L, where $m \le L \le m\ell_{\max}$. Unique decodability allows at most 2^L strings of total length L, so the sum is at most $m\ell_{\max}$. Take m-th roots and let $m \to \infty$.
2. **Huffman codes.** Huffman's algorithm (Huffman 1952) builds a prefix code by repeatedly merging the two least probable symbols into one symbol of their summed probability, and then assigning 0 and 1 to the two members of each merge.
   1. Run it on $P = (0.6, 0.3, 0.1)$ and on $(0.4, 0.3, 0.2, 0.1)$. Report the codeword lengths and the expected length of each code, next to the Shannon code and H_2.
   2. Prove that Huffman's code has the smallest expected length of all prefix codes. First show that in an optimal code $p_i > p_j$ implies $\ell_i \le \ell_j$, and that the two longest codewords have the same length and can be taken to be siblings in the tree. Then show that merging the two least probable symbols i, j reduces the problem by one symbol with $\mathbb{E}[\ell] = \mathbb{E}[\ell'] + p_i + p_j$, where ℓ' are the lengths of the merged code. Conclude by induction. Hint: exchange two codewords and compare the expected lengths.
   3. For the source that emits `a` with probability 0.9, show that every symbol code spends at least one bit per symbol. Then code *pairs* of symbols as a four-symbol source with Huffman's algorithm and check the text's 0.645 bits per original symbol. Do the same for triples. How does the slack over the entropy compare with the bound $1/n$ in each case?
   4. Consider the linear guessing code with lengths $1, 2, \dots, k-1, k-1$. Sort the probabilities, $p_1 \ge \cdots \ge p_k$, and show that Huffman's algorithm can return these lengths whenever $p_i \ge \sum_{j>i} p_j$ for every i, so that linear guessing is then an optimal prefix code. The source $(0.4, 0.3, 0.2, 0.1)$ violates the condition. Is linear guessing optimal for it?
3. code **Exact arithmetic coding.** ● The interval narrowing of Section 37.2.2.1 stores `lo` and `hi` as floating-point numbers. Draw a message of $n = 1000$ symbols, each of which is `a` with probability 0.9 and `b` otherwise.
   1. Run the floating-point narrowing on this message with the model $q(\texttt{a}) = 0.9$. After how many symbols does `hi - lo` first differ from the product of the probabilities by more than 0.1%, and after how many does it become zero? Relate both answers to the 53-bit significand of a double-precision number.

2. Repeat the narrowing with exact rationals (`fractions.Fraction`). Name a point of the final interval as the text's cell does, write a decoder that replays the model to recover the message, and compare the code length with the surprisal of the message and with $nH_2(0.9, 0.1)$.
3. Encode the same message with the mismatched model $q(\mathtt{a}) = 0.7$. Compare the increase in code length with $nD_{\mathrm{KL}}^{(2)}(P\|Q)$. Why do the two numbers differ for a single message?

4. [extended] **Adaptive context-model compression.** ●● The text's `adaptive_bits` is a complete description length in the sense of Equation 37.18, with the cost of learning the model inside it. By Equation 37.17, an arithmetic coder driven by its predictions spends at most two bits more. Blier and Ollivier (2018) use such prequential codes to measure the description length of deep networks.
   1. Extend the text's experiment to $m = 0, 1, \ldots, 8$. For each m, compare the adaptive code length with the per-character cross-entropy of the order-m model whose counts are taken from the entire text, and with `zlib`.
   2. Explain why the adaptive code length has a minimum over m while the cross-entropy of the fitted model keeps falling. Which order does MDL select? Which term of Equation 37.18 does the cross-entropy of the fitted model leave out?
   3. Design an adaptive model that combines contexts of several orders, for example by falling back to shorter contexts when a long context has few counts. Report a table of bits per character for every model you tried, together with `zlib`, and a paragraph on the design choice that mattered most.

5. **Two-part code for a coin.** ● A sequence $\mathcal{D}$ of n coin flips contains j heads, and $\hat{\theta} = j/n$. Hypothesis $\mathcal{M}_0$ is the fair coin: it has no parameter to describe, and $L(\mathcal{D} \mid \mathcal{M}_0) = n$ bits. Hypothesis $\mathcal{M}_1$ is a coin with bias θ. Its encoder sends the point θ_q nearest to $\hat{\theta}$ on a grid of spacing Δ in $(0, 1)$, which costs $L(\mathcal{M}_1) = \log_2(1/\Delta)$ bits, and the decoder knows n and Δ.
   1. Use the decomposition Equation 37.14 to show that $L(\mathcal{D} \mid \mathcal{M}_1) = nH_2(\hat{\theta}, 1-\hat{\theta}) + nD_{\mathrm{KL}}^{(2)}(\hat{\theta}\,\|\,\theta_q)$, where the KL divergence is that of the Bernoulli distributions with these parameters. In nats, this KL divergence is $(\theta_q - \hat{\theta})^2/(2\hat{\theta}(1-\hat{\theta}))$ to second order. Derive the expansion, and show that the total length Equation 37.18 of $\mathcal{M}_1$ exceeds $nH_2(\hat{\theta}, 1-\hat{\theta})$ by at most about $\log_2(1/\Delta) + n\Delta^2/(8\hat{\theta}(1-\hat{\theta})\ln 2)$ bits.
   2. Which spacing Δ minimizes this bound? The decoder must know Δ before it decodes θ_q. Why can the code therefore not use the minimizer? Show that the spacing $\Delta = 1/\sqrt{n}$, which depends on n alone, gives an excess of $\frac{1}{2}\log_2 n + O(1)$ bits whenever $\hat{\theta}$ stays bounded away from 0 and 1.
   3. Take $n = 1000$ and the approximation $L(\mathcal{M}_1) + L(\mathcal{D} \mid \mathcal{M}_1) \approx nH_2(\hat{\theta}, 1-\hat{\theta}) + \frac{1}{2}\log_2 n$ bits. The text showed that $\mathcal{M}_1$ wins at $j = 600$ by 24 bits. Which hypothesis wins at $j = 520$? Approximately how far from $n/2$ must j lie for $\mathcal{M}_1$ to win, and how many standard deviations $\sqrt{n}/2$ of j under the fair coin is that?

6. **Entropy rate of a two-state chain.** A chain on $\{\mathtt{a}, \mathtt{b}\}$ moves from `a` to `b` with probability 0.1 and from `b` to `a` with probability 0.3, and it starts from its stationary distribution.
   1. Find the stationary distribution and $H(X_1)$ in bits.

2. Show that $H(X_n \mid X_{n-1}) = \sum_s \pi_s H_2(r_s, 1-r_s)$ bits, where π is the stationary distribution and r_s is the exit probability of state s, and that the entropy rate equals this value. Compute it.
3. Show that $\frac{1}{n}H(X_1, \dots, X_n) = \frac{1}{n}\big(H(X_1) + (n-1)H_{\text{rate}}\big)$ and that it decreases to H_{rate}.
4. Consider two limits. With both exit probabilities equal to $\frac{1}{2}$ the chain is i.i.d. and the rate is 1 bit. With both exit probabilities equal to r, the rate tends to 0 as $r \to 0$, while $H(X_1)$ stays at 1 bit. What does $H(X_1)$ say about how well the chain compresses?
5. code Simulate 10^5 steps of the first chain and code them with the text's `adaptive_bits` at orders 0, 1 and 2. Which orders reach the rate, and what does the order-0 length converge to?

7. code **Typical and high-probability sets.** ● The code of Section 37.2.3.2 declares a binary string of length $n = 200$ from a Bernoulli source with $p = 0.3$ typical when its surprisal per symbol lies within $\delta = 0.1$ bit of the entropy H in bits. It finds $2^{188.8}$ typical strings, inside the bounds $0.989 \cdot 2^{156.3} \le |A_\delta| \le 2^{196.3}$ of the AEP proposition and well above $2^{nH} = 2^{176.3}$.

   1. Which string is the most probable one? Compute its surprisal per symbol and decide whether it is typical. Why does this string not matter for compression?
   2. Let A be the typical set and let B be any set of strings with $P(B) \ge 1 - \eta$. Show that $P(A \cap B) \ge 1 - \eta - P(A^c)$, and conclude that $|B| \ge \big(1 - \eta - P(A^c)\big)\, 2^{n(H-\delta)}$.
   3. The smallest set with probability at least 0.9 collects strings in order of decreasing probability. Group the strings by their number j of ones, as the code does, and compute $\frac{1}{n}\log_2$ of the size of this set for $n = 200$, 1000, 5000, and 20000. Compare it with H and with $\frac{1}{n}\log_2 |A_\delta|$ at the same n. Which of the two sets grows like 2^{nH}, and why does the typical set with a fixed δ not?

   Adapted from David Tse, EE376A/Stats376A: Information Theory, Homework 2, Problem 6.

8. **Perplexity across tokenizers.** Perplexity is computed per token, so it depends on how a text is split into tokens.

   1. Show that b^{CE_b} is the same number for every log base b, where CE_b is the per-token cross-entropy in base-b units.
   2. Two language models are evaluated on the same text of 10^6 characters. The tokenizer of model A splits it into 2.5×10^5 tokens, and A reaches a per-token perplexity of 20. The tokenizer of model B splits it into 6.7×10^5 tokens, and B reaches a per-token perplexity of 8. Convert both results to bits per character with Equation 37.24. Which model predicts the text better?
   3. Take the entropy rate of the text to be one bit per character, a value inside the range that Shannon (1951) estimated for English. By Equation 37.23 the expected per-token perplexity is at least 2^H, where H is the entropy rate per *token* in bits. What is the lowest perplexity any model can reach in expectation with the tokenizer of A, and with that of B? How far is each model from its floor?
   4. Explain in two sentences why the two perplexities cannot be compared directly while the two values of bits per character can.

37.3 f-Divergences

Throughout this book, whenever we fit a model by maximum likelihood, we measure how well it matches the data with a single number: the KL divergence. As we saw at the end of Section 37.1, however, KL divergence has a glaring blind spot: it jumps to infinity the moment a model assigns zero probability to an event that actually occurs. A classifier rarely runs into this problem, because a softmax layer never outputs an exact zero. A generative network that maps a low-dimensional noise vector into image space runs into it immediately, because its samples lie on a thin surface in image space that can miss the real data altogether. How else can we compare two distributions, and which comparisons can still distinguish a model that is nearly right from one that is far off?

The simplest possible example already reveals what can go wrong. Suppose the data sit at a single point, $P = \delta_0$, while the model places all its mass at a single point a distance d away, $Q = \delta_d$. Intuitively, a model at $d = 0.01$ is nearly right and one at $d = 100$ is far off, so a useful loss should tell them apart. Table 37.2 shows what five standard comparisons report.

Table 37.2: Five comparisons of the point masses δ_0 and δ_d. The MMD row uses the RBF kernel with length scale 1.

comparison	$d = 0.01$	$d = 1$	$d = 100$
KL divergence $D_{\mathrm{KL}}(P\|Q)$	∞	∞	∞
Jensen–Shannon divergence	$\log 2$	$\log 2$	$\log 2$
total variation distance	1	1	1
maximum mean discrepancy MMD^2	0.0001	0.79	2.00
Wasserstein distance W_1	0.01	1	100

Look closely at the table: three of the five rows do not budge as d varies. A model trained with any of those three receives the same loss at $d = 0.01$ as at $d = 100$, and therefore zero gradient with respect to d. The original generative adversarial network (GAN) of Chapter 16 sat in this predicament, since its objective reduces to the Jensen–Shannon row up to constants.

Those three frozen rows belong to a single family. KL divergence compares P and Q through the ratio p/q of their densities. If we keep that density ratio and vary the convex function applied to it, we obtain a broad family called the *f-divergences*. All of its members share one proof of non-negativity, one method for estimation from samples, and one structural reason for being blind to the distance d. The remaining two rows, which do move with d, avoid the density ratio altogether; we will study them in Section 37.4.

Once the family is in place, we will address three questions that recur throughout deep learning. First, does it matter which distribution comes first in KL, the data as in maximum likelihood or the model as in variational inference (Section 37.3.2)? Second, what does a small divergence actually guarantee to someone trying to tell P from Q (Section 37.3.3)? Third, what can we still compute when both distributions are known only through samples (Section 37.3.4)? Section 37.2 ended on that last question, and adversarial training rests on its answer.

Throughout this section, all logarithms are natural, so KL and Jensen–Shannon divergences are measured in nats. We write P for the data or target distribution and Q for the model, p and q for their probability mass functions or densities, and $u = p/q$ for their *density ratio*.

```
from scipy import integrate, optimize
```

We can check every entry of Table 37.2 directly in code. The next cell represents δ_0 and δ_d as the probability vectors $(1, 0)$ and $(0, 1)$ on the two-point support $\{0, d\}$ and evaluates each comparison from its definition. Three of these definitions follow in this section and the other two in the next.

```
def kl_div(p, q):
    """D_KL(P||Q) on a finite support: 0 log 0 = 0, and inf if q = 0 < p."""
    s = p > 0
    with np.errstate(divide='ignore'):
        return np.sum(p[s] * np.log(p[s] / q[s]))

def five_comparisons(atoms, p, q, ell=1.0):
    """KL, JS, TV, RBF MMD^2 and W1 between pmfs p, q on sorted atoms."""
    m = (p + q) / 2                                  # the mixture for JS
    K = np.exp(-(atoms[:, None] - atoms[None, :]) ** 2 / (2 * ell ** 2))
    F_gap = np.abs(np.cumsum(p) - np.cumsum(q))[:-1]  # |F_P - F_Q| between atoms
    return {'KL': kl_div(p, q), 'JS': (kl_div(p, m) + kl_div(q, m)) / 2,
            'TV': np.abs(p - q).sum() / 2, 'MMD^2': (p - q) @ K @ (p - q),
            'W1': np.sum(F_gap * np.diff(atoms))}

delta_0, delta_d = np.array([1.0, 0.0]), np.array([0.0, 1.0])  # on atoms (0, d)
for d in (0.01, 1.0, 100.0):
    vals = five_comparisons(np.array([0.0, d]), delta_0, delta_d)
    print(f'd = {d:6}: ' + '  '.join(f'{k} {v:.4g}' for k, v in vals.items()))
```

```
d =   0.01: KL inf  JS 0.6931  TV 1  MMD^2 0.0001  W1 0.01
d =    1.0: KL inf  JS 0.6931  TV 1  MMD^2 0.7869  W1 1
d =  100.0: KL inf  JS 0.6931  TV 1  MMD^2 2  W1 100
```

37.3.1 The f-Divergence Family

Before generalizing KL divergence, let us pin down the minimum properties that any comparison between two distributions should satisfy. The f-divergence family then emerges naturally when we ask which ingredients the non-negativity proof of KL truly uses.

37.3.1.1 Divergences and Metrics

At a minimum, a "distance between distributions" should vanish when the two distributions coincide and stay strictly positive when they differ. A **divergence** is any function of two distributions with this property:

$$D(P, Q) \geq 0, \qquad D(P, Q) = 0 \quad \text{if and only if} \quad P = Q.$$

Nothing else is required. A **metric**, like the Euclidean distance $\|\mathbf{x} - \mathbf{y}\|$ between two vectors (Section 1.3), must additionally be symmetric, $D(P, Q) = D(Q, P)$, and obey the triangle

inequality, $D(P,R) \le D(P,Q) + D(Q,R)$. KL divergence is a divergence: Gibbs' inequality (Section 37.1) guarantees that $D_{\mathrm{KL}}(P\|Q) \ge 0$, with equality only when $P = Q$. It fails both metric axioms, however, and a pair of univariate Gaussians shows both failures immediately. As we computed in that section, the KL divergence between two Gaussians is

$$D_{\mathrm{KL}}\big(\mathcal{N}(\mu_1,\sigma_1^2)\,\big\|\,\mathcal{N}(\mu_2,\sigma_2^2)\big) = \log\frac{\sigma_2}{\sigma_1} + \frac{\sigma_1^2 + (\mu_1-\mu_2)^2}{2\sigma_2^2} - \frac{1}{2}. \tag{37.25}$$

For $P = \mathcal{N}(0,1)$ and $Q = \mathcal{N}(0,4)$, this formula gives $D_{\mathrm{KL}}(P\|Q) \approx 0.318$ nats in one direction and $D_{\mathrm{KL}}(Q\|P) \approx 0.807$ nats in the other, so KL is not symmetric. The triangle inequality fails for a different reason. Take three unit-variance Gaussians centered at 0, 1, and 2: the formula gives $D_{\mathrm{KL}} = \frac{1}{2}$ for each adjacent pair, yet 2 for the outer pair, and $2 > \frac{1}{2} + \frac{1}{2}$. Because KL grows like the *square* of the distance between the means, it behaves like a squared Euclidean distance rather than a true length.

Neither failure needs repair. Minimizing a divergence uses neither symmetry nor the triangle inequality, and the asymmetry of KL turns out to be a genuine modeling choice (Section 37.3.2). Still, whenever a proof needs to chain comparisons from P to Q and from Q to R, the triangle inequality becomes essential. Fortunately, the family we are about to construct includes true metrics as well.

37.3.1.2 The f-Divergence Template

Gibbs' inequality is what makes KL a valid divergence. Let us inspect its proof closely, because any other function that shares the same two properties will yield a divergence too.

First, rewrite KL as an expectation under Q of a function of the density ratio $u(x) = p(x)/q(x)$:

$$D_{\mathrm{KL}}(P\|Q) = \sum_x p(x)\log\frac{p(x)}{q(x)} = \sum_x q(x)\,\frac{p(x)}{q(x)}\log\frac{p(x)}{q(x)} = \mathbb{E}_{x\sim Q}\big[u(x)\log u(x)\big].$$

Viewed this way, KL weights each outcome by Q and penalizes its density ratio through the function $u\log u$. Let us check this formula on our running example from Section 37.1, the discrete pair $P = (0.6, 0.3, 0.1)$ and $Q = (0.2, 0.5, 0.3)$. Here the ratios are $u = 3$, 0.6, and $\frac{1}{3}$, weighted by 0.2, 0.5, and 0.3:

$$\begin{aligned} &0.2\,(3\log 3) + 0.5\,(0.6\log 0.6) + 0.3\,(\tfrac{1}{3}\log\tfrac{1}{3}) \\ &\qquad = 0.659 - 0.153 - 0.110 = 0.396 \text{ nats}, \end{aligned}$$

matching the KL divergence we found earlier.

Second, look at the proof of non-negativity. In Section 37.1, we applied Jensen's inequality once to $-\log(q/p)$ under P. Run instead under Q on the function $u\log u$, that same one-step argument uses only two facts: the function is convex, and it equals zero at $u = 1$. Nothing stops us from replacing $u\log u$ by any function with those two properties. Let $f : (0,\infty) \to \mathbb{R}$ be convex with $f(1) = 0$, and define $f(0) = \lim_{u\downarrow 0} f(u)$, which may be $+\infty$. Assume for now that $p(x) = 0$ wherever $q(x) = 0$, written $P \ll Q$, so that an expectation under Q sees all of P. The **f-divergence** of P with respect to Q is (Csiszár 1967)

$$D_f(P\|Q) = \mathbb{E}_{x\sim Q}\left[f\left(\frac{p(x)}{q(x)}\right)\right]. \tag{37.26}$$

Point by point, the ratio $u = p/q$ measures how much P over- or under-represents x relative to Q, and f assigns a penalty that vanishes when $u = 1$. We call f the *divergence function*. Much of the literature calls it the generator of the divergence, and our code below stores these functions in a dictionary named `generators`. In the prose, however, we reserve the word *generator* for a network that produces samples, as in Section 16.2.

To pin down when the divergence equals zero, we need one mild refinement of convexity: we call f *strictly convex at* 1 if some line through $(1, 0)$ lies strictly below f everywhere else, meaning $f(u) > c\,(u-1)$ for all $u \neq 1$ and some slope c.

Proposition (non-negativity). *For any convex f with $f(1) = 0$ and $P \ll Q$,*

$$D_f(P\|Q) \geq 0,$$

and if f is strictly convex at 1, then $D_f(P\|Q) = 0$ if and only if $P = Q$.

Proof. Jensen's inequality (Section 35.3.3.1) states that $\mathbb{E}[f(U)] \geq f(\mathbb{E}[U])$ for any convex f and random variable U. Apply it to $U = u(x)$ with $x \sim Q$. Everything rests on a single identity: the density ratio averages to one under Q, since $\mathbb{E}_Q[u] = \sum_x q(x)\,p(x)/q(x) = \sum_x p(x) = 1$. Therefore

$$D_f(P\|Q) = \mathbb{E}_Q[f(u)] \geq f(\mathbb{E}_Q[u]) = f(1) = 0.$$

For the equality case, take the supporting line $c\,(u-1)$ lying strictly below f away from 1. Then $D_f(P\|Q) \geq \mathbb{E}_Q[c\,(u-1)] = 0$, and the gap is the expectation of $f(u) - c\,(u-1)$, which is strictly positive wherever $u \neq 1$. Hence $D_f(P\|Q) = 0$ forces $u(x) = 1$ at every x with $q(x) > 0$, meaning $P = Q$. ■

The identity $\mathbb{E}_Q[u] = 1$ that powered this proof has a second handy consequence we will use repeatedly: adding any multiple $c\,(u-1)$ to f leaves every divergence unchanged, because $\mathbb{E}_Q[c\,(u-1)] = 0$. The functions $u \log u$ and $u \log u - u + 1$, for example, define the same KL divergence.

So far we have assumed $P \ll Q$, yet our opening point-mass example violates that condition. A generative model starts training in the same regime, with its samples and the data occupying disjoint regions, so we must extend D_f to handle zeros in q. Outcomes with $p(x) = q(x) = 0$ contribute nothing. Where $p(x) = 0 < q(x)$, the contribution is simply $q(x)\,f(0)$, as the definition already says. Where $q(x) = 0 < p(x)$, the ratio blows up, so we define the contribution as the limit of $q\,f(p/q)$ as $q \downarrow 0$. Substituting $t = p/q$ rewrites this as $p\,f(t)/t$, whose limit is $p(x)\,f'(\infty)$ with $f'(\infty) := \lim_{t\to\infty} f(t)/t$. For a convex function f, the secant $f(t)/t$ and the derivative $f'(t)$ share the same limit at infinity, which explains the notation. Either $f(0)$ or $f'(\infty)$ may be $+\infty$.

Now apply this rule to the point masses δ_0 and δ_d for any $d \neq 0$. At $x = 0$ we have $q = 0 < p = 1$, and at $x = d$ we have $p = 0 < q = 1$, so

$$D_f(\delta_0\|\delta_d) = f(0) + f'(\infty) \qquad \text{for every } d \neq 0.$$

Notice that d has vanished from the right-hand side. This tells us something far stronger than Table 37.2 alone showed: *every* f-divergence is constant on disjoint supports. It cannot distinguish a near miss from a distant one, so a model trained on it gets zero gradient from the distance separating the supports. For KL, $f(u) = u \log u$ gives $f(0) = 0$ and $f'(\infty) = \lim_{t\to\infty} \log t = \infty$, reproducing the first row of Table 37.2. The other two frozen rows follow as soon as we write down their divergence functions.

37.3.1.3 Common Divergence Functions

Each row of Table 37.3 corresponds to one choice of f, and Figure 37.7 plots six of these curves side by side. The column $f''(1)$, recording the curvature at $u = 1$, will play a central role in the next subsection.

Table 37.3: Common divergence functions. Each row gives the function f of one divergence, its curvature at $u = 1$, whether the divergence is symmetric, and whether it (or its square root) is a metric.

Divergence	$f(u)$	$f''(1)$	Symmetric?	Metric?
Kullback–Leibler (forward)	$u \log u$	1	no	no
reverse KL	$-\log u$	1	no	no
Pearson χ^2	$(u-1)^2$	2	no	no
squared Hellinger d_{H}^2	$(\sqrt{u}-1)^2$	$\frac{1}{2}$	yes	d_{H} is a metric
total variation	$\frac{1}{2}\lvert u-1\rvert$	none (corner)	yes	yes
Jensen–Shannon	$\frac{u}{2}\log u - \frac{u+1}{2}\log\frac{u+1}{2}$	$\frac{1}{4}$	yes	$\sqrt{\mathrm{JS}}$ is a metric
α-divergence ($\alpha \neq 0, 1$)	$\dfrac{u^{\alpha} - \alpha u + \alpha - 1}{\alpha(\alpha-1)}$	1	only at $\alpha = \frac{1}{2}$	square root at $\alpha = \frac{1}{2}$

Reverse KL uses $f(u) = -\log u$, because $\mathbb{E}_Q[-\log(p/q)] = \mathbb{E}_Q[\log(q/p)] = D_{\mathrm{KL}}(Q\|P)$. Swapping the arguments of KL is therefore just another choice of f, so both directions sit side by side in the same table. Exercise 1a gives the general recipe for swapping the arguments of any f-divergence.

Total variation uses $f(u) = \frac{1}{2}|u-1|$, which simplifies to $\sum_x q(x)\,\frac{1}{2}|p(x)/q(x) - 1| = \frac{1}{2}\sum_x |p(x) - q(x)|$. We write it as $\mathrm{TV}(P, Q)$. It is half the ℓ_1 distance between the probability vectors; on our running pair, $\mathrm{TV} = \frac{1}{2}(0.4 + 0.2 + 0.2) = 0.4$. We will unpack what this number means for distinguishing P from Q in Section 37.3.3. On the point masses, its divergence function gives $f(0) + f'(\infty) = \frac{1}{2} + \frac{1}{2} = 1$, accounting for the third row of Table 37.2.

Pearson's χ^2 divergence uses the parabola $f(u) = (u-1)^2$, giving $\chi^2(P\|Q) = \sum_x q(x)\,(p(x)/q(x) - 1)^2 = \sum_x (p(x) - q(x))^2/q(x)$. On the running pair, it evaluates to $0.16/0.2 + 0.04/0.5 + 0.04/0.3 = 1.0133$. Notice how heavily a mismatch is weighted where Q is small: the first outcome alone, where $q = 0.2$, contributes 0.8 of the total.

The **Jensen–Shannon divergence** (Lin 1991) symmetrizes KL by comparing both distributions against their even mixture $M = \frac{1}{2}(P + Q)$ (with mass function m):

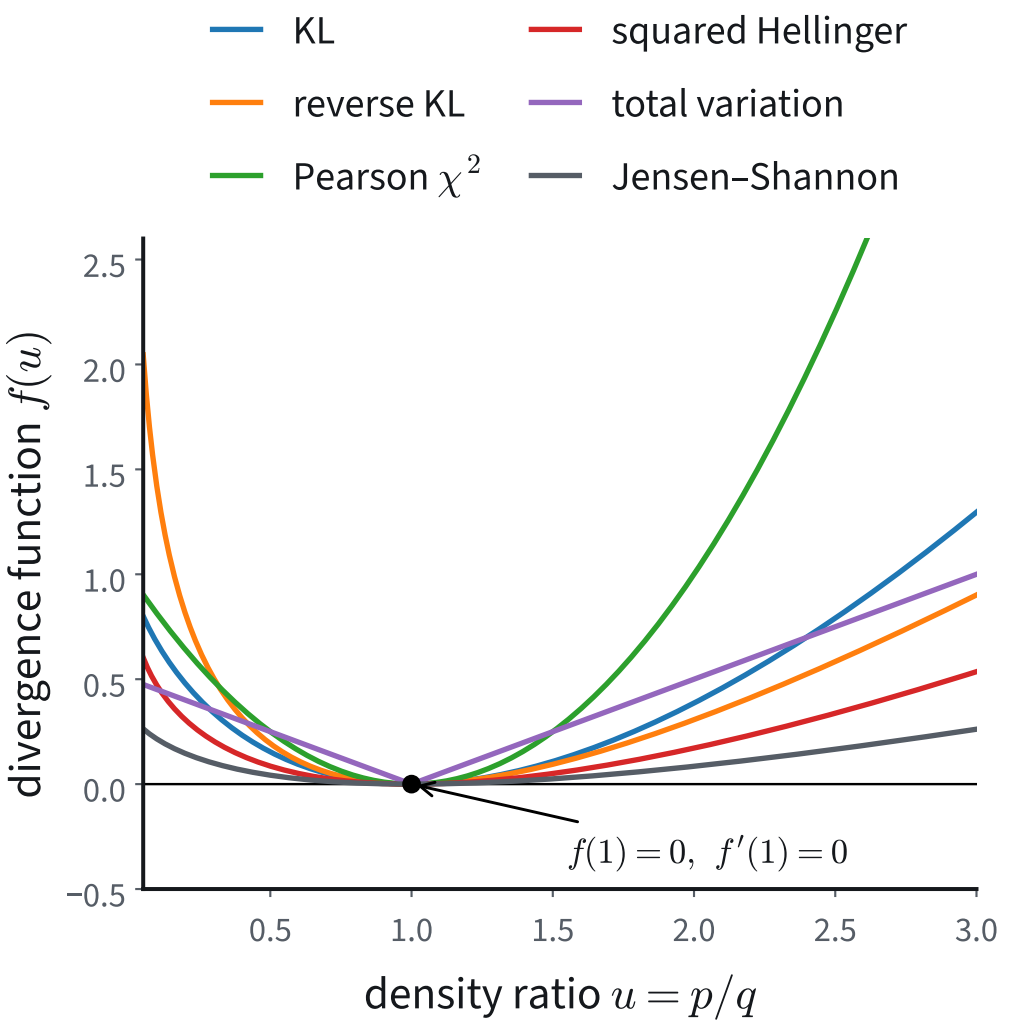


Figure 37.7: Six divergence functions, with the smooth curves shifted by $f'(1)(u-1)$ so that each touches zero tangentially at $u = 1$ without changing its divergence. Near $u = 1$ the smooth curves are parabolas with the curvatures $1, 1, 2, \frac{1}{2}, \frac{1}{4}$ listed in the table, while total variation has a sharp corner instead. Away from $u = 1$ they penalize disagreement very differently: reverse KL blows up as $u \to 0$, KL and χ^2 grow superlinearly, and total variation grows linearly.

$$\mathrm{JS}(P,Q) = \frac{1}{2}D_{\mathrm{KL}}(P \,\|\, M) + \frac{1}{2}D_{\mathrm{KL}}(Q \,\|\, M). \tag{37.27}$$

Because M puts mass wherever either P or Q does, the ratios p/m and q/m can never blow up: $p/m = 2p/(p+q) \le 2$, so each KL term is at most $\log 2$. Hence JS is symmetric and bounded, $0 \le \mathrm{JS}(P,Q) \le \log 2$, and it hits its ceiling $\log 2$ only when P and Q have disjoint supports (where $p/m = 2$ wherever $p > 0$). Remarkably, its square root is a true metric (Endres and Schindelin 2003). Writing Equation 37.27 in terms of the ratio u yields the divergence function in the table (Exercise 1c), which satisfies $f(0) = f'(\infty) = \frac{1}{2}\log 2$. Their sum is $\log 2$, the second row of Table 37.2. Our single identity $D_f(\delta_0\|\delta_d) = f(0) + f'(\infty)$ has now explained all three frozen rows.

The **squared Hellinger divergence** is $d_{\mathrm{H}}^2(P,Q) = \sum_x \left(\sqrt{p(x)} - \sqrt{q(x)}\right)^2$. Its square root d_{H} is simply the Euclidean distance between the vectors $\sqrt{p}$ and $\sqrt{q}$, which immediately makes d_{H} a metric (Exercise 2b).

The final row of the table is a continuous family inside the family, the α-**divergences** (Amari 2016), generated by

$$f_\alpha(u) = \frac{u^\alpha - \alpha u + \alpha - 1}{\alpha(\alpha - 1)}. \tag{37.28}$$

For every $\alpha \neq 0, 1$, this function is convex with $f_\alpha(1) = 0$ and curvature $f_\alpha''(1) = 1$. As $\alpha \to 1$ it approaches $u\log u - u + 1$, and as $\alpha \to 0$ it approaches $-\log u + u - 1$. Since adding a multiple of $u-1$ changes no divergence, the two limits recover forward and reverse KL. Halfway between them, at $\alpha = \frac{1}{2}$, we get $2(\sqrt{u} - 1)^2$ (twice squared Hellinger), while $\alpha = 2$ gives half of χ^2. A single parameter α therefore interpolates smoothly from reverse KL through Hellinger to forward KL and on to χ^2 (Exercise 1d connects this family to Rényi's divergence).

Let us evaluate the first six rows on our running pair in both argument orders. KL should print 0.396 and 0.365 nats, matching Section 37.1, while TV and $\chi^2(P\|Q)$ should match the hand calculations above.

```
def f_divergence(f, p, q):
    """D_f(P||Q) = sum_x q(x) f(p(x)/q(x)); nats for KL/JS, else unitless."""
    return np.sum(q * f(p / q))

generators = {
    'KL':          lambda u: u * np.log(u),
    'reverse KL':  lambda u: -np.log(u),
    'chi^2':       lambda u: (u - 1) ** 2,
    'Hellinger^2': lambda u: (np.sqrt(u) - 1) ** 2,
    'TV':          lambda u: 0.5 * np.abs(u - 1),
    'JS':          lambda u: u / 2 * np.log(u) - (u + 1) / 2 * np.log((u + 1) / 2),
}

p = np.array([0.6, 0.3, 0.1])
q = np.array([0.2, 0.5, 0.3])
for name, f in generators.items():
```

```
    print(f'{name:12s} D_f(P||Q) = {f_divergence(f, p, q):.4f}   '
          f'D_f(Q||P) = {f_divergence(f, q, p):.4f}')
```

```
KL           D_f(P||Q) = 0.3961   D_f(Q||P) = 0.3653
reverse KL   D_f(P||Q) = 0.3653   D_f(Q||P) = 0.3961
chi^2        D_f(P||Q) = 1.0133   D_f(Q||P) = 0.8000
Hellinger^2  D_f(P||Q) = 0.1862   D_f(Q||P) = 0.1862
TV           D_f(P||Q) = 0.4000   D_f(Q||P) = 0.4000
JS           D_f(P||Q) = 0.0911   D_f(Q||P) = 0.0911
```

As expected, the KL row is asymmetric, and the reverse-KL row is simply the KL row with its columns swapped. The χ^2 row is asymmetric too (1.0133 against 0.8000). By contrast, squared Hellinger (0.1862), TV (0.4), and JS are symmetric, with JS at 0.0911 nats sitting well below its ceiling of $\log 2 \approx 0.693$. Six divergences have looked at the same pair of distributions and returned values spanning an order of magnitude, from 0.09 to 1.01. Yet if we multiply KL by 2, squared Hellinger by 4, and JS by 8, the three rescaled values are 0.79, 0.74, and 0.73. Why would three quite different functions of the density ratio nearly agree, and where did the factors 2, 4, and 8 come from?

37.3.1.4 The Local Quadratic

The curves in Figure 37.7 behave very differently far from $u = 1$. Near $u = 1$, however, they collapse into a single function up to a constant factor, which is perhaps the most surprising property of the entire family. Nothing later in this section depends on this subsection, so you can skip it on a first reading.

Because adding a linear term $c\,(u-1)$ leaves every divergence unchanged, neither the value $f(1) = 0$ nor the slope $f'(1)$ affects D_f. The first property that distinguishes one smooth divergence function from another is its curvature $f''(1)$. Near $u = 1$, every smooth f looks like the parabola $\frac{1}{2}f''(1)(u-1)^2$, and the expectation of $(u-1)^2$ under Q is none other than the χ^2 divergence, $\mathbb{E}_Q[(u-1)^2] = \sum_x (p(x)-q(x))^2/q(x)$. Let $r = \max_x |u(x)-1|$ measure how far the density ratio strays from 1.

Proposition (local quadratic). *Let $P \ll Q$, let $0 < r < 1$, let f be three times differentiable on $[1-r, 1+r]$ with $|f'''| \le B$ there, and let $|u(x)-1| \le r$ for every x with $q(x) > 0$. Then*

$$\left| D_f(P\|Q) - \frac{f''(1)}{2}\,\chi^2(P\|Q) \right| \le \frac{Br}{6}\,\chi^2(P\|Q). \tag{37.29}$$

Proof. Since $f(1) = 0$, Taylor's theorem with remainder gives $f(u) = f'(1)(u-1) + \frac{1}{2}f''(1)(u-1)^2 + R(u)$ with $|R(u)| \le B|u-1|^3/6$. Take $\mathbb{E}_Q$ term by term. The linear term vanishes because $\mathbb{E}_Q[u-1] = 0$, eliminating the only first-order difference between one f and another. The quadratic term becomes $\frac{1}{2}f''(1)\,\chi^2(P\|Q)$, and the remainder satisfies $\mathbb{E}_Q|R| \le \frac{B}{6}\,\mathbb{E}_Q|u-1|^3 \le \frac{Br}{6}\,\mathbb{E}_Q(u-1)^2 = \frac{Br}{6}\,\chi^2$. ∎

In short, $D_f \approx \frac{1}{2}f''(1)\,\chi^2$ with a relative error proportional to r: whenever P is close to Q, all smooth f-divergences are scalar multiples of one another. The factors 2, 4, and 8 we applied to KL, squared Hellinger, and JS are simply $2/f''(1)$, and the target all three products approximate is $\chi^2(P\|Q) = 1.013$. On our running pair they still fall about a quarter short, which is no surprise: the largest ratio is 3, so $r = 2$, outside the proposition's

hypothesis $r < 1$. To watch the quadratic approximation tighten, let us slide P toward Q along the line segment $P_t = Q + t\,(P - Q)$. This scales every $u - 1$ by t, giving $r = 2t$. For four smooth divergences, we print the ratio $D_f(P_t\|Q)/(\frac{1}{2}f''(1)\,\chi^2(P_t\|Q))$, which should converge to 1 at a rate proportional to t.

```
curvature = {'KL': 1.0, 'reverse KL': 1.0, 'Hellinger^2': 0.5, 'JS': 0.25}
for t in (0.5, 0.1, 0.01):
    p_t = q + t * (p - q)                    # moves P_t toward Q as t -> 0
    chi2 = f_divergence(generators['chi^2'], p_t, q)
    ratios = [f_divergence(generators[name], p_t, q) / (c / 2 * chi2)
              for name, c in curvature.items()]
    print(f't = {t:4}: ' + '  '.join(f'{name} {r:.3f}'
                                     for name, r in zip(curvature, ratios)))
```

```
t =  0.5: KL 0.844  reverse KL 0.747  Hellinger^2 0.789  JS 0.783
t =  0.1: KL 0.956  reverse KL 0.917  Hellinger^2 0.936  JS 0.935
t = 0.01: KL 0.995  reverse KL 0.990  Hellinger^2 0.993  JS 0.993
```

At $t = 0.5$, the four divergences sit at roughly 75% to 85% of their quadratic approximations; at $t = 0.1$ they exceed 91%, and at $t = 0.01$ they exceed 99%. Between the last two rows t drops tenfold and each shortfall drops nearly tenfold too, matching the linear factor r in the proposition. The bound itself is conservative: for KL at $t = 0.1$ we have $r = 0.2$ and $|f'''(u)| = 1/u^2 \le 1/0.8^2$, which allows a relative error of about 10%, whereas the actual gap is only 4%. The takeaway for optimization is practical: smooth f-divergences disagree meaningfully only on distant distributions, which is the regime early in training when the model is still far from the data.

Inside a parametric family, this common local quadratic is governed by the Fisher information matrix $\mathbf{I}(\theta) = \mathbb{E}\big[\nabla_\theta \log q_\theta \, \nabla_\theta \log q_\theta^\top\big]$ of Section 36.6. For a small parameter step $\mathbf{h}$, the density ratio of $q_{\theta+\mathbf{h}}$ to q_θ expands as $u \approx 1 + \mathbf{h}^\top \nabla_\theta \log q_\theta$. Therefore $\chi^2(q_{\theta+\mathbf{h}}\|q_\theta) = \mathbb{E}[(u-1)^2] \approx \mathbf{h}^\top \mathbf{I}(\theta)\,\mathbf{h}$, so every smooth D_f has Hessian $f''(1)\,\mathbf{I}(\theta)$ at $\mathbf{h} = \mathbf{0}$. The natural gradient of Section 35.5, though usually motivated by KL divergence, would yield the same update under any smooth f-divergence. Total variation is the lone exception in our table: its divergence function has a corner at $u = 1$ and no second derivative there, and Exercise 4b explores how that changes its behavior as P approaches Q.

37.3.2 Forward and Reverse KL

Maximum likelihood, the training criterion behind most models in this book, minimizes KL divergence with the data in the first argument. Variational inference minimizes KL divergence with the model in the first argument. Both orders appear as separate rows in Table 37.3, and when we fit a model Q_θ (with density q_θ) to a target P that lies outside the model family, they push the model in opposite directions:

- **Forward KL**, $D_{\mathrm{KL}}(P\,\|\,Q_\theta) = \mathbb{E}_P[\log(p/q_\theta)]$, takes its expectation over the *truth*. Whenever $q_\theta(x) \to 0$ at a point where $p(x) > 0$, the divergence blows up. Forward KL is therefore *zero-avoiding*, or *mode-covering*: the model must stretch to cover every mode of the data, that is, every region where p concentrates, even if that forces it to spill mass across empty regions P never visits.
- **Reverse KL**, $D_{\mathrm{KL}}(Q_\theta\,\|\,P) = \mathbb{E}_{Q_\theta}[\log(q_\theta/p)]$, takes its expectation over the *model*. Here the model is penalized heavily for placing mass where the truth has none, while

any mode the model ignores contributes nothing to the objective. Reverse KL is therefore *zero-forcing*, or *mode-seeking*: the model locks onto the portion of the target it can explain well.

In practice, the choice of direction is often settled by what we can actually compute. Forward KL requires samples from P and the density of the model, and minimizing it is just maximum likelihood: Section 36.6.2.1 shows that the average negative log-likelihood equals the cross-entropy from the empirical distribution to the model, which differs from forward KL only by a constant independent of θ. Reverse KL requires samples from the model and the density of the target, and it works even when the target density is known only up to a normalizing constant. Variational inference (Section 36.7) sits in this setting: its target P is a posterior with an intractable normalizer, so it minimizes $D_{\mathrm{KL}}(Q_\theta \| P)$ over tractable approximations Q_θ.

When samples and densities are available on both sides, however, the direction becomes a genuine design choice. A student network distilled from a teacher (Section 26.1) spreads out to cover everything the teacher might say when the teacher comes first, and concentrates on the teacher's likeliest outputs when the student comes first. Similarly, the KL penalty in Section 15.3 places the learned policy first, keeping it away from actions that the reference policy never takes.

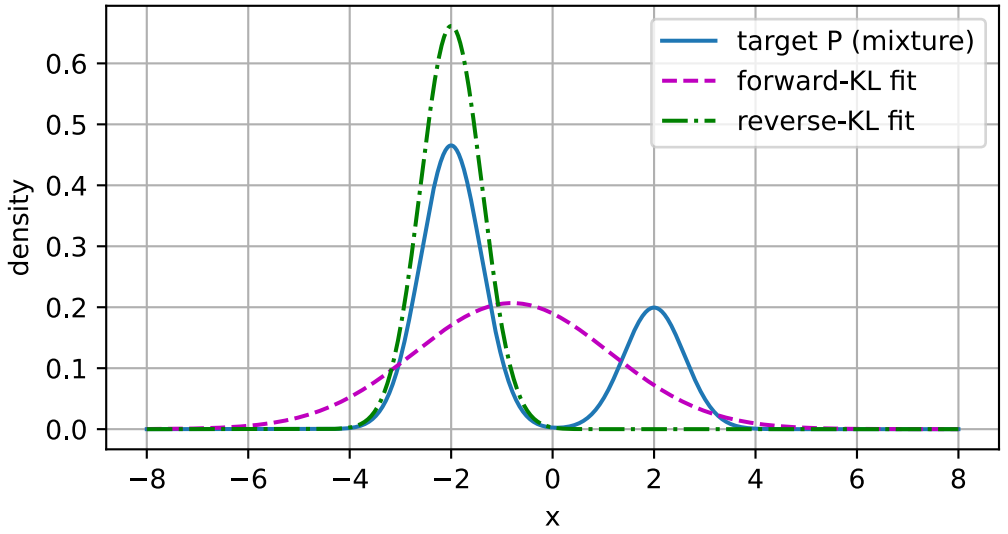
If the model family contains P, both directions recover P. They part ways when the model is *misspecified*, and the clearest illustration is fitting a single Gaussian to a bimodal mixture. For forward KL, no optimization is even needed. Because forward KL equals a constant minus $\mathbb{E}_P[\log q(X)]$, its minimizer is the maximum-likelihood Gaussian with P replacing the empirical distribution, and maximum likelihood in an exponential family matches the expected sufficient statistics (Section 36.4.3.2): $\mu^* = \mathbb{E}_P[X]$ and $(\sigma^*)^2 = \mathrm{Var}_P(X)$, provided the variance is finite. Reverse KL has no such closed-form solution, and it can have a separate local minimum near each mode.

For the bimodal mixture $P = 0.7\,\mathcal{N}(-2, 0.6^2) + 0.3\,\mathcal{N}(2, 0.6^2)$, we can predict the forward fit in advance. Its mean is $0.7(-2) + 0.3(2) = -0.8$. Its variance is the within-component variance plus the between-component variance, which for weights w_1, w_2 and means separated by Δ equals $w_1 w_2 \Delta^2$, giving $0.36 + 0.7 \cdot 0.3 \cdot 4^2 = 3.72$. For the reverse fit, we run a derivative-free optimizer over $\theta = (\mu, \log \sigma)$ from two starting points, once at the origin and once near the smaller mode, evaluating each KL by numerical quadrature on a grid. For each fit, the next cell reports the mean, standard deviation, and both divergences.

```
w, mus, sigmas = np.array([0.7, 0.3]), np.array([-2.0, 2.0]), np.array([0.6, 0.6])
x = np.linspace(-8.0, 8.0, 4001)

def log_components(x, w, mus, sigmas):
    """log(w_k N(x; mu_k, sigma_k^2)) for every component k; shape (k, len(x))."""
    return (np.log(w)[:, None] - 0.5 * np.log(2 * np.pi * sigmas[:, None] ** 2)
            - (x - mus[:, None]) ** 2 / (2 * sigmas[:, None] ** 2))

def log_mix(x, w, mus, sigmas):
    """Log-density of the mixture sum_k w_k N(mu_k, sigma_k^2), computed stably."""
    c = log_components(x, w, mus, sigmas)
    c_max = c.max(axis=0)
    return c_max + np.log(np.exp(c - c_max).sum(axis=0))
```

```
    log_p = lambda x: log_mix(x, w, mus, sigmas)

    def log_q(x, theta):
        mu, log_sigma = theta
        return (-0.5 * np.log(2 * np.pi) - log_sigma
                - (x - mu) ** 2 / (2 * np.exp(2 * log_sigma)))

    def kl(log_a, log_b, x):
        """D_KL(A||B) by quadrature on the grid x; in nats."""
        return integrate.trapezoid(np.exp(log_a) * (log_a - log_b), x)

    fwd = lambda th: kl(log_p(x), log_q(x, th), x)     # D_KL(P || Q_theta)
    rev = lambda th: kl(log_q(x, th), log_p(x), x)     # D_KL(Q_theta || P)

    th_fwd = optimize.minimize(fwd, x0=[0.0, 0.0], method='Nelder-Mead').x
    th_rev = optimize.minimize(rev, x0=[0.0, 0.0], method='Nelder-Mead').x
    th_rev2 = optimize.minimize(rev, x0=[2.5, -0.5], method='Nelder-Mead').x

    print(f'{"fit":26s}{"mu":>8s}{"sigma":>7s}'
          f'{"forward KL":>12s}{"reverse KL":>12s}')
    for name, th in [('forward-KL fit', th_fwd), ('reverse-KL fit', th_rev),
                     ('reverse-KL fit, 2nd start', th_rev2)]:
        print(f'{name:26s}{th[0]:+8.3f}{np.exp(th[1]):7.3f}'
              f'{fwd(th):12.3f}{rev(th):12.3f}')

    d2l.plot(x, [np.exp(log_p(x)), np.exp(log_q(x, th_fwd)),
                 np.exp(log_q(x, th_rev))], 'x', 'density',
             legend=['target P (mixture)', 'forward-KL fit', 'reverse-KL fit'],
             figsize=(6, 3))
```

```
fit                            mu  sigma  forward KL  reverse KL
forward-KL fit             -0.800  1.929       0.558       1.274
reverse-KL fit             -1.998  0.603       5.981       0.356
reverse-KL fit, 2nd start  +1.995  0.607      14.534       1.202
```



The forward fit matches our calculation: $\mu = -0.800$ and $\sigma = 1.929 \approx \sqrt{3.72}$. It is a broad bell curve draped over both modes, putting substantial mass in the valley between them where P has almost none. The first reverse fit lands at $\mu = -1.998$, $\sigma = 0.603$: it hugs the dominant component and leaves almost no mass on the minor one. Its reverse KL, 0.356 nats, is close to $\log(1/0.7) \approx 0.357$, the cost of treating the 70% component as if it were the entire distribution. When started near the minor mode, the optimizer settles instead on $\mu = +1.995$, with reverse KL 1.202 nats, close to $\log(1/0.3) \approx 1.204$. Which mode a variational approximation captures therefore depends on its initialization.

Because reverse KL averages under the model, any mode the model fails to visit is never inspected. The objective only notices that the chosen mode carries weight 0.7 (or 0.3) instead of 1. That is why variational posteriors notoriously underestimate uncertainty: here the reverse fit has standard deviation 0.60, whereas the target's is 1.93.

The last two columns highlight how sharply the two criteria disagree on the same model. Forward KL, the maximum-likelihood criterion, penalizes the dominant-mode fit at 5.98 nats (more than ten times the 0.558 nats of the broad fit) and the minor-mode fit at 14.5 nats. In both cases, the dropped mode sits more than six of the model's standard deviations away from its mean, where $\log q$ is deeply negative (Exercise 3c). Reverse KL ranks the first two fits in the opposite order, 0.356 nats against 1.27: what it cannot forgive is the broad fit's mass sitting in the empty valley between the modes.

Our boundary rules for vanishing densities capture this contrast in the limit. For forward KL, $f(u) = u \log u$ has $f'(\infty) = \infty$, so placing zero model mass where P has positive mass costs infinity, while $f(0) = 0$ means placing model mass where P has none adds no boundary term at all. For reverse KL, $f(u) = -\log u$, those two values swap places.

37.3.3 Total Variation, Pinsker's Inequality, and Data Processing

Knowing that a divergence equals 0.4 tells us very little on its own until we know what that number actually guarantees. For total variation, the operational meaning is completely transparent, and Pinsker's inequality passes much of that meaning on to KL divergence.

37.3.3.1 Total Variation

The divergence function $\frac{1}{2}|u-1|$ is the simplest entry in Table 37.3: it penalizes only the absolute distance of the ratio from 1. Its divergence, half the ℓ_1 distance between p and q, admits a second, equivalent form that makes the number immediately interpretable. Total variation is the largest gap between the probabilities that P and Q assign to any event A:

$$\mathrm{TV}(P,Q) = \frac{1}{2}\sum_x |p(x)-q(x)| = \sup_A |P(A)-Q(A)|. \tag{37.30}$$

If $\mathrm{TV}(P,Q) = 0.03$, no event whatsoever can separate the two distributions by more than three percentage points. The supremum is achieved by the event $A^\star = \{x : p(x) > q(x)\}$ of outcomes that P favors over Q. To verify this, pick any event A. Then $P(A) - Q(A) = \sum_{x\in A}(p(x)-q(x)) \le \sum_{x\in A^\star}(p(x)-q(x))$, because switching from A to $A^\star$ only adds non-negative terms and drops non-positive ones. Since $\sum_x (p(x)-q(x)) = 0$, the positive and negative parts of $p-q$ have equal total magnitude, so the right-hand side equals $\frac{1}{2}\sum_x |p(x)-q(x)|$; the same bound applies to $Q(A)-P(A)$.

On our running pair, $A^\star$ consists of the first outcome alone, giving $\mathrm{TV} = 0.6 - 0.2 = 0.4$, as we found earlier. Figure 37.8 illustrates the same geometry for two continuous densities. Total variation is also a true metric: both forms are manifestly symmetric, and the triangle inequality follows directly from the ℓ_1 formula (Exercise 2a).

The supremum form gives TV a concrete statistical meaning. Imagine handing a tester a single sample, drawn with equal probability $\frac{1}{2}$ from P or from Q, and asking which distribution generated it. Any deterministic test answers "P" when the sample falls in some region A, and "Q" otherwise, succeeding with probability

$$\tfrac{1}{2}P(A) + \tfrac{1}{2}(1-Q(A)) = \tfrac{1}{2}(1+P(A)-Q(A)) \le \tfrac{1}{2}(1+\mathrm{TV}(P,Q)),$$

with equality at $A = A^\star$. Every test takes this form for some A (a randomized test merely averages deterministic ones and cannot do better), so even the optimal tester beats a fair coin flip by at most $\mathrm{TV}/2$. On our running pair, the optimal test answers "P" on the first outcome and is right 70% of the time. Both the discriminator of a GAN (trying to tell real data from generated samples) and a detector of machine-generated text are testers of this kind, and total variation sets a hard ceiling on how accurate either can be from a single sample.

37.3.3.2 Pinsker's Inequality

Total variation bounds every statistical test, while KL divergence is what we can usually compute or control in practice because neural network losses are built from log-probabilities. Pinsker's inequality bridges the two: a small KL divergence certifies that no test can reliably tell P from Q. Policy trust regions in Section 15.2 are a classic example: the monotonic improvement guarantee is proved in terms of total variation, yet enforced through a bound on KL.

Proposition (Pinsker's inequality). (Pinsker 1964; Csiszár 1967) *For distributions P and Q on a finite or countable set,*

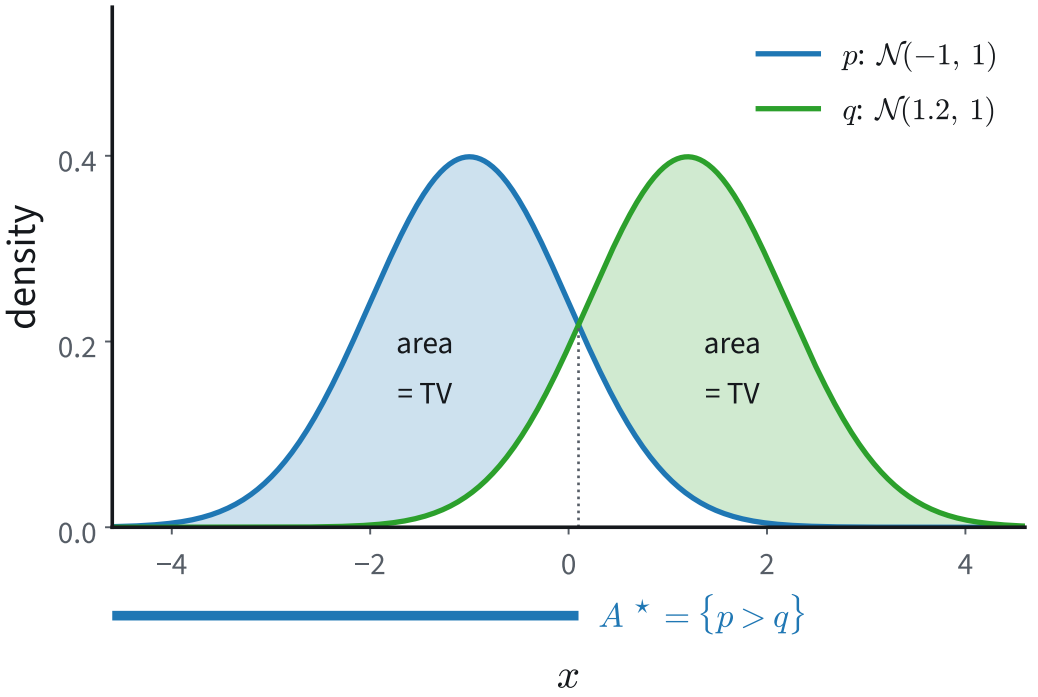


Figure 37.8: Two equal-variance densities and the regions where they disagree. The lobe where $p > q$ and the lobe where $q > p$ have equal area, and that common area is the total variation distance. The event attaining the supremum is $A^\star = \{p > q\}$, here the half-line left of the crossing: guessing P there and Q elsewhere is the optimal single-sample test.

$$\mathrm{TV}(P,Q) \;\le\; \sqrt{\tfrac{1}{2}\, D_{\mathrm{KL}}(P\|Q)}. \tag{37.31}$$

Proof. *Step 1: two outcomes.* Write $\mathrm{Ber}(a)$ for a coin with heads probability a. For $a \in [0,1]$ and $b \in (0,1)$, we have $D_{\mathrm{KL}}(\mathrm{Ber}(a)\|\mathrm{Ber}(b)) = a\log\frac{a}{b} + (1-a)\log\frac{1-a}{1-b}$ and $\mathrm{TV}(\mathrm{Ber}(a),\mathrm{Ber}(b)) = |a-b|$. We therefore need to show that $\rho(a) = D_{\mathrm{KL}}(\mathrm{Ber}(a)\|\mathrm{Ber}(b)) - 2(a-b)^2$ is non-negative. Fix b. Clearly $\rho(b)=0$, the first derivative $\rho'(a) = \log\frac{a}{b} - \log\frac{1-a}{1-b} - 4(a-b)$ also vanishes at $a=b$, and for $0<a<1$ the second derivative is

$$\rho''(a) = \frac{1}{a(1-a)} - 4 \ge 0,$$

because $a(1-a) \le \tfrac{1}{4}$. A convex function with both value and slope zero at $a=b$ is non-negative everywhere on $(0,1)$, and by continuity (with $0\log 0 = 0$) at the endpoints $a=0$ and $a=1$ as well. If b is 0 or 1, the KL divergence is infinite unless $a=b$, so the inequality holds trivially.

Step 2: reduction to two outcomes. Let $A^\star = \{p > q\}$, and set $a = P(A^\star)$ and $b = Q(A^\star)$, so that $\mathrm{TV}(P,Q) = a - b$. Coarsening the space by merging all outcomes inside $A^\star$ into one bucket and all outcomes outside $A^\star$ into another can only *decrease* KL. That follows from the log-sum inequality: for non-negative numbers v_i and w_i,

$$\sum_i v_i \log\frac{v_i}{w_i} \;\ge\; \Big(\sum_i v_i\Big)\log\frac{\sum_i v_i}{\sum_i w_i},$$

which is itself one application of Jensen's inequality to $t \mapsto t\log t$. Applying it separately to $A^\star$ and its complement gives

$$D_{\mathrm{KL}}(P\|Q) \;\ge\; D_{\mathrm{KL}}(\mathrm{Ber}(a)\|\mathrm{Ber}(b)) \;\ge\; 2(a-b)^2 = 2\,\mathrm{TV}(P,Q)^2,$$

which rearranges to Equation 37.31. ∎

On our running pair, Pinsker's inequality reads $0.4 \le \sqrt{0.396/2} = 0.445$, and we can trace Step 2 by hand. Merging the second and third outcomes collapses the pair to two coins, $(0.6, 0.4)$ and $(0.2, 0.8)$. Their TV distance remains 0.4, their KL divergence drops from 0.396 to $0.6\log 3 + 0.4\log\frac{1}{2} = 0.382$ nats, and Step 1 lower bounds that by $2\,(0.4)^2 = 0.32$.

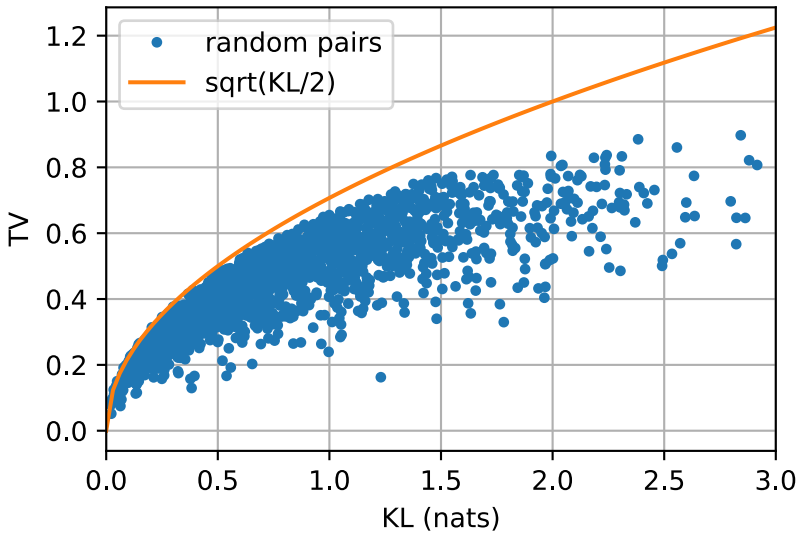
For a coin of bias 0.6 compared with a fair coin, KL is $0.6\log 1.2 + 0.4\log 0.8 = 0.0201$ nats, while $2\,(0.1)^2 = 0.02$: for nearly equal coins, Pinsker's inequality is almost tight. Crucially, however, there is no useful converse. A model that puts $q = 0$ on a single rare outcome has infinite KL even when its TV distance is tiny. The bound also becomes vacuous for well-separated distributions, since TV can never exceed 1: two unit-variance Gaussians separated by 50 have TV ≈ 1, whereas Equation 37.25 gives $\mathrm{KL} = 50^2/2 = 1250$ nats.

How tight is Pinsker's bound on typical distributions, and can the constant $\frac{1}{2}$ be improved? To check the first question, we sample $10{,}000$ random pairs of distributions on five outcomes uniformly from the simplex. To check the second, we let two coins approach each

other at $a = b = \frac{1}{2}$: Step 1 lost nothing to second order there, so the ratio $\mathrm{TV}/\sqrt{\mathrm{KL}/2}$ should climb all the way to 1.

```
rng = np.random.default_rng()
pr, qr = rng.dirichlet(np.ones(5), 10000), rng.dirichlet(np.ones(5), 10000)
tv = 0.5 * np.abs(pr - qr).sum(axis=1)
kl_pq = np.sum(pr * np.log(pr / qr), axis=1)
ratio = tv / np.sqrt(kl_pq / 2)
print(f'TV / sqrt(KL/2) over 10,000 random pairs: median {np.median(ratio):.2f}, '
      f'max {ratio.max():.3f}')
for delta in (0.1, 0.01, 0.001):
    a, b = np.array([0.5, 0.5]), np.array([0.5 + delta, 0.5 - delta])
    print(f'coins 1/2 vs 1/2 + {delta:5}: TV / sqrt(KL/2) = '
          f'{delta / np.sqrt(kl_div(a, b) / 2):.6f}')
grid = np.linspace(0, 3, 100)
d2l.plot([kl_pq[:2000], grid], [tv[:2000], np.sqrt(grid / 2)], 'KL (nats)',
         'TV', legend=['random pairs', 'sqrt(KL/2)'], fmts=('.', '-'),
         xlim=[0, 3], figsize=(4.5, 3))
```

```
TV / sqrt(KL/2) over 10,000 random pairs: median 0.78, max 0.986
coins 1/2 vs 1/2 +   0.1: TV / sqrt(KL/2) = 0.989881
coins 1/2 vs 1/2 +  0.01: TV / sqrt(KL/2) = 0.999900
coins 1/2 vs 1/2 + 0.001: TV / sqrt(KL/2) = 0.999999
```



Every point in the scatter plot falls beneath the curve $\sqrt{\mathrm{KL}/2}$. The median pair sits near 0.8 of the bound, and the tightest of the $10{,}000$ random pairs comes within a few hundredths of 1. Once $\mathrm{KL} > 2$, the curve rises above 1 and the bound becomes vacuous. For nearly fair coins, the ratio climbs through 0.9899, 0.9999, and 0.999999, confirming that the constant $\frac{1}{2}$ cannot be tightened further (Exercise 4c). Pinsker's original proof had a weaker constant; the sharp factor $\frac{1}{2}$ is due to Csiszár (1967) and, independently, to Kullback and Kemperman.

37.3.3.3 Data Processing

Can a feature extractor, a quantizer, or any other post-processing step make two distributions easier to tell apart? In Pinsker's proof, merging outcomes could only decrease KL, and the chain rule of Section 37.1.2.4 showed the same for discarding one variable of a pair. The same principle holds for every f-divergence and every form of processing. A *channel* $\kappa(y \mid x)$ is any rule, deterministic or randomized, that transforms an input x into an output y. Passing P through the channel produces the output distribution $(P\kappa)(y) = \sum_x \kappa(y \mid x)\, p(x)$, and passing Q produces $Q\kappa$ analogously.

Proposition (data processing for f-divergences). *For any channel κ, convex f with $f(1) = 0$, and $P \ll Q$ on a finite or countable set,*

$$D_f(P\kappa \,\|\, Q\kappa) \;\le\; D_f(P\|Q). \tag{37.32}$$

Proof. Since $\sum_y \kappa(y \mid x) = 1$, we can expand $D_f(P\|Q) = \sum_y \sum_x \kappa(y \mid x)\, q(x)\, f(p(x)/q(x))$. Fix an output y with $(Q\kappa)(y) > 0$. The normalized weights $\kappa(y \mid x)\, q(x)/(Q\kappa)(y)$ sum to one over x, and averaging the density ratios $p(x)/q(x)$ under these weights yields $(P\kappa)(y)/(Q\kappa)(y)$. Applying Jensen's inequality to the inner sum therefore bounds it below by $(Q\kappa)(y)\, f((P\kappa)(y)/(Q\kappa)(y))$, which is the y-th term of $D_f(P\kappa\|Q\kappa)$. Any output y with $(Q\kappa)(y) = 0$ must have $\kappa(y \mid x) = 0$ wherever $q(x) > 0$, hence $(P\kappa)(y) = 0$ as well, contributing nothing to either side. Summing over y completes the proof. ■

No matter which divergence function we pick, processing data can never make two distributions more distinguishable. In deep learning, this inequality is often read in reverse: a divergence measured on learned features, quantized outputs, or any other summary of the data is always a lower bound on the divergence between the raw inputs. Merging outcomes in Pinsker's proof is simply the deterministic channel $x \mapsto \mathbf{1}(x \in A^\star)$. On our running pair, it lowered KL from 0.396 to 0.382, and it lowers χ^2 from 1.013 to $0.16/0.2 + 0.16/0.8 = 1$. The KL special case is also the engine behind the data-processing inequality for mutual information in Section 37.6.2.2.

An invertible transformation acts as a channel in both directions, forcing equality in Equation 37.32: every f-divergence is invariant to relabeling outcomes or rescaling the axes. By contrast, the last two rows of Table 37.2 do change when you rescale the axis. What those distances gain in geometric sensitivity to d, they give up in scale invariance.

37.3.4 The Fenchel Conjugate and the Critic Bound

Suppose we have a generative network that produces samples from a distribution Q, alongside a dataset of samples from the true distribution P. Neither collection of samples gives us access to the density ratio $p(x)/q(x)$ required by Equation 37.26. Can we still estimate D_f from samples alone? As it turns out, we can lower-bound any f-divergence using only expectations under P and Q, and that lower bound turns any divergence into an adversarial training objective.

37.3.4.1 Derivation of the Bound

The key tool is the convex conjugate from Section 35.7.4. For a convex function f, its conjugate is defined by

$$f^*(t) = \sup_u \, (ut - f(u)),$$

so that $-f^*(t)$ is the vertical intercept of the tangent line to f with slope t (Figure 37.9). For $f(u) = u \log u$, the supremum occurs where $\log u + 1 = t$, giving $f^*(t) = e^{t-1}$. Because a convex function always lies above every one of its tangent lines, we obtain the Fenchel–Young inequality:

$$f(u) \geq ut - f^*(t) \quad \text{for all } t, \tag{37.33}$$

with equality whenever t matches the slope of f at u, namely $t = f'(u)$.

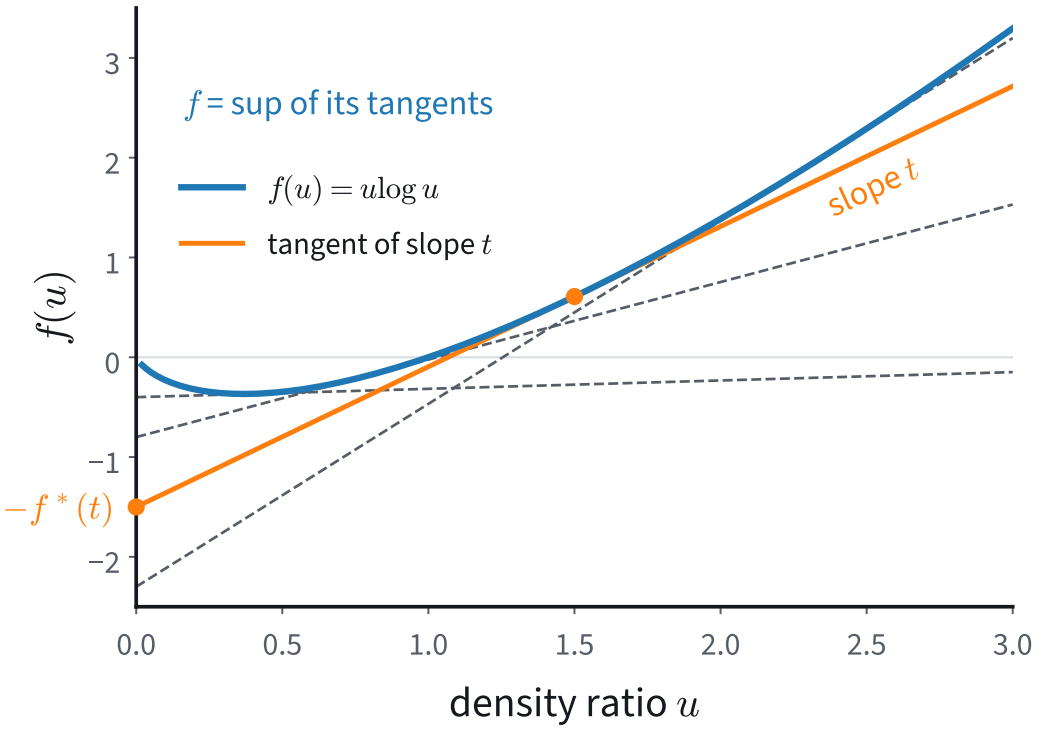


Figure 37.9: A convex divergence function, here $f(u) = u \log u$, is the upper envelope of its tangent lines. The tangent of slope t has vertical intercept $-f^*(t)$, here $-e^{t-1}$. Keeping a single tangent gives the Fenchel–Young lower bound, and taking the supremum over slopes recovers $f(u)$. Applying this inequality pointwise to the density ratio yields the critic bound.

Now apply Equation 37.33 *pointwise inside the divergence*, letting the slope vary with x. Specifically, pick any function $T(x)$, called the **critic**, and substitute $u = p(x)/q(x)$ and $t = T(x)$.

Proposition (critic bound). *Let f be convex and differentiable with $f(1) = 0$, let $P \ll Q$, and let T be any critic with $\mathbb{E}_P|T| < \infty$ and $\mathbb{E}_Q|f^*(T)| < \infty$. Then*

$$D_f(P\|Q) \;\geq\; \mathbb{E}_{x\sim P}[T(x)] - \mathbb{E}_{x\sim Q}[f^*(T(x))]. \tag{37.34}$$

Moreover, the supremum of the right-hand side over all critics equals $D_f(P\|Q)$, and it is attained by $T^\star(x) = f'(p(x)/q(x))$ wherever $T^\star$ is finite (Nguyen et al. 2010; Nowozin et al. 2016).

Proof. Multiply Equation 37.33 at $u = p(x)/q(x)$ and $t = T(x)$ by $q(x) \geq 0$:

$$q(x)\, f\!\left(\frac{p(x)}{q(x)}\right) \;\geq\; p(x)\, T(x) - q(x)\, f^*(T(x)).$$

Summing (or integrating) over x yields Equation 37.34. Because Fenchel–Young holds with equality at $t = f'(u)$, choosing $T^\star(x) = f'(p(x)/q(x))$ turns the inequality into an equality at every x, and therefore in expectation as well. ■

Both terms on the right-hand side of Equation 37.34 are simple expectations, of T over real data and of $f^*(T)$ over generated samples, requiring no explicit densities at all. We can therefore parameterize T as a neural network and train it on minibatches to maximize the bound. For KL divergence, the bound reads $D_{\mathrm{KL}}(P\|Q) \geq \mathbb{E}_P[T] - \mathbb{E}_Q[e^{T-1}]$ for every critic T. The **f-GAN** of Nowozin et al. (2016) (after which Equation 37.34 is also called the f-GAN bound) turns this variational form into a two-player game: the critic maximizes the bound so that it tracks the divergence, while the generator producing Q minimizes it. Section 16.2 implements and trains such pairs.

37.3.4.2 The Original GAN

The original GAN of Goodfellow et al. (2014) falls right out of this framework when we choose $f(u) = u \log u - (u+1)\log(u+1)$. Its convex conjugate is $f^*(t) = -\log(1 - e^t)$ for $t < 0$. If we write the critic as the logarithm of a discriminator $D(x) \in (0,1)$, setting $T = \log D$, the critic bound becomes the familiar binary cross-entropy game:

$$\mathbb{E}_P[T] - \mathbb{E}_Q[f^*(T)] = \mathbb{E}_{x\sim P}[\log D(x)] + \mathbb{E}_{x\sim Q}[\log(1 - D(x))].$$

The optimal critic $T^\star = f'(p/q) = \log\left(p/(p+q)\right)$ corresponds to the Bayes-optimal discriminator $D^\star = p/(p+q)$. Comparing with the Jensen–Shannon divergence function f_{JS} in Table 37.3, we find that $f(u) = 2f_{\mathrm{JS}}(u) - (u-1)\log 2 - \log 4$: twice the Jensen–Shannon function, plus a linear term in $u-1$ (which changes no divergence) and the constant $f(1) = -\log 4$ (which only shifts the value). At $D^\star$, the minimax game therefore achieves $2\,\mathrm{JS}(P,Q) - \log 4$, which on our running pair equals $2(0.0911) - 1.386 = -1.204$.

37.3.4.3 Limits of the Bound

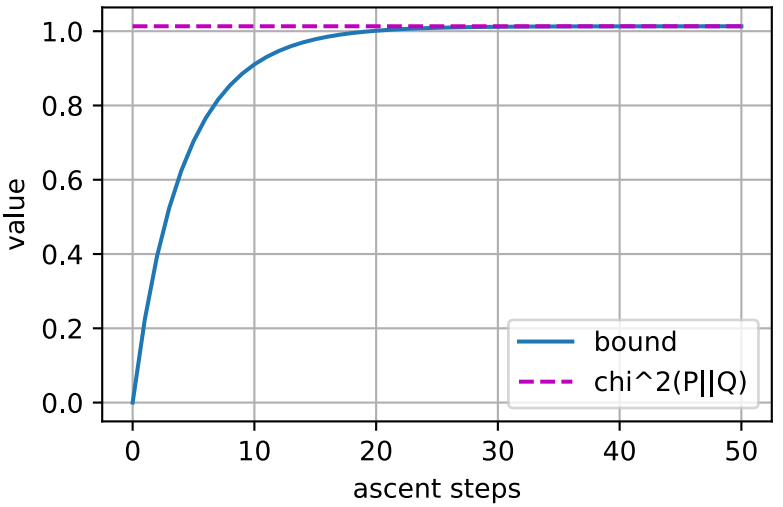
In practice, the critic bound runs into two limitations, and we can measure the first one directly. Because the bound is tight only at $T^\star$, an undertrained critic reports less than the true divergence. Let us watch how a critic closes that gap as it trains.

On our discrete running pair, expectations are simple sums and a critic is just a vector of three numbers, one per outcome. Take the χ^2 divergence, with $f(u) = (u-1)^2$, conjugate $f^*(t) = t + t^2/4$, and optimal critic $T^\star = 2(p/q - 1)$. (Here the conjugate takes the supremum over all real u; Exercise 6a derives it and checks that this loses nothing at $T^\star$.) Starting from $T = 0$, where the bound is 0, we run gradient ascent with step size 1 on the bound, whose derivative with respect to $T(x)$ is $p(x) - q(x)\,(1 + T(x)/2)$. By the proposition, the ascent must climb from below toward $\chi^2(P\|Q) = 1.0133$. As a sanity check, the final line

also evaluates an overconfident critic $2T^\star$ that points in the right direction but overshoots by a factor of two.

```
def fgan_bound(T, p, q, f_star):
    """E_P[T] - E_Q[f*(T)]: a lower bound on D_f(P||Q) for any critic T."""
    return np.sum(p * T) - np.sum(q * f_star(T))

chi2_star = lambda t: t + t ** 2 / 4    # conjugate of f(u) = (u - 1)^2
exact = f_divergence(generators['chi^2'], p, q)
T, bounds = np.zeros(3), []             # untrained critic: one number per outcome
for step in range(51):
    bounds.append(fgan_bound(T, p, q, chi2_star))
    T = T + (p - q * (1 + T / 2))       # ascent step 1 along p - q f*'(T)
shown = (0, 1, 2, 5, 10, 20, 50)
print(f'exact chi^2(P||Q) = {exact:.4f}')
print('steps ' + ''.join(f'{s:8d}' for s in shown))
print('bound ' + ''.join(f'{bounds[s]:8.4f}' for s in shown))
T_star = 2 * (p / q - 1)                # the optimal critic f'(p/q)
print(f'overconfident critic 2 T*: bound = '
      f'{fgan_bound(2 * T_star, p, q, chi2_star):.4f}')
d2l.plot(np.arange(51), [bounds, [exact] * 51], 'ascent steps', 'value',
         legend=['bound', 'chi^2(P||Q)'], figsize=(4.5, 2.8))
```

```
exact chi^2(P||Q) = 1.0133
steps        0       1       2       5      10      20      50
bound   0.0000  0.2240  0.3935  0.7036  0.9107  1.0013  1.0133
overconfident critic 2 T*: bound = 0.0000
```



After five steps, the critic certifies 0.70 out of 1.01 (about 70% of the true divergence); after ten steps it reaches 90%, and after twenty steps 99%. At no point does the bound overshoot the true value. Even the overconfident critic $2T^\star$ reports zero, no better than an untrained critic. With exact expectations, any adversarial divergence estimate is an underestimate, and the number itself never reveals how much is missing. When expectations are replaced by finite-sample averages, it remains a valid lower bound (in expectation) only on fresh

samples that were not used to train the critic.

The gradient update also reveals where the missing divergence hides. Because $T(x)$ moves at a rate proportional to $q(x)$, the critic learns slowest where the model's samples are rarest. On our running pair, that is the first outcome, where the density ratio is largest and most of the divergence lives (Exercise 6b).

The second limitation brings us back to disjoint supports, now viewed from the critic's perspective. Extending $T^\star = f'(p/q)$ formally beyond the hypothesis $P \ll Q$, the ratio equals ∞ on the data support and 0 on the model support, so $T^\star$ takes the constant value $f'(\infty)$ on one and $f'(0)$ on the other. Because the generator updates its parameters using the gradient of the critic evaluated at its own samples, a critic that is locally constant there provides zero gradient. The generator receives no signal telling it which way to move.

Both limitations stem from the same root: the density ratio p/q. Finite samples never hand us p/q directly, and whenever the supports separate, the ratio collapses to 0 or ∞ and erases all trace of how far apart the distributions sit. In Section 37.4, we turn to comparisons that do not rely on density ratios at all.

37.3.5 Summary

KL divergence is one member of a broad family: an f-divergence averages a convex function of the density ratio p/q under Q. Jensen's inequality guarantees non-negativity for every member, all smooth members agree up to their curvature $f''(1)$ when P and Q are close, and no post-processing of the data can increase any f-divergence. Choosing the direction of KL is a modeling decision: forward KL (maximum likelihood) stretches to cover every mode, whereas reverse KL (variational inference) can lock onto a single one. Operationally, the best single-sample test distinguishing P from Q succeeds with probability $\frac{1}{2}(1 + \mathrm{TV})$, and Pinsker's inequality bounds TV by $\sqrt{D_{\mathrm{KL}}/2}$. Using the convex conjugate, any critic lower-bounds D_f via expectations over P and Q alone, which is how adversarial training estimates divergences from samples. Yet every member shares the same blind spot of the density ratio: on disjoint supports, D_f freezes at the constant $f(0) + f'(\infty)$, regardless of how near or far the supports lie.

37.3.6 Exercises

1. **Divergence functions and argument order.** Each row of Table 37.3 is one choice of f in the template Equation 37.26.
   1. For a divergence function f, define $f^\diamond(u) = u\, f(1/u)$. Show that $f^\diamond$ is convex with $f^\diamond(1) = 0$ and that $D_f(Q\|P) = D_{f^\diamond}(P\|Q)$. Compute $f^\diamond$ for the KL function $u \log u$ and for the χ^2 function $(u-1)^2$, and write the divergence that the second one defines as a sum over x.
   2. Adding $c\,(u-1)$ to f does not change the divergence. Show that D_f is symmetric whenever $f^\diamond(u) = f(u) + c\,(u-1)$ for some constant c. Check this condition for the rows of Table 37.3 that are marked symmetric, and show that $D_{f_\alpha}(Q\|P) = D_{f_{1-\alpha}}(P\|Q)$ for the function Equation 37.28. Use this identity to confirm the entry of the α-divergence row in the "Symmetric?" column.
   3. Expand Equation 37.27 in terms of the ratio $u = p/q$ and recover the Jensen–Shannon function of Table 37.3.
   4. *Rényi divergence.* For $\alpha > 0$, $\alpha \neq 1$, define $D_\alpha(P\|Q) = \frac{1}{\alpha-1} \log \sum_x p(x)^\alpha\, q(x)^{1-\alpha}$ (Rényi 1961). Show that $\sum_x p^\alpha q^{1-\alpha} = 1 + \alpha(\alpha -$

1) $D_{f_\alpha}(P\|Q)$ and that D_α is an increasing function of D_{f_α}. Deduce that $D_\alpha \geq 0$ and that D_α satisfies the data-processing inequality Equation 37.32. Find $\lim_{\alpha\to 0} D_\alpha(P\|Q)$, and explain why it is 0 whenever p and q are positive on the same outcomes. Why does the Rényi family therefore *not* reach reverse KL as $\alpha \to 0$, while D_{f_α} does?

2. **Metrics among divergences.** The last column of Table 37.3 records which divergences are metrics.
   1. Prove the triangle inequality for TV from its half-ℓ_1 form.
   2. Show that the Hellinger distance $d_{\mathrm{H}} = \sqrt{d_{\mathrm{H}}^2}$ is the Euclidean distance between the vectors $\left(\sqrt{p(x)}\right)_x$ and $\left(\sqrt{q(x)}\right)_x$, so that d_{H} is a metric. Find three distributions on two outcomes for which both d_{H}^2 and JS violate the triangle inequality.
3. **Cost of dropping a mode.** A model Q misses part of a data distribution P.
   1. Let $P = (1-w)\,Q + wR$ with $0 < w < 1$, where R is a distribution on a set to which Q assigns probability zero. Compute $D_{\mathrm{KL}}(P\|Q)$, $D_{\mathrm{KL}}(Q\|P)$, $\mathrm{TV}(P,Q)$ and $d_{\mathrm{H}}^2(P,Q)$ as functions of w. Which direction of KL penalizes the missing mass? How do the finite divergences behave as $w \to 0$?
   2. Now let $P = w_1\mathcal{N}(m_1, s^2) + w_2\mathcal{N}(m_2, s^2)$ with separation $\Delta = |m_2 - m_1|$, and let the model $Q = \mathcal{N}(m_1, s^2)$ keep only the first component. Show that $D_{\mathrm{KL}}(Q\|P) \leq \log(1/w_1)$ for every Δ, with equality in the limit $\Delta \to \infty$, and relate the limit to (a). Use the bound to explain the two reverse-KL values, 0.356 and 1.202 nats, that the fit of a Gaussian to the mixture prints.
   3. Assume that the components are well separated, so that almost every x has non-negligible density under at most one of them. Show that $D_{\mathrm{KL}}(P\|Q) \approx w_2\Delta^2/(2s^2) + w_1 \log w_1 + w_2 \log w_2$. Evaluate the approximation for the mixture of that experiment, and check it with `kl(log_p(x), log_q(x, [-2.0, np.log(0.6)]), x)`. The experiment prints two values in its forward-KL column to compare with: 5.981 nats at the reverse-KL fit, whose parameters $(-1.998, 0.603)$ are not quite $(-2, 0.6)$, and 0.558 nats at the forward fit.
4. **Local quadratic regime.** The proposition Equation 37.29 bounds the gap between D_f and $\frac{1}{2}f''(1)\,\chi^2$.
   1. Exercise 7b of Section 37.1 finds the order at which the two directions of the Gaussian KL divergence first differ. Which property of the KL and reverse-KL functions explains that order? Carry the Taylor expansion of the proof one term further, and show that the ratio which the local-quadratic experiment prints is $1 + c\,t\,f'''(1)/f''(1) + O(t^2)$ with $c = \mathbb{E}_Q[(u-1)^3]/(3\chi^2(P\|Q))$, about 0.49 for the running pair. Compute $f'''(1)/f''(1)$ for the four functions. Why is the shortfall of reverse KL at $t = 0.01$ twice that of KL, and why do squared Hellinger and JS print the same value there?
   2. The total-variation function $\frac{1}{2}|u-1|$ has no second derivative at 1. Let $P_t = Q + t\,(P-Q)$ for $t \in (0,1]$. How do $\mathrm{TV}(P_t, Q)$ and $\chi^2(P_t\|Q)$ scale with t? Explain from this why Pinsker's inequality Equation 37.31 bounds TV by the square root of KL and not by a multiple of KL.
   3. Let P and Q be coins with heads probabilities b and $b+\delta$. Use the proposition to show that $D_{\mathrm{KL}}(P\|Q) = \delta^2/(2b(1-b)) + O(\delta^3)$. Exercise 5a of Section 37.2 obtains the same expansion directly from the Bernoulli KL divergence. The factor $1/(b(1-b))$ is the Fisher information of the coin (Section 36.6). Find the limit of $\mathrm{TV}/\sqrt{D_{\mathrm{KL}}/2}$ as $\delta \to 0$, and the values of b for which Pinsker's inequality is tight in this limit. The coin experiment that follows Pinsker's inequality takes $b = \frac{1}{2}$

and prints a ratio of 0.999999 at $\delta = 10^{-3}$. Why does the ratio approach 1 at the rate δ^2 and not δ?

5. **Hellinger bounds and testing.** ● Pinsker's inequality is vacuous once $D_{\mathrm{KL}} > 2$. Let $\mathrm{BC}(P,Q) = \sum_x \sqrt{p(x)q(x)} = 1 - \frac{1}{2}d_{\mathrm{H}}^2(P,Q)$ be the Bhattacharyya coefficient.
   1. Prove that $\frac{1}{2}d_{\mathrm{H}}^2(P,Q) \le \mathrm{TV}(P,Q)$, using $|p-q| = |\sqrt{p}-\sqrt{q}|\,(\sqrt{p}+\sqrt{q})$.
   2. Prove that $\mathrm{TV}(P,Q) \le d_{\mathrm{H}}(P,Q)\sqrt{1 - \frac{1}{4}d_{\mathrm{H}}^2(P,Q)}$ with the Cauchy–Schwarz inequality of Section 33.1, and rewrite the bound as $\mathrm{TV}^2 \le 1 - \mathrm{BC}^2$.
   3. Use Jensen's inequality to show that $\mathrm{BC}(P,Q) \ge e^{-D_{\mathrm{KL}}(P\|Q)/2}$, and conclude the Bretagnolle–Huber inequality $\mathrm{TV}(P,Q) \le \sqrt{1 - e^{-D_{\mathrm{KL}}(P\|Q)}}$ (Bretagnolle and Huber 1978; Canonne 2022). For which values of D_{KL} is it tighter than Equation 37.31?
   4. A test sees n independent samples, all from P or all from Q, each case with probability $\frac{1}{2}$. Use (c) and the additivity of KL over independent samples from Section 37.1.2.4 to show that every test errs with probability at least $\frac{1}{4}e^{-nD_{\mathrm{KL}}(P\|Q)}$. How many samples does an error probability of at most η require? Why can Pinsker's inequality not give a bound of this kind?
6. **Conjugates and critics.** The critic bound Equation 37.34 needs the conjugate f^* of the divergence function.
   1. Derive the conjugate $f^*(t) = t + t^2/4$ of the χ^2 function $f(u) = (u-1)^2$, taking the supremum over $u \in \mathbb{R}$, and its optimal critic $T^\star = 2(p/q - 1)$. Redo the computation over the function's domain $u \in (0,\infty)$, and show that the supremum is $t + t^2/4$ for $t \ge -2$ and -1 for $t \le -2$.
   2. Use the $\mathbb{R}$-conjugate. Show that for every critic T the gap between $\chi^2(P\|Q)$ and the bound equals $\frac{1}{4}\mathbb{E}_Q\big[(T-T^\star)^2\big]$. Deduce that a scaled critic $T = cT^\star$ certifies $\chi^2(P\|Q)\,(2c - c^2)$, which explains the value that the experiment prints for $2T^\star$. Show that one ascent step of the experiment multiplies $T(x) - T^\star(x)$ by $1 - q(x)/2$. Which outcome of the running pair accounts for most of the gap after ten steps, and why? For which step sizes does the ascent converge?
   3. Compute the conjugate of the total-variation function $f(u) = \frac{1}{2}|u-1|$ over $u \in (0,\infty)$. Show that critics with values in $[-\frac{1}{2}, \frac{1}{2}]$ suffice to maximize the bound, and conclude that $\mathrm{TV}(P,Q) = \sup_g \left(\mathbb{E}_P[g] - \mathbb{E}_Q[g]\right)$ over all functions g with $|g| \le \frac{1}{2}$, a supremum of the kind that Section 37.4.1 studies. Which g attains it?

37.4 Test Functions and Transport

A model whose samples miss the data learns nothing from an f-divergence: on disjoint supports, every member of that family reduces to a constant (Section 37.3.1.2), and a constant has zero gradient. That diagnosis came from an extreme example, two point masses. Real distributions usually have spread: two Gaussians overlap everywhere because their tails never reach zero. Does a little overlap bring the gradient back? The f-divergences also left us with a practical burden: because they depend on the density ratio p/q, which raw samples do not provide, we had to train a critic network just to estimate the divergence (Section 37.3.4). Can we compare two distributions from samples alone, with no critic at all?

To answer the first question with numbers, let the data distribution be $P = \mathcal{N}(0,1)$ and the model be $Q = \mathcal{N}(d,1)$, the same bell curve shifted by d standard deviations, where d

is the model's single parameter. Below, after importing the libraries we need, we evaluate the five comparisons from Table 37.2 on this pair along with their slopes in d, which are the gradients the model would receive during training. The formulas used in the code are derived in the previous section and this one; for now, focus only on the printed numbers, which we will refer to as the opening table.

```
from scipy import optimize
from scipy.stats import norm
```

```
def js(d, x=np.linspace(-12, 20, 6401)):
    """JS divergence of N(0, 1) and N(d, 1) by quadrature on the grid x."""
    log_p, log_q = norm.logpdf(x), norm.logpdf(x, loc=d)
    log_m = np.logaddexp(log_p, log_q) - math.log(2)    # log of the mixture
    terms = np.exp(log_p) * (log_p - log_m) + np.exp(log_q) * (log_q - log_m)
    return terms.sum() * (x[1] - x[0]) / 2

comparisons = {                          # N(0, 1) against N(d, 1), RBF ell = 1
    'KL': lambda d: d ** 2 / 2,
    'JS': js,
    'TV': lambda d: 2 * norm.cdf(d / 2) - 1,
    'MMD^2': lambda d: 2 / math.sqrt(3) * (1 - math.exp(-d ** 2 / 6)),
    'W1': lambda d: d}
h = 1e-3                                 # step of the central difference
print('   d' + ''.join(f'{name:>8s} {"slope":>7s}' for name in comparisons))
for d in (0.5, 1.0, 2.0, 4.0, 8.0):
    print(f'{d:4.1f}' + ''.join(
        f'{f(d):8.3f} {(f(d + h) - f(d - h)) / (2 * h):7.1e}'
        for f in comparisons.values()))
```

```
  d      KL   slope      JS   slope      TV   slope   MMD^2   slope      W1   slope
0.5   0.125 5.0e-01   0.030 1.2e-01   0.197 3.9e-01   0.047 1.8e-01   0.500 1.0e+00
1.0   0.500 1.0e+00   0.111 2.0e-01   0.383 3.5e-01   0.177 3.3e-01   1.000 1.0e+00
2.0   2.000 2.0e+00   0.337 2.2e-01   0.683 2.4e-01   0.562 4.0e-01   2.000 1.0e+00
4.0   8.000 4.0e+00   0.633 6.9e-02   0.954 5.4e-02   1.074 1.1e-01   4.000 1.0e+00
8.0  32.000 8.0e+00   0.693 2.0e-04   1.000 1.3e-04   1.155 7.2e-05   8.000 1.0e+00
```

When the two Gaussians are half a standard deviation apart, all five slopes lie between 0.1 and 1, so any of these objectives would train the model effectively. By $d = 8$, however, the Jensen–Shannon divergence has saturated at 0.693 and total variation at 1.000 (the same constants $\log 2$ and 1 we saw for disjoint point masses), and both slopes have fallen to roughly 10^{-4}. Overlapping tails do not rescue them. The KL divergence still has a healthy slope equal to d itself, but only because a Gaussian model assigns positive density to every data point; an implicit generator whose samples lie on a low-dimensional surface has no such density. That leaves the two distances that remained informative for point masses. MMD^2 grows initially but stalls by $d = 8$, reaching 1.155 with a slope below 10^{-4}. Only W_1 stays linear, equaling d with a slope of 1 at every separation.

Neither MMD nor W_1 relies on the density ratio p/q. Instead, MMD compares how P and Q *average* a class of functions: averages can be estimated directly from samples, and the choice of function class governs how far apart two distributions can be and still produce a useful signal. W_1 begins from a geometric question: how far must probability mass travel to reshape one distribution into the other? Surprisingly, it turns out to be a comparison of

averages as well. Two results organize the rest of this section. Over a fixed set of features, the comparison of averages admits a closed form with no auxiliary network to train. Meanwhile, the minimum cost of moving mass equals the largest gap in averages over functions of slope at most one, which is the duality behind the Wasserstein GAN (Section 16.2). Throughout, P denotes the data and Q the model, and we work with concrete examples on the real line.

37.4.1 Integral Probability Metrics and MMD

37.4.1.1 Classes of Test Functions

Total variation is a proper metric, yet it still flattened out in the opening table. Its variational form explains both why that happened and how to fix it: total variation is the largest disagreement between P and Q over the probability of any event A (Section 37.3.3), namely $\mathrm{TV}(P, Q) = \sup_A |P(A) - Q(A)|$. Since the probability of an event is simply the expectation of its indicator function, total variation is a supremum, taken over a class of functions, of the difference between two averages. Nothing forces us to restrict ourselves to indicator functions. Given any class $\mathcal{G}$ of *test functions* g, the **integral probability metric** (IPM) (Müller 1997) is defined as

$$\mathrm{IPM}_{\mathcal{G}}(P, Q) = \sup_{g \in \mathcal{G}} \left(\mathbb{E}_{x \sim P}[g(x)] - \mathbb{E}_{x \sim Q}[g(x)]\right). \tag{37.35}$$

Think of $\mathcal{G}$ as a panel of auditors, each scoring points in the space; the IPM reports the largest gap in average score that any auditor on the panel can uncover. Every class we consider is symmetric, meaning $-g$ belongs to $\mathcal{G}$ whenever g does, so the gap is always nonnegative and symmetric in P and Q. Whether the IPM vanishes only when $P = Q$, as the word "metric" promises, depends on whether $\mathcal{G}$ is rich enough to distinguish any two distinct distributions. The critic of Section 37.3.4 played a similar auditing role, but with a crucial difference: in the variational bound $D_f(P\|Q) \geq \mathbb{E}_P[T] - \mathbb{E}_Q[f^*(T)]$, the convex conjugate f^* automatically keeps an unconstrained critic T in check. In an IPM, g is averaged directly under both P and Q, so nothing constrains g except membership in $\mathcal{G}$. Everything therefore hinges on choosing the right class.

Three function classes take center stage here:

- Bounded functions with $|g| \leq \frac{1}{2}$ recover total variation. An indicator function minus $\frac{1}{2}$ satisfies this bound, the constant cancels in the difference of averages, and $g = \frac{1}{2}\,\mathrm{sign}(p - q)$ attains the supremum.
- Functions that are linear in a fixed set of features yield the **maximum mean discrepancy** (MMD) (Gretton et al. 2012), which we construct next.
- Functions with slope at most one yield the Wasserstein distance, covered in detail in Section 37.4.2.

Why not simply choose the largest possible class $\mathcal{G}$ every time? A useful class must satisfy two competing requirements, and making $\mathcal{G}$ too large breaks the first. First, sample averages must track true expectations. While samples let us estimate expectations directly without a density ratio, an overly rich panel of auditors will always find spurious gaps between finite samples. For example, two independent samples of $1{,}000$ points drawn from the *same* continuous distribution share no points at all. The indicator function of the first sample averages to 1 on the first sample and 0 on the second, so the empirical total variation

is 1 regardless of whether P and Q are identical. To avoid this, $\mathcal{G}$ must be small or smooth enough to exclude such spiky functions.

Second, the gap should keep growing as P and Q move farther apart, which requires $\mathcal{G}$ to contain functions that grow with distance. Total variation has none: because its test functions are bounded by $\frac{1}{2}$, its gap can never exceed 1. The largest disagreement occurs on the event $\{p > q\}$, which for the two Gaussians in the opening table is the half-line $x < d/2$. Computing the probability of that half-line under both Gaussians gives $\mathrm{TV} = \Phi(d/2) - \Phi(-d/2) = 2\Phi(d/2) - 1$, where Φ is the standard normal cumulative distribution function (CDF). That value reaches 0.954 at $d = 4$ and lies within 10^{-4} of 1 at $d = 8$. Once a step function separates the two bell curves almost completely, pushing them farther apart changes the score by virtually nothing. Total variation fails both requirements, and we will judge our next two classes against both.

37.4.1.2 Mean Features and Kernels

Let us build a better class from the ground up, starting from the smallest possible panel of auditors and testing each step on a single pair: $P = \mathcal{N}(0, 1)$ against $Q = \mathcal{N}(0, 4)$, which share the same mean but differ in variance.

Start with the simplest choice, linear test functions $g(x) = w\,x$ with $|w| \le 1$. Here $\mathbb{E}_P g - \mathbb{E}_Q g = w\,(\mathbb{E}_P X - \mathbb{E}_Q X)$, so taking the supremum over $|w| \le 1$ gives $|\mathbb{E}_P X - \mathbb{E}_Q X|$, the distance between the two means. That panel is blind to variance, and on our test pair it reports 0.

To see spread as well as location, add a second feature. Define the feature map $\varphi(x) = (x, x^2)$ and take linear combinations $g(x) = \mathbf{w}^\top \varphi(x)$ with $\|\mathbf{w}\| \le 1$. Writing $\mathbf{m}_P = \mathbb{E}_P[\varphi(x)]$ for the *mean feature* of P (and similarly $\mathbf{m}_Q$ for Q), the gap in averages is $\mathbb{E}_P g - \mathbb{E}_Q g = \mathbf{w}^\top(\mathbf{m}_P - \mathbf{m}_Q)$. By the Cauchy–Schwarz inequality, the maximum of a linear function over the unit ball is simply the Euclidean norm of its coefficient vector:

$$\mathrm{MMD}(P, Q) = \sup_{\|\mathbf{w}\| \le 1} \mathbf{w}^\top(\mathbf{m}_P - \mathbf{m}_Q) = \|\mathbf{m}_P - \mathbf{m}_Q\|. \tag{37.36}$$

Thus MMD is the distance between the mean feature vectors of P and Q, and the supremum has disappeared altogether. On our test pair, the mean features are $\mathbf{m}_P = (0, 1)$ and $\mathbf{m}_Q = (0, 4)$, giving $\mathrm{MMD}^2 = 0^2 + 3^2 = 9$: the comparison now detects differences in both means and second moments.

Next, expand the squared norm. Because the product of two expectations equals the expectation of a product over independent draws, we can write $\|\mathbf{m}_P\|^2 = \mathbb{E}\langle \varphi(x), \varphi(x')\rangle$ for independent $x, x' \sim P$. Doing the same for independent $y, y' \sim Q$ and the cross term gives $\|\mathbf{m}_P - \mathbf{m}_Q\|^2 = \mathbb{E}\langle \varphi(x), \varphi(x')\rangle + \mathbb{E}\langle \varphi(y), \varphi(y')\rangle - 2\,\mathbb{E}\langle \varphi(x), \varphi(y)\rangle$. Notice that the features $\varphi(x)$ never appear alone; only their inner products do. Defining the **kernel** $k(x, x') = \langle \varphi(x), \varphi(x')\rangle$, we obtain

$$\mathrm{MMD}^2(P, Q) = \mathbb{E}_{x,x' \sim P}[k(x, x')] + \mathbb{E}_{y,y' \sim Q}[k(y, y')] - 2\,\mathbb{E}_{x \sim P, y \sim Q}[k(x, y)]. \tag{37.37}$$

This expression has an immediate geometric reading: the first two terms measure average similarity *within* P and *within* Q, while the third measures similarity *across* them. When

draws from P and Q intermingle, within-sample and cross-sample similarities match and the three terms cancel; when they separate into distinct clusters, the within terms dominate. Our first step corresponds to the *linear kernel* $k(x, x') = xx'$, and our second to the *quadratic kernel* $k(x, x') = xx' + x^2x'^2$.

Even the quadratic kernel still has a blind spot: any two distributions with the same mean and variance have distance zero. Adding x^3, then x^4, and so on would never end. The decisive step is therefore to choose the kernel directly as a similarity measure, without ever writing down a finite list of features. The *RBF kernel* $k(x, x') = e^{-(x-x')^2/(2\ell^2)}$ treats two points as similar when they lie within roughly a *length scale* ℓ of each other. Although we chose it directly, it is still an inner product of features: factoring it as $e^{-x^2/(2\ell^2)}\, e^{-x'^2/(2\ell^2)}\, e^{xx'/\ell^2}$ and expanding the last factor as a power series reveals an inner product over infinitely many features $\varphi_j(x) = x^j e^{-x^2/(2\ell^2)}/(\ell^j\sqrt{j!})$ for $j = 0, 1, 2, \ldots$, which are damped monomials of every order with rapidly decaying weights. Our panel now contains infinitely many auditors, yet evaluating its verdict still requires only the three averages in Equation 37.37. In $\mathbb{R}^n$, we simply replace $(x - x')^2$ with $\|x - x'\|^2$ and nothing else changes.

With this kernel, no blind spot remains: $\mathrm{MMD}(P, Q) = 0$ if and only if $P = Q$ (Gretton et al. 2012), making RBF-MMD a true metric. Kernels with this property are called *characteristic*; the linear and quadratic kernels are not.

For Gaussians, we can evaluate the RBF expectations in closed form. If $Z \sim \mathcal{N}(\mu, s^2)$, a standard Gaussian integral gives

$$\mathbb{E}\big[e^{-Z^2/(2\ell^2)}\big] = \Big(1 + \frac{s^2}{\ell^2}\Big)^{-1/2} \exp\Big(-\frac{\mu^2}{2(\ell^2 + s^2)}\Big). \tag{37.38}$$

Because the difference of two independent Gaussians is again Gaussian with the sum of their variances, we can read off all three terms of Equation 37.37 immediately. On our test pair $\mathcal{N}(0, 1)$ versus $\mathcal{N}(0, 4)$ at $\ell = 1$, the differences $x - x'$, $y - y'$, and $x - y$ all have mean zero and variances 2, 8, and 5, yielding $\mathrm{MMD}^2 = 3^{-1/2} + 9^{-1/2} - 2\cdot 6^{-1/2} \approx 0.094$. For the two shifted Gaussians of the opening table, the same integral gives $\frac{2}{\sqrt{3}}\big(1 - e^{-d^2/6}\big)$ (Exercise 1a). For the point masses of Table 37.2, there is no averaging to do at all, and Equation 37.37 immediately gives $1 + 1 - 2k(0, d) = 2 - 2e^{-d^2/2}$, matching the fourth row of that table. In both cases, only the cross term depends on d, and it decays rapidly once d exceeds a few length scales. The gap therefore flattens out, failing our second requirement: an RBF auditor is smooth, but nearsighted.

37.4.1.3 Estimation from Samples

Where a kernel shines is our first requirement: reliable estimation from samples. Given finite samples from P and Q, each expectation in Equation 37.37 becomes an average over pairs of points. To obtain an unbiased estimator (Gretton et al. 2012), we omit the diagonal terms pairing a point with itself, which would otherwise bias the within-sample averages upward. Below, we evaluate this estimator on $N = 1{,}000$ draws from $\mathcal{N}(0, 1)$ and $1{,}000$ draws from $\mathcal{N}(0, 4)$ under the linear, quadratic, and RBF kernels, whose population values we just computed as 0, 9, and 0.094. For the RBF kernel, we also test two additional pairs: a second independent sample from $\mathcal{N}(0, 1)$ (population value 0), and a shifted sample from $\mathcal{N}(0.5, 1)$ (population value 0.047 in the opening table). The goal is to see how tightly the sample estimates track these population values.

```
def mmd2_unbiased(x, y, k):
    """Unbiased MMD^2 between samples x and y for the kernel k(a, b)."""
    kxx = k(x[:, None], x[None, :])
    kyy = k(y[:, None], y[None, :])
    n, m = len(x), len(y)
    return ((kxx.sum() - np.trace(kxx)) / (n * (n - 1))
            + (kyy.sum() - np.trace(kyy)) / (m * (m - 1))
            - 2 * k(x[:, None], y[None, :]).mean())

linear = lambda a, b: a * b                          # features phi(x) = x
quadratic = lambda a, b: a * b + a ** 2 * b ** 2     # features (x, x^2)
rbf = lambda a, b: np.exp(-(a - b) ** 2 / 2)         # RBF kernel, ell = 1

rng = np.random.default_rng()
n = 1000
x_p = rng.standard_normal(n)                 # P = N(0, 1)
y_var = 2 * rng.standard_normal(n)           # N(0, 4)
y_same = rng.standard_normal(n)              # N(0, 1) again
y_shift = rng.standard_normal(n) + 0.5       # N(0.5, 1)
for name, y, k, pop in [
        ('linear,    N(0,1) vs N(0,4)', y_var, linear, 0.0),
        ('quadratic, N(0,1) vs N(0,4)', y_var, quadratic, 9.0),
        ('RBF,       N(0,1) vs N(0,4)', y_var, rbf,
         3 ** -0.5 + 9 ** -0.5 - 2 * 6 ** -0.5),
        ('RBF,       N(0,1) vs N(0,1)', y_same, rbf, 0.0),
        ('RBF,       N(0,1) vs N(0.5,1)', y_shift, rbf,
         2 / np.sqrt(3) * (1 - np.exp(-1 / 24)))]:
    print(f'{name:30s} MMD^2 = {mmd2_unbiased(x_p, y, k):+.4f}  '
          f'(population {pop:.4f})')
```

```
linear,    N(0,1) vs N(0,4)    MMD^2 = -0.0033  (population 0.0000)
quadratic, N(0,1) vs N(0,4)    MMD^2 = +10.2618  (population 9.0000)
RBF,       N(0,1) vs N(0,4)    MMD^2 = +0.1180  (population 0.0942)
RBF,       N(0,1) vs N(0,1)    MMD^2 = +0.0001  (population 0.0000)
RBF,       N(0,1) vs N(0.5,1)  MMD^2 = +0.0351  (population 0.0471)
```

The linear kernel returns a number close to zero because the two means match. Specifically, its unbiased estimate equals the squared difference of the two sample means minus an estimate of that difference's variance, roughly $(1+4)/1{,}000 = 0.005$; as a result, it rarely dips much below -0.005, while about one run in twenty exceeds $+0.015$. The quadratic kernel lands near 9, give or take 3, because fourth powers amplify sampling noise. Under the RBF kernel, the estimate is roughly 0.094 ± 0.03 for the variance-shifted pair and 0.047 ± 0.02 for the mean-shifted pair, so the sampling error can be a noticeable fraction of the signal. Between two independent samples from the *same* $\mathcal{N}(0,1)$ distribution, however, the RBF estimate is remarkably tight, staying within a few thousandths of zero (and occasionally dipping slightly below it) because the first-order sampling errors in the three averages cancel out. Where total variation reported a spurious gap of 1 between two samples from the same distribution, the RBF kernel sees none.

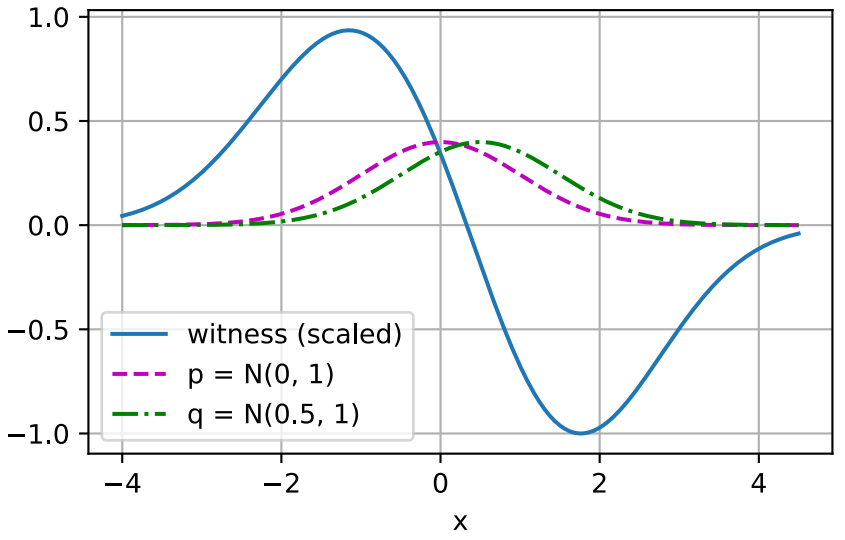
In short, the RBF estimator has relatively large error when measuring a non-zero gap, yet extraordinary precision when $P = Q$. That makes it a powerful *two-sample test*: with $1{,}000$ points per sample, an RBF MMD^2 of even a few hundredths is strong evidence that the two distributions differ, obtained without training a critic or evaluating a single density.

37.4.1.4 The Witness

Because an IPM is defined as a supremum, it is natural to ask which test function actually achieved the maximum gap. For MMD, the supremum in Equation 37.36 is attained by $\mathbf{w}^\star \propto \mathbf{m}_P - \mathbf{m}_Q$. The resulting test function, known as the **witness**, is $g^\star(x) \propto \langle \mathbf{m}_P - \mathbf{m}_Q, \varphi(x) \rangle = \mathbb{E}_P[k(x, X)] - \mathbb{E}_Q[k(x, Y)]$, which again requires only kernel evaluations. For our shifted Gaussians $\mathcal{N}(0, 1)$ and $\mathcal{N}(0.5, 1)$, the two densities are mirror images across $x = 0.25$, so the population witness changes sign right at 0.25. Below, we estimate the witness from our samples on a grid and plot it alongside the two densities. We also print $\mathbb{E}_P g - \mathbb{E}_Q g$ for the unscaled witness $g = \langle \mathbf{m}_P - \mathbf{m}_Q, \varphi \rangle$, which equals $\|\mathbf{m}_P - \mathbf{m}_Q\|^2 = \mathrm{MMD}^2$, estimated here with the self-pairs included.

```
g = lambda s: (rbf(s[:, None], x_p[None, :]).mean(axis=1)    # the unscaled
               - rbf(s[:, None], y_shift[None, :]).mean(axis=1))  # witness
grid = np.linspace(-4.0, 4.5, 400)
witness = g(grid)
print(f'witness changes sign at x = {grid[np.argmax(witness < 0)]:.2f}')
print(f'E_P g - E_Q g = {g(x_p).mean() - g(y_shift).mean():.4f}')
d2l.plot(grid, [witness / np.abs(witness).max(), norm.pdf(grid),
                norm.pdf(grid, loc=0.5)], 'x', '',
         legend=['witness (scaled)', 'p = N(0, 1)', 'q = N(0.5, 1)'],
         figsize=(5, 3))
```

```
witness changes sign at x = 0.35
E_P g - E_Q g = 0.0360
```



As expected, the witness is positive where P has higher density and negative where Q has higher density, crossing zero near $x = 0.25$ (with $1{,}000$ points per sample, the crossing fluctuates by a tenth or two across runs). The printed gap matches the unbiased estimate from the previous cell to within about 0.001, with the small discrepancy coming from the self-pairs.

The tails of the witness plot reveal something even more important than the crossing. To

shrink the gap $\mathbb{E}_P[g]-\mathbb{E}_Q[g]$, a generative model must push each of its samples uphill along g, so the slope of the witness at a model sample is the gradient signal that sample receives. Between the two bell curves, the witness is steep and provides a strong pull toward the data. A few length scales out in the tails, however, the witness flattens to zero, leaving any distant model sample with almost no gradient. We are seeing the MMD^2 column of the opening table from the vantage point of a single sample.

Fixed-kernel MMD therefore plays two distinct roles in practice: as a non-parametric two-sample test, and as a training objective for generative models that avoids adversarial optimization (Li et al. 2015). For instance, the kernel inception distance used to evaluate image generators (Section 16.5) is simply the unbiased MMD estimator applied to deep image features with a cubic kernel. Choosing the length scale ℓ carefully is essential: at $\ell = 1$, the slope at $d = 8$ collapsed to 7×10^{-5} (Exercise 1b explores larger scales). In practice, one often sets ℓ to the median pairwise distance between samples or sums RBF kernels across multiple length scales, the strategy used when training a GAN *generator* with MMD in Section 16.2.

37.4.2 Optimal Transport and the Wasserstein Distance

One column of the opening table never flattened out. W_1 kept a constant slope of 1 because it incorporates the geometry of the underlying space, which f-divergences ignore and a fixed kernel sees only locally. Under W_1, the point mass $\delta_{0.01}$ is close to δ_0 simply because its mass need only travel a distance of 0.01. The idea goes back to 1781, when Gaspard Monge asked how to move a pile of earth into a pit of equal volume with the least total work. That minimum work is a natural measure of how far apart two distributions are.

37.4.2.1 Transport Plans

Consider two discrete distributions supported on the six points $0, 1, \dots, 5$ of the real line, with P concentrated on the left and Q on the right, as shown in the top left panel of Figure 37.10. To reshape P into Q, we must specify how much mass moves from each source x to each destination y. Such a *transport plan* is a joint distribution γ over pairs (x, y) whose marginals are P and Q: the total mass leaving each x equals $P(x)$, and the total mass arriving at each y equals $Q(y)$. In probability theory, a transport plan is also called a *coupling* of P and Q. The **Wasserstein-1 distance** (also known as the earth mover's distance) is the minimum expected distance traveled over all valid plans. Writing $\Pi(P, Q)$ for the set of all couplings of P and Q on $\mathbb{R}^n$ with finite first moments,

$$W_1(P, Q) = \inf_{\gamma \in \Pi(P,Q)} \mathbb{E}_{(x,y)\sim\gamma}\big[\|x - y\|\big]. \tag{37.39}$$

The subscript 1 refers to the first power of the distance; we will meet the squared-distance version W_2 shortly. Between two point masses δ_0 and δ_d, there is only one possible plan, moving all mass from 0 to d, so $W_1(\delta_0, \delta_d) = |d|$, with gradient ± 1 for $d \neq 0$. For the two shifted Gaussians of the opening table, translating every point by d is a valid plan of cost d, and no plan can do better because $\mathbb{E}_\gamma|x - y| \geq |\mathbb{E}_\gamma[x - y]| = d$ for every coupling γ. In both cases, W_1 recovers the separation d itself, providing the clean gradient signal that f-divergences lack.

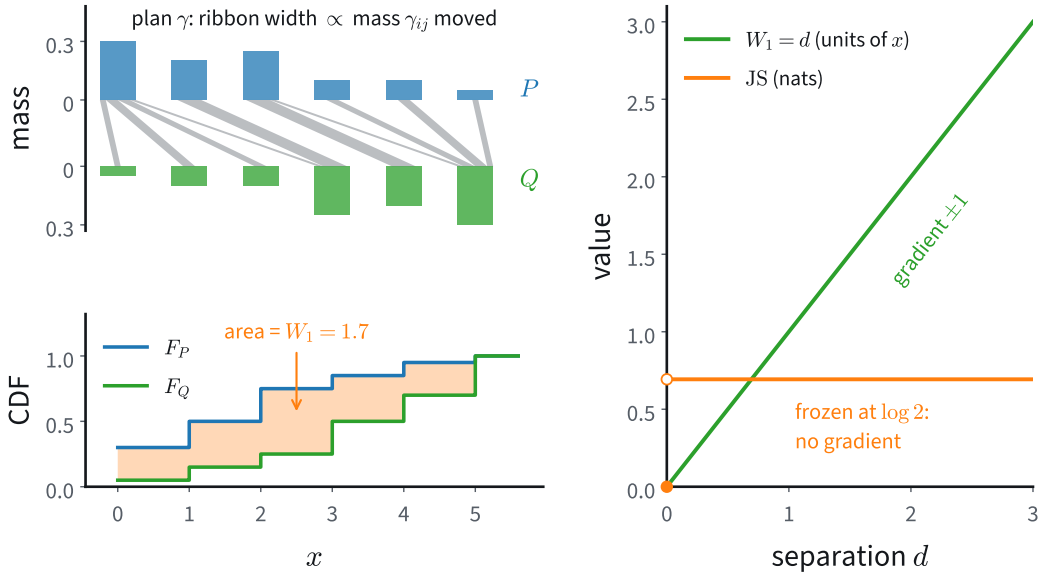


Figure 37.10: Transport on the line. Top left: a transport plan between two distributions on six points; each ribbon carries mass from a point of P (top) to a point of Q (bottom), with width proportional to the mass moved. Bottom left: the CDFs of P and Q; the shaded area between them is 1.7. Right: for two point masses separated by d, the Jensen–Shannon divergence (in nats) stays flat at $\log 2$, whereas W_1 (in units of x) grows linearly with d.

37.4.2.2 Kantorovich–Rubinstein Duality

As a computational recipe, Equation 37.39 looks intimidating: it minimizes over joint distributions γ, which are awkward even to parameterize from samples. On finite point sets, however, it is a familiar optimization problem: a linear program. The unknowns are the entries of the plan, and both the transport cost and the marginal constraints are linear in those entries. Every linear program has a *dual* problem that maximizes over multipliers for its constraints (Section 35.7.1). For optimal transport, the dual replaces the search over joint distributions with a maximization over a single function, and that reformulation is what makes W_1 practical in deep learning.

Proposition (Kantorovich–Rubinstein duality). *For distributions P and Q on $\mathbb{R}^n$ with finite first moments,*

$$W_1(P, Q) = \sup_{\|g\|_{\mathrm{Lip}} \le 1} \left(\mathbb{E}_{x \sim P}[g(x)] - \mathbb{E}_{y \sim Q}[g(y)] \right), \tag{37.40}$$

where the supremum runs over all 1-Lipschitz functions g, meaning $|g(x) - g(y)| \le \|x - y\|$ for all x, y.

An infimum over couplings has turned into a supremum over a single test function: W_1 is precisely the IPM Equation 37.35 over functions with slope at most 1, the third class we promised earlier. A 1-Lipschitz auditor cannot make sudden jumps, so it cannot memorize individual sample points the way an indicator function does. Yet unlike bounded functions or RBF kernels, it can grow without limit, satisfying our second requirement. On the two

shifted Gaussians of the opening table, for example, $g(x) = -x$ is 1-Lipschitz and has expectations 0 under P and $-d$ under Q, achieving a gap of $d = W_1(P, Q)$. Better still, its gradient points toward the data with magnitude 1 everywhere, no matter how far a model sample lies from the data.

Because a neural network can approximate such a test function, the **WGAN** (Arjovsky et al. 2017) trains the dual objective Equation 37.40 directly, using a neural critic constrained to be approximately 1-Lipschitz in the role of g. Section 16.2 compares this objective with other adversarial losses and trains such a critic, whose slope a gradient penalty pulls toward one. Section 16.4 shows why later recipes center their penalties at zero instead: a penalty that prefers unit slope keeps pushing a generator that already matches the data, while one centered at zero only keeps the critic's gradient small at the samples and enforces no Lipschitz constraint. Keep in mind that when a critic is only approximately Lipschitz and restricted to a neural architecture, the score it reports is no longer exact W_1.

37.4.2.3 Transport on the Line

We prove Kantorovich–Rubinstein duality in one dimension, where the argument also delivers a closed-form formula requiring no optimization at all; Peyré and Cuturi (2019) give the proof in higher dimensions. The intuition is simply to count how much mass crosses each point $t \in \mathbb{R}$. To the left of t (including t), P holds mass $F_P(t)$ and Q holds mass $F_Q(t)$, so any valid transport plan must move at least $|F_P(t) - F_Q(t)|$ of mass across t. Moreover, a parcel moved from x to y crosses every t between x and y, so its distance $|x - y|$ equals the length of the interval of points it crosses. Summing over all parcels, the total transport cost is the integral over t of the total mass crossing t.

Proposition (W_1 on the line). *For distributions P and Q on $\mathbb{R}$ with CDFs F_P, F_Q and finite first moments,*

$$W_1(P, Q) = \int_{-\infty}^{\infty} |F_P(t) - F_Q(t)|\, dt = \sup_{\|g\|_{\mathrm{Lip}} \le 1} \big(\mathbb{E}_P[g] - \mathbb{E}_Q[g]\big), \tag{37.41}$$

so Kantorovich–Rubinstein duality holds on the real line.

Proof. *Step 1: Every plan costs at least the area between the CDFs.* For $(X, Y) \sim \gamma$, write $|X - Y| = \int \mathbf{1}(t \text{ lies between } X \text{ and } Y)\, dt$. Because the integrand is non-negative, we may exchange expectation and integration:

$$\begin{aligned} \mathbb{E}_\gamma |X - Y| &= \int \Big(\gamma(X \le t < Y) + \gamma(Y \le t < X) \Big)\, dt \\ &\ge \int \big| \gamma(X \le t < Y) - \gamma(Y \le t < X) \big|\, dt. \end{aligned}$$

The key step is simplifying the difference inside the last integral. Adding and subtracting $\gamma(X \le t, Y \le t)$ gives $\gamma(X \le t < Y) - \gamma(Y \le t < X) = \gamma(X \le t) - \gamma(Y \le t) = F_P(t) - F_Q(t)$, since γ has marginals P and Q. Thus every transport plan costs at least $\int |F_P - F_Q|$.

Step 2: One plan achieves this lower bound. Match quantiles directly: send the lowest tenth of P to the lowest tenth of Q, and so on up the line. Formally, let U be uniform on $(0, 1)$

and set $X = F_P^{-1}(U)$ and $Y = F_Q^{-1}(U)$, where the quantile function $F^{-1}(\lambda) = \inf\{x : F(x) \geq \lambda\}$ for $\lambda \in (0,1)$ satisfies $F^{-1}(\lambda) \leq t$ if and only if $\lambda \leq F(t)$. This defines a valid coupling because $\Pr(X \leq t) = \Pr(U \leq F_P(t)) = F_P(t)$ and similarly for Y. We call it the *monotone plan*, since X and Y increase together. Under this coupling, $\gamma(X \leq t < Y) = \Pr\left(F_Q(t) < U \leq F_P(t)\right)$ and $\gamma(Y \leq t < X) = \Pr\left(F_P(t) < U \leq F_Q(t)\right)$. At every t, at least one of these two probabilities is zero, meaning the monotone plan never sends mass in both directions across the same point. The inequality in Step 1 therefore holds with equality, so the minimum transport cost equals the area between the CDFs.

Step 3: No 1-Lipschitz test function beats any plan. This *weak duality* argument holds in every dimension. For any 1-Lipschitz function g and any coupling γ, we have $g(x) - g(y) \leq \|x - y\|$ pointwise; taking the expectation under γ immediately gives $\mathbb{E}_P[g] - \mathbb{E}_Q[g] \leq \mathbb{E}_\gamma \|x - y\|$ (compare the weak duality of Section 35.7.1).

Step 4: One 1-Lipschitz test function achieves the area. It suffices to consider functions of the form $g(x) = g(0) + \int_0^x g'(t)\,dt$ with $|g'| \leq 1$: every such g is 1-Lipschitz, and Step 3 already bounds all others. The constant $g(0)$ cancels in $\mathbb{E}_P[g] - \mathbb{E}_Q[g]$. Rewriting $\int_0^X g'(t)\,dt = \int_0^\infty g'(t)\,\mathbf{1}(X > t)\,dt - \int_{-\infty}^0 g'(t)\,\mathbf{1}(X \leq t)\,dt$ and exchanging expectation and integration (justified by Fubini's theorem from Section 34.6, given the finite first moments) yields

$$\mathbb{E}_P[g] - \mathbb{E}_Q[g] = \int_{-\infty}^{\infty} g'(t)\,\left(F_Q(t) - F_P(t)\right)\,dt.$$

Choosing $g'(t) = \operatorname{sign}(F_Q(t) - F_P(t))$ turns the right-hand side into $\int |F_P(t) - F_Q(t)|\,dt$. Combining all four steps, the supremum over 1-Lipschitz functions is at least the area (Step 4) and at most the minimum plan cost (Step 3), which equals the area (Steps 1 and 2). ■

On the real line, then, W_1 is simply the area between the two CDFs, and for two equal-sized samples it reduces to the mean absolute difference of the sorted values (Exercise 2a). It is remarkable that a full linear program collapses so cleanly. Let us verify this collapse on the six-point example from Figure 37.10. Across the five intervals between adjacent atoms, the CDF gaps are 0.25, 0.35, 0.5, 0.35, and 0.25, summing to 1.7, the shaded area in the figure. The code below computes W_1 in three ways: by summing the five CDF gaps, by solving the 36-variable linear program over 6×6 transport matrices with row sums p and column sums q, and by constructing the monotone plan of Step 2 (where each route carries the overlap of the corresponding quantile intervals) and evaluating its cost. Finally, it checks whether the linear program returned the monotone plan.

```
atoms = np.array([0.0, 1.0, 2.0, 3.0, 4.0, 5.0])
p_w = np.array([0.30, 0.20, 0.25, 0.10, 0.10, 0.05])
q_w = np.array([0.05, 0.10, 0.10, 0.25, 0.20, 0.30])

F_p, F_q = np.cumsum(p_w), np.cumsum(q_w)
w1_cdf = np.sum(np.abs(F_p - F_q)[:-1] * np.diff(atoms))

C = np.abs(atoms[:, None] - atoms[None, :])      # cost matrix |x_i - y_j|
n_atoms = len(atoms)
A_eq = np.zeros((2 * n_atoms, n_atoms * n_atoms))
for i in range(n_atoms):
```

```
    A_eq[i, i * n_atoms:(i + 1) * n_atoms] = 1   # row sums    = p
    A_eq[n_atoms + i, i::n_atoms] = 1            # column sums = q
res = optimize.linprog(C.ravel(), A_eq=A_eq,
                       b_eq=np.concatenate([p_w, q_w]), method='highs')
plan_lp = res.x.reshape(n_atoms, n_atoms)
# Monotone plan (Step 2): source i holds the quantiles (F_p[i-1], F_p[i]],
# destination j holds (F_q[j-1], F_q[j]]; route i -> j carries their overlap
lo_p, lo_q = F_p - p_w, F_q - q_w
mono = np.clip(np.minimum(F_p[:, None], F_q[None, :])
               - np.maximum(lo_p[:, None], lo_q[None, :]), 0, None)
print(f'W1 from the CDF gaps     : {w1_cdf:.4f}')
print(f'W1 from the LP           : {res.fun:.4f}')
print(f'cost of the monotone plan: {np.sum(mono * C):.4f}')
print(f'LP plan = monotone plan  : {np.allclose(plan_lp, mono)}, '
      f'mass the LP plan moves leftward: {np.tril(plan_lp, -1).sum():.2f}')
```

```
W1 from the CDF gaps     : 1.7000
W1 from the LP           : 1.7000
cost of the monotone plan: 1.7000
LP plan = monotone plan  : False, mass the LP plan moves leftward: 0.00
```

All three methods agree on $W_1 = 1.7$: summing five CDF gaps gives the exact same answer as solving a 36-variable linear program. The last line, however, holds a small surprise: the linear program did not return the monotone plan. Instead, it found a different plan with the exact same cost, which likewise moves zero mass to the left.

The optimal W_1 plan is therefore not unique here, and the reason is easy to spot. Because $F_P \geq F_Q$ everywhere, no mass ever needs to move leftward. For *any* plan that moves mass strictly rightward ($y \geq x$), we have $|x - y| = y - x$, so its total cost equals the difference of the two means, $3.35 - 1.65 = 1.7$, regardless of how the routes cross. The monotone plan drawn in the figure is simply one of many equally good plans.

Two tools that appear elsewhere in the book replace the absolute distance with its square: the Fréchet inception distance for evaluating image generators (Section 16.5), and mini-batch optimal transport in flow matching, which pairs noise with data by minimizing squared distance (Section 38.7). The **Wasserstein-2 distance** averages the squared distance and takes the square root afterwards: $W_2(P, Q) = \left(\inf_\gamma \mathbb{E}_\gamma \|x - y\|^2\right)^{1/2}$. Under a squared cost, crossing two transport ribbons always costs strictly more than uncrossing them, so on the line the monotone plan becomes the *unique* optimum (Exercise 2d). Between two one-dimensional Gaussians with means μ_1, μ_2 and standard deviations σ_1, σ_2, it evaluates cleanly to $W_2^2 = (\mu_1 - \mu_2)^2 + (\sigma_1 - \sigma_2)^2$ (Exercise 4a). The Fréchet inception distance is the multivariate counterpart of this formula applied to Gaussians fitted to image features, with covariance matrices replacing the variances.

37.4.2.4 Entropic Regularization and Sinkhorn Iterations

This subsection may be skipped on a first reading. Beyond one dimension, there is in general no closed-form CDF formula, and the linear program over N points on each side has N^2 variables. The standard way to make optimal transport fast and differentiable is **entropic regularization** (Cuturi 2013): given a cost matrix $\mathbf{C}$ and a regularization strength $\varepsilon > 0$, we add $\varepsilon \sum_{ij} \gamma_{ij}(\log \gamma_{ij} - 1)$ to the transport cost $\sum_{ij} \gamma_{ij} C_{ij}$. Because any valid plan satisfies

$\sum_{ij}\gamma_{ij} = 1$, this extra term equals $-\varepsilon$ times the entropy of γ plus a constant; it discourages plans that concentrate all their mass on a few sharp routes. Entropic regularization is what turned optimal transport into a practical loss function for deep learning, enabling generative models to be trained directly on regularized transport costs (Genevay et al. 2018).

Remarkably, the regularized problem solves almost in closed form. Forming the Lagrangian (Section 35.7.1) with a multiplier ξ_i for each row-sum constraint $\sum_j \gamma_{ij} = p_i$ and a multiplier ζ_j for each column-sum constraint $\sum_i \gamma_{ij} = q_j$, and setting the derivative with respect to γ_{ij} to zero, we obtain $C_{ij} + \varepsilon \log \gamma_{ij} - \xi_i - \zeta_j = 0$, or $\gamma_{ij} = e^{-(C_{ij}-\xi_i-\zeta_j)/\varepsilon}$. Each route carries mass decaying exponentially in how far its cost C_{ij} exceeds $\xi_i + \zeta_j$, measured in units of ε. In matrix form,

$$\gamma_\varepsilon = \operatorname{diag}(\mathbf{a})\,\mathbf{K}\,\operatorname{diag}(\mathbf{b}), \qquad \mathbf{K} = e^{-\mathbf{C}/\varepsilon} \text{ (entrywise)}, \tag{37.42}$$

where $\mathbf{a} = e^{\xi/\varepsilon}$ and $\mathbf{b} = e^{\zeta/\varepsilon}$ are positive scaling vectors. The regularized plan is simply the *Gibbs kernel* $\mathbf{K}$ (a matrix of positive weights, unrelated to the kernels of MMD), rescaled row by row and column by column. Only $\mathbf{a}$ and $\mathbf{b}$ remain to be found, and each set of marginal constraints determines one of them: with $\mathbf{b}$ fixed, setting $\mathbf{a} = \mathbf{p}/(\mathbf{K}\mathbf{b})$ (entrywise) enforces the row sums $\mathbf{p}$; with $\mathbf{a}$ fixed, setting $\mathbf{b} = \mathbf{q}/(\mathbf{K}^\top\mathbf{a})$ enforces the column sums $\mathbf{q}$.

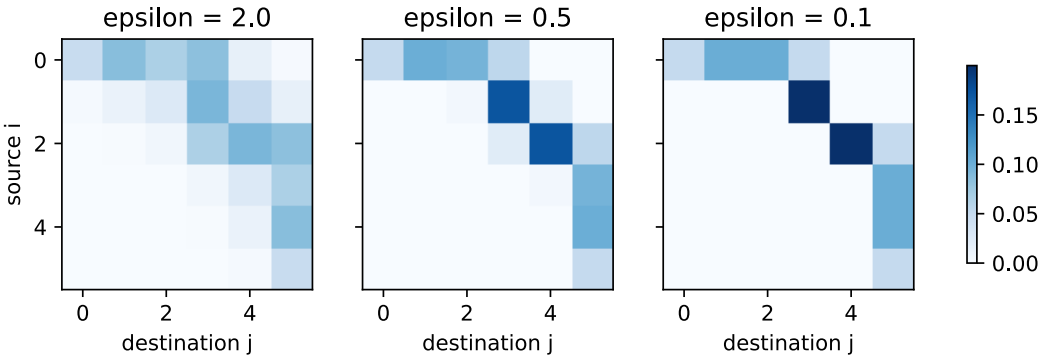
The **Sinkhorn iterations** alternate these two updates, updating $\mathbf{b}$ and then $\mathbf{a}$ until both marginals match. After the $\mathbf{a}$ update the row sums are exact, so we stop once the column sums match $\mathbf{q}$ within 10^{-9}. Every step requires only two matrix-vector products, making the algorithm fast on GPUs and trivial to differentiate through. Below we report the *entropic cost*, meaning the transport cost $\sum_{ij}(\gamma_\varepsilon)_{ij}C_{ij}$ of the regularized plan (without the entropy penalty). As $\varepsilon \to 0$, this cost approaches the exact unregularized optimum, but convergence slows dramatically (Peyré and Cuturi 2019) and the entries of $\mathbf{K}$ can underflow; production implementations therefore work in log-space on $\log \mathbf{a}$ and $\log \mathbf{b}$ (Exercise 5b).

To see how small ε must be, and how expensive a small ε becomes, we apply Sinkhorn to the six-point example with the squared cost $C_{ij} = (x_i - x_j)^2$, where the monotone plan of Figure 37.10 is the unique optimum and its ten active routes cost $W_2^2 = 3.5$. The cell below verifies this optimum with a linear program and runs Sinkhorn at three values of ε, reporting the iteration count, entropic cost, and number of routes carrying mass above 10^{-3}.

```
def sinkhorn(p, q, C, eps, tol=1e-9, max_iters=10 ** 6):
    """Entropic OT by Sinkhorn iterations, stopped once the column sums are
    within tol of q (L1); returns the plan, its cost and the iterations."""
    K = np.exp(-C / eps)                          # the Gibbs kernel
    a = np.ones_like(p)
    for it in range(1, max_iters + 1):
        b = q / (K.T @ a)                         # match the column sums
        a = p / (K @ b)                           # match the row sums ...
        if np.abs(b * (K.T @ a) - q).sum() < tol:  # ... columns still right?
            break
    plan = a[:, None] * K * b[None, :]
    return plan, np.sum(plan * C), it

C2 = (atoms[:, None] - atoms[None, :]) ** 2       # squared cost (x_i - y_j)^2
```

```
res2 = optimize.linprog(C2.ravel(), A_eq=A_eq,
                        b_eq=np.concatenate([p_w, q_w]), method='highs')
print(f'squared cost, LP optimum : {res2.fun:.5f}')
plans, eps_values = [], (2.0, 0.5, 0.1)
for eps in eps_values:
    plan, cost, iters = sinkhorn(p_w, q_w, C2, eps)
    plans.append(plan)
    print(f'epsilon = {eps}: {iters:6d} iterations, entropic cost = '
          f'{cost:.5f}, routes above 1e-3 = {(plan > 1e-3).sum()}')
d2l.show_heatmaps(d2l.tensor(np.stack(plans)[None]), 'destination j',
                  'source i', titles=[f'epsilon = {e}' for e in eps_values],
                  figsize=(9, 2.8), cmap='Blues')
```

```
squared cost, LP optimum : 3.50000
epsilon = 2.0:     47 iterations, entropic cost = 4.06489, routes above 1e-3 = 25
epsilon = 0.5:    186 iterations, entropic cost = 3.56963, routes above 1e-3 = 14
epsilon = 0.1: 154325 iterations, entropic cost = 3.50002, routes above 1e-3 = 10
```



The linear program confirms $W_2^2 = 3.5$. At $\varepsilon = 2$, the regularized plan smears each source's mass across many destinations (25 of the 36 routes carry more than 10^{-3}), pushing the cost up to 4.06. At $\varepsilon = 0.5$, the plan tightens to 14 active routes with cost 3.57. By $\varepsilon = 0.1$, only the ten routes of the monotone plan survive, and the cost reaches 3.50002, within 2×10^{-5} of the exact optimum. In the heatmaps, a blurry band sharpens into a clean staircase. The natural scale for ε is the cost gap between competing routes, which is of order 1 on this grid; setting ε to a tenth of that scale recovers the optimum accurately.

The iteration counts reveal the catch. Sinkhorn needed about 50 iterations at $\varepsilon = 2$ and about 190 at $\varepsilon = 0.5$, but roughly 150,000 at $\varepsilon = 0.1$. Cutting ε by a factor of five increased the work by a factor of more than 800. Sinkhorn iterations are fast only when you can afford a blurred transport plan (Exercise 6b tracks this growth quantitatively).

37.4.3 Fixed Kernels and Trained Critics

MMD and W_1 split our two requirements right down the middle. A fixed kernel defines a small, smooth function class: MMD reduces to three sample averages, and its estimation error from N points decays as $O(N^{-1/2})$ in *any* dimension (Gretton et al. 2012). Kernels do not escape the curse of dimension that way, unfortunately. The error rate is dimension-free, but the size of the signal is not: with a fixed kernel, the population MMD between two

given distributions typically shrinks as the dimension grows (Ramdas et al. 2015), which is why the kernel inception distance and MMD losses are computed on learned features, not on raw pixels. A fixed kernel also loses sight of the data beyond a few length scales. By contrast, 1-Lipschitz functions provide a steady gradient at every separation, which is exactly what a model far from its data needs. However, even though they cannot jump like indicators, the class of all 1-Lipschitz functions is still enormous in high dimensions: for two samples in dimension $n > 2$, the empirical W_1 error decays only as $O(N^{-1/n})$ (Weed and Bach 2019). In 100 dimensions, halving that estimation error would require 2^{100} times as much data. Nobody estimates W_1 directly from high-dimensional samples that way. Instead, a WGAN critic searches over a much smaller class of neural networks, while Sinkhorn iterations handle smaller discrete matching problems.

Both kernels and adversarial critics start by drawing samples from the model. For some generative models, sampling itself is the bottleneck, and Section 37.5 asks how we can compare two distributions without drawing model samples at all.

37.4.4 Summary

Overlapping tails do not rescue f-divergences: when two unit Gaussians sit eight standard deviations apart, both the Jensen–Shannon divergence and total variation have slopes near 10^{-4}. Integral probability metrics sidestep density ratios by comparing how P and Q average a class of test functions, and the choice of class determines both sample efficiency and how far the metric can see. Test functions linear in fixed features yield MMD, the distance between mean feature vectors: it evaluates in closed form as three kernel averages with no critic to train, and estimates at rate $O(N^{-1/2})$ regardless of dimension, though the MMD itself tends to shrink as the dimension grows and an RBF kernel goes blind beyond a few length scales. Functions of slope at most one yield W_1, the minimum mass-times-distance cost of transporting P into Q, which maintains slope one at every separation and underpins the Wasserstein GAN critic. In one dimension, W_1 collapses to the area between the two CDFs; in higher dimensions, it requires either a trained critic or, on small discrete problems, Sinkhorn iterations with entropic regularization.

37.4.5 Exercises

1. code **MMD in closed form.** Use the Gaussian integral Equation 37.38 for the RBF kernel with length scale ℓ, with $P = \mathcal{N}(0, 1)$ and $Q = \mathcal{N}(d, 1)$.
    1. Show that $\text{MMD}^2 = \frac{2\ell}{\sqrt{\ell^2+2}}\big(1 - e^{-d^2/(2(\ell^2+2))}\big)$, and check that $\ell = 1$ gives the MMD^2 column of the opening table. Which length scale maximizes MMD^2 for small shifts d? What happens as $\ell \to 0$ and as $\ell \to \infty$?
    2. Evaluate the slope in d at $d = 8$ for $\ell = 1, 4$ and 16. What do the three values say about the length scale for a model that starts far from its data?
2. code **One-dimensional transport.** ● On the real line, W_1 is the area between the CDFs Equation 37.41.
    1. For two empirical distributions on samples $x_1, \dots, x_N$ and $y_1, \dots, y_N$ of equal size, use the monotone plan of the proof to show that $W_1 = \frac{1}{N}\sum_i |x_{(i)} - y_{(i)}|$, the mean absolute difference of the sorted values. Check this against the linear program of the six-point example on two random samples of size 7.

2. Compute the sorted-sample W_1 between `x_p` and `y_shift` and between `x_p` and `y_same` of the MMD experiment. Then average it over 20 pairs of independent samples of size N from $\mathcal{N}(0,1)$, for $N \in \{10^2, 10^3, 10^4\}$, and find the rate at which it decreases. The estimate is never negative. Why does this rule out unbiasedness when $P = Q$?
3. Change `q_w` to $(0.40, 0.05, 0.05, 0.10, 0.10, 0.30)$, so that the two CDFs cross, and rerun the cell that computes W_1. Build the test function g with $g(0) = 0$ and slope $\text{sign}(F_Q(t) - F_P(t))$ between neighboring atoms, as in Step 4 of the proof, and check that $\mathbb{E}_P[g] - \mathbb{E}_Q[g]$ equals the LP value. For every route (i, j) that carries mass in the LP solution `plan_lp`, compare $g(x_i) - g(x_j)$ with $|x_i - x_j|$. Why must the two agree on every such route for any optimal plan and any optimal test function?
4. *Uniqueness under the squared cost.* Suppose a plan moves mass from x to y' and from x' to y, with $x < x'$ and $y < y'$, so that the two ribbons cross. Show that under the squared cost, sending the same mass from x to y and from x' to y' saves $2(x' - x)(y' - y)$ per unit of mass, and deduce that the monotone plan is the only optimal one. What does the same exchange save under the cost $|x - y|$ when all four moves go rightward? Find two ribbons that cross in `plan_lp`, the plan that the linear program returned for the six-point example, and check your answer on them.

3. **Optimal critics on the line.** Step 4 of the proof of Equation 37.41 shows that a test function with slope $g'(t) = \text{sign}(F_Q(t) - F_P(t))$ attains W_1.

1. For $P = \mathcal{N}(0,1)$ and $Q = \mathcal{N}(d,1)$ with $d > 0$, show that $F_P > F_Q$ everywhere, so that Step 4 returns $g(x) = -x$, the critic that the text found by inspection.
2. For $P = \mathcal{N}(0,1)$ and $Q = \mathcal{N}(0,4)$, show that the two CDFs cross only at 0, that $g(x) = -|x|$ is optimal, and that $W_1 = \sqrt{2/\pi} \approx 0.80$. (For $X \sim \mathcal{N}(0, \sigma^2)$, $\mathbb{E}|X| = \sigma\sqrt{2/\pi}$.) The linear kernel reported 0 on this pair. Which way does the critic move the samples of Q?
3. Let $P = \frac{1}{2}\delta_0 + \frac{1}{2}\delta_3$ and $Q = \frac{1}{2}\delta_1 + \frac{1}{2}\delta_3$. Find $W_1(P, Q)$ and describe every optimal critic of the form used in Step 4. Then show in general that such a critic is optimal if and only if $g'(t) = \text{sign}(F_Q(t) - F_P(t))$ at almost every t with $F_P(t) \neq F_Q(t)$, the points that mass has to cross, while its slope elsewhere is free.
4. A WGAN critic is often trained with a penalty on $(|g'(x)| - 1)^2$ at points x between the data and the generated samples (Gulrajani et al. 2017). Explain this choice by (c). Which critics are optimal when $P = Q$, and which of them does the penalty prefer? Section 16.4 shows what follows for training.

4. code W_2 **between Gaussians and the Fréchet distance.** On the line the monotone plan $X = F_P^{-1}(U)$, $Y = F_Q^{-1}(U)$ is optimal for the squared cost.

1. Show that for $P = \mathcal{N}(\mu_1, \sigma_1^2)$ and $Q = \mathcal{N}(\mu_2, \sigma_2^2)$ the monotone plan is $Y = \mu_2 + (\sigma_2/\sigma_1)(X - \mu_1)$, and deduce that $W_2^2(P, Q) = (\mu_1 - \mu_2)^2 + (\sigma_1 - \sigma_2)^2$. With the same plan, recover $W_1 = |\mu_1 - \mu_2|$ for equal variances, the W_1 column of the opening table.
2. Let both standard deviations be σ and the means d apart. Then $W_2^2 = d^2$ for every σ, while the KL divergence between the two Gaussians is $d^2/(2\sigma^2)$. What do the two report as $\sigma \to 0$, when the Gaussians become the point masses of

Table 37.2?

3. In $\mathbb{R}^n$ the squared W_2 between $\mathcal{N}(\mu_1, \Sigma_1)$ and $\mathcal{N}(\mu_2, \Sigma_2)$ is the squared *Fréchet distance* $\|\mu_1 - \mu_2\|^2 + \operatorname{tr}(\Sigma_1 + \Sigma_2 - 2\,(\Sigma_1^{1/2}\Sigma_2\Sigma_1^{1/2})^{1/2})$ (Dowson and Landau 1982). Check that it reduces to (a) for $n = 1$ and, for diagonal covariance matrices, to the sum of (a) over the coordinates. Prove the diagonal case directly: the squared cost is a sum over coordinates, so no plan costs less than the sum of the coordinatewise optima, and the monotone plans of the coordinates together attain it.
4. The Fréchet inception distance fits a Gaussian to each of two feature distributions and reports the squared Fréchet distance between the fits. Let P be the mixture $0.7\,\mathcal{N}(-2, 0.6^2) + 0.3\,\mathcal{N}(2, 0.6^2)$ and $Q = \mathcal{N}(-0.8, 3.72)$, which has the same mean and variance. What is the Fréchet distance between the two fits? Estimate $W_2^2(P, Q)$ from 10^5 sorted draws of each (you should find about 0.63). What does a Fréchet distance of zero certify, and what does it not?

5. code **Entropic transport and Sinkhorn divergences.** ●● Regularize the transport LP of the six-point example by the divergence from the independent plan,

$$\mathrm{OT}_\varepsilon(P, Q) = \min_\gamma \sum_{i,j} \gamma_{ij} C_{ij} + \varepsilon\, D_{\mathrm{KL}}(\gamma \,\|\, p\, q^\top),$$

where γ ranges over the plans with row sums p and column sums q.

1. Show that $D_{\mathrm{KL}}(\gamma \,\|\, p\, q^\top) = -H(\gamma) + H(P) + H(Q)$ for every such plan, where H denotes the entropy, so that `sinkhorn` returns the minimizer. Show that $\mathrm{OT}_\varepsilon(P, Q) \to \sum_{i,j} p_i q_j C_{ij}$ as $\varepsilon \to \infty$, and that $\mathrm{OT}_\varepsilon(P, P) > 0$ for every $\varepsilon > 0$ unless P is a point mass. With $C_{ij} = |x_i - x_j|$, show that the distributions P on the atoms that minimize the limit $\sum_{i,j} p_i q_j C_{ij}$ put all their mass on medians of Q, for `q_w` the atoms 3 and 4. What would a model trained on OT_ε with a large ε learn?
2. The Sinkhorn divergence $S_\varepsilon(P, Q) = \mathrm{OT}_\varepsilon(P, Q) - \frac{1}{2}\mathrm{OT}_\varepsilon(P, P) - \frac{1}{2}\mathrm{OT}_\varepsilon(Q, Q)$ removes this bias (Genevay et al. 2018; Feydy et al. 2019). Compute $S_\varepsilon(P, Q)$ for `p_w` and `q_w` with the absolute cost `C` at $\varepsilon \in \{10^{-3}, 10^{-2}, \dots, 10^{3}\}$. At $\varepsilon = 10^{-3}$ the entries of $\mathbf{K}$ underflow, so carry out the Sinkhorn updates on $\log \mathbf{a}$ and $\log \mathbf{b}$ there, with `scipy.special.logsumexp`. Check that S_ε approaches W_1 as $\varepsilon \to 0$ and $\frac{1}{2}\mathcal{E}(P, Q)$ as $\varepsilon \to \infty$, where $\mathcal{E}(P, Q) = 2\mathbb{E}|X - Y| - \mathbb{E}|X - X'| - \mathbb{E}|Y - Y'|$ is the energy distance (Székely and Rizzo 2013), the expression Equation 37.37 with $-|x - y|$ in place of the kernel. Derive the second limit from (a).

6. code **Sinkhorn iterations at small ε.** ● Use `sinkhorn`, `p_w` and `q_w` of the six-point example.

1. *Stopping early.* Call `sinkhorn` with the squared cost `C2` at $\varepsilon = 0.1$ and `max_iters=5000`, so that the loop ends before the column sums have settled. The cost that it returns is below the optimum 3.5. Print the column sums of the returned matrix minus `q_w`. Which destinations receive too little, and why does that lower the cost? Why can no matrix with row sums `p_w` and column sums `q_w` cost less than 3.5?
2. *The number of iterations.* Count the iterations that `sinkhorn` needs with `C2` at $\varepsilon \in \{2, 1, 0.5, 0.3, 0.2, 0.15, 0.1\}$, and plot the logarithm of the count against $1/\varepsilon$. What kind of growth do the points at small ε show, and what count does it predict for $\varepsilon = 0.05$?

3. *Ties.* Under the absolute cost `C` every plan that moves mass only rightward is optimal for `p_w` and `q_w`. Run `sinkhorn` with `C` at $\varepsilon = 0.1$ and 0.02. How many routes carry mass? Show that the limit plan has the form $\gamma_{ij} = a_i b_j$ for $j \geq i$ and 0 otherwise. On plans the regularizer is $-\varepsilon$ times the entropy plus a constant. Use this to explain why the limit is the plan of largest entropy among the optimal plans.

37.5 Score-Based Comparisons

A softmax layer turns ten logits into probabilities by exponentiating them and dividing by their sum, and nobody gives that sum a second thought. Now imagine letting a neural network assign a score not to ten classes, but to every possible input. By convention the sign is reversed: the network outputs an energy $E(\mathbf{x})$ that is low for plausible inputs (where a logit would be high) and high for implausible ones, defining a density proportional to $\tilde{q}(\mathbf{x}) = e^{-E(\mathbf{x})}$. The sum over ten classes becomes an integral over all possible inputs, $Z = \int \tilde{q}(\mathbf{x})\, d\mathbf{x}$, which in high dimensions is completely intractable. Such an *energy-based model* can tell us how plausible an input is relative to another, but not its actual probability.

None of the distribution comparisons we have developed so far can handle this situation. Maximum likelihood requires the normalized density $q = \tilde{q}/Z$. Sample-based methods, whether built from a critic (Section 37.3.4), a kernel, or a transport plan (Section 37.4), avoid Z, but they require drawing samples from the model, and an energy function does not hand out samples. Each sample must be generated by a long random walk taking many small noisy steps toward lower energy, with every step requiring a gradient through the network (Section 38.6.2).

Computing a single gradient, however, is cheap: it costs just one backward pass, and it turns out to be all we need. The gradient of $\log \tilde{q}$ with respect to the input $\mathbf{x}$, which measures the slope of the log-density, eliminates Z completely because a constant has zero slope. This gradient is called the *score*, and in this section we compare distributions using scores alone. The mean squared difference between two scores gives the objective of score matching, which underlies diffusion models (Chapter 17). Yet comparing slopes comes with an important blind spot that anyone training a score-based model should understand: because a slope is purely local, it cannot tell how probability mass is divided between modes that lie far apart. We will measure just how blind it is.

As in the previous sections, P denotes the data distribution, which we observe only through samples, and Q denotes the model, which we now know only through its score s_Q. We ask two central questions: what does a difference of slopes tell us about two distributions (and what does it miss), and how can we compare slopes at all when the data, a finite set of points, have no slope? Section 38.6.1 builds directly on these two answers. We also explore a third question, how to test whether a sample came from a model we cannot sample from, which can be skipped on a first reading. Finally, we pull all the comparisons from this chapter together into a table of training objectives and practical guidance on choosing among them.

```
from scipy import integrate
```

Our running examples are one-dimensional Gaussians and mixtures of two Gaussians. We start with three small helpers: `log_components` and `log_mix` evaluate the log-density of a Gaussian mixture stably by factoring out the largest term so the tails do not underflow (Sec-

tion 35.8.2), and `kl` computes the KL divergence between two densities on a grid in nats.

```
def log_components(x, w, mus, sigmas):
    """log(w_k N(x; mu_k, sigma_k^2)) for every component k; shape (k, len(x))."""
    return (np.log(w)[:, None] - 0.5 * np.log(2 * np.pi * sigmas[:, None] ** 2)
            - (x - mus[:, None]) ** 2 / (2 * sigmas[:, None] ** 2))

def log_mix(x, w, mus, sigmas):
    """Log-density of the mixture sum_k w_k N(mu_k, sigma_k^2), computed stably."""
    c = log_components(x, w, mus, sigmas)
    c_max = c.max(axis=0)
    return c_max + np.log(np.exp(c - c_max).sum(axis=0))

def kl(log_a, log_b, x):
    """D_KL(A||B) by quadrature on the grid x; in nats."""
    return integrate.trapezoid(np.exp(log_a) * (log_a - log_b), x)
```

37.5.1 The Score and the Fisher Divergence

37.5.1.1 The Score

At every point in space, the score answers a simple geometric question: in which direction does the density rise, and how steeply? Formally, for a distribution P with a differentiable density p, the **score** is the gradient of the log-density *with respect to the data point*:

$$s_P(\mathbf{x}) = \nabla_{\mathbf{x}} \log p(\mathbf{x}). \tag{37.43}$$

Be careful not to confuse this with classical statistics, where "score" refers to the gradient of the log-likelihood with respect to the *parameters* (Section 36.6); here, we differentiate with respect to $\mathbf{x}$. For a one-dimensional Gaussian $\mathcal{N}(\mu, \sigma^2)$, the score is simply

$$s(x) = -\frac{x - \mu}{\sigma^2},$$

which acts like a spring pulling x back toward the mean μ with stiffness $1/\sigma^2$. For a mixture $p = \sum_k w_k\, p_k$, applying the chain rule gives $s(x) = \sum_k r_k(x)\, s_k(x)$, where $r_k(x) = w_k p_k(x)/p(x)$ is the posterior probability, or *responsibility*, that x came from component k. Each point therefore feels a weighted combination of the component springs. Near any mode, that mode's responsibility is close to 1, so the point feels that mode's spring alone; along the valley between modes, the responsibilities smoothly hand control from one attractor to the next, as illustrated in Figure 37.11. We implement this formula directly below, computing the responsibilities in log space for numerical stability.

```
def mix_score(x, w, mus, sigmas):
    """Score of a Gaussian mixture: responsibility-weighted component scores."""
    r = np.exp(log_components(x, w, mus, sigmas) - log_mix(x, w, mus, sigmas))
    return np.sum(r * -(x - mus[:, None]) / sigmas[:, None] ** 2, axis=0)
```

What makes the score so useful for energy-based models is that it ignores any normalizing constant. If $q(\mathbf{x}) = \tilde{q}(\mathbf{x})/Z$, then

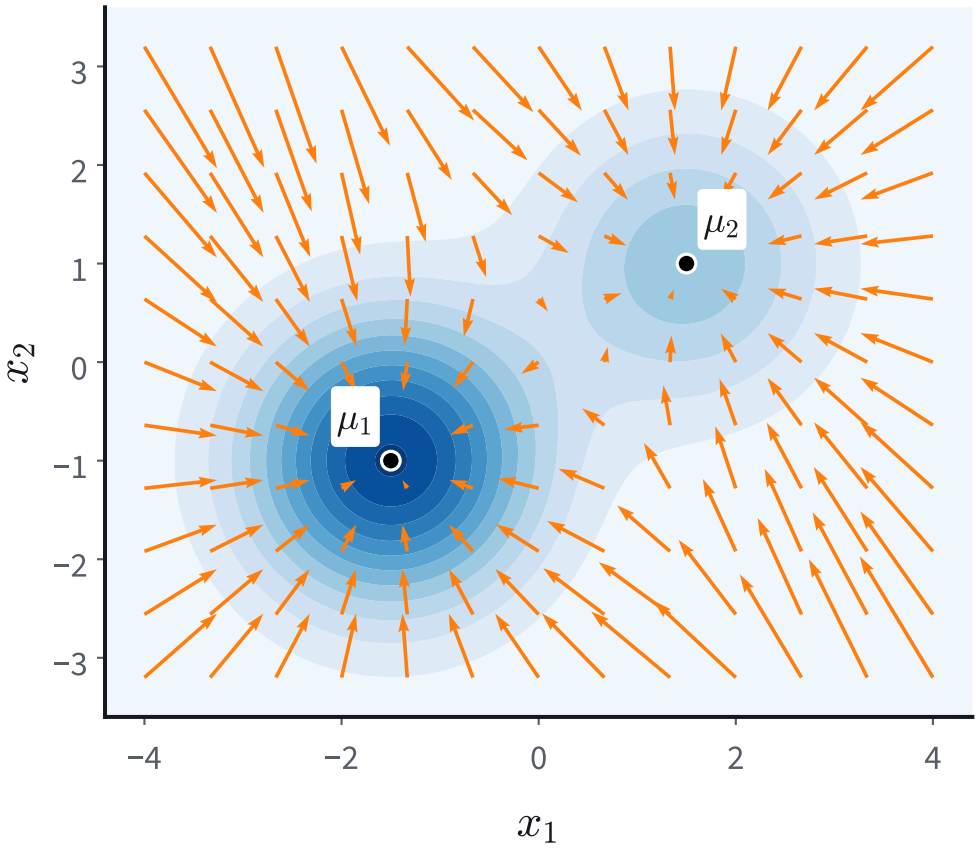


Figure 37.11: The score field of a two-component Gaussian mixture. Arrows show the gradient of the log-density: near each mode the field acts as a spring pulling toward that mode's mean, and along the valley between modes the responsibilities hand points off from one attractor to the other. Density contours are underlaid for reference.

$$\nabla_{\mathbf{x}} \log q(\mathbf{x}) = \nabla_{\mathbf{x}} \log \tilde{q}(\mathbf{x}) - \nabla_{\mathbf{x}} \log Z = \nabla_{\mathbf{x}} \log \tilde{q}(\mathbf{x}),$$

because Z does not depend on $\mathbf{x}$. For example, the unnormalized quartic density $\tilde{q}(x) = e^{-x^4/4}$ has the score $-x^3$, and we can evaluate it everywhere without ever learning that $Z \approx 2.56$.

37.5.1.2 The Fisher Divergence

Because scores drop Z, any comparison built from scores applies directly to unnormalized models. The most natural choice is the mean squared difference between the two score fields averaged under the data, known as the **Fisher divergence**:

$$D_{\mathrm{F}}(P\|Q) = \frac{1}{2}\, \mathbb{E}_{\mathbf{x}\sim P}\left[\left\| s_P(\mathbf{x}) - s_Q(\mathbf{x}) \right\|^2\right]. \tag{37.44}$$

Clearly $D_{\mathrm{F}}(P\|Q) \geq 0$, and it is a genuine divergence, zero if and only if $P = Q$ (Section 37.3.1.1), provided both densities are strictly positive on the same connected region. On a connected region, matching scores force $\log p - \log q$ to be constant, and because both densities must integrate to 1, that constant must be zero.

Comparing two Gaussians reveals an important difference between the Fisher divergence and the KL divergence. When both distributions have variance σ^2 and their means are a distance Δ apart, their scores differ by the constant Δ/σ^2 at every x, so

$$D_{\mathrm{F}} = \frac{\Delta^2}{2\sigma^4} \qquad \text{whereas} \qquad D_{\mathrm{KL}} = \frac{\Delta^2}{2\sigma^2},$$

using the Gaussian KL formula from Section 37.1. When $\sigma = 1$, the two values match. Now rescale the coordinate x by a factor of 2, doubling both Δ and σ: the KL divergence stays unchanged, whereas the Fisher divergence drops to a quarter of its previous value. Unlike KL divergence, which is dimensionless (measured in nats), the Fisher divergence carries units of $1/x^2$, as a squared slope must. Its numerical value therefore depends on how the data are scaled, so a score-matching loss on one dataset cannot be compared directly with the loss on another, nor interpreted as a coding budget in nats. Exercise 1a works out the case of unequal variances.

37.5.1.3 Mixing Weights of Distant Modes

The expectation in Equation 37.44 averages a difference of slopes over the data distribution P. When the data have multiple well-separated modes, these two ingredients combine badly. Consider two mixtures built from the exact same Gaussian components, but with swapped mixing weights:

$$P_w = w\,\mathcal{N}(-m, 1) + (1-w)\,\mathcal{N}(m, 1), \qquad w = 0.7 \text{ against } w = 0.3.$$

As distributions, these two mixtures are far apart. Once the modes are well separated, their KL divergence approaches that of the discrete weight vectors $(0.7, 0.3)$ and $(0.3, 0.7)$, which is $0.7 \log \frac{7}{3} + 0.3 \log \frac{3}{7} = 0.339$ nats regardless of how large m becomes.

Their scores, however, barely differ at all. Near either mode, that mode's responsibility is close to 1 in *both* mixtures, so both score fields reduce to that mode's spring. Explicitly (as derived in Exercise 2b), the score of P_w is

$$s_w(x) = -x + m\tanh(mx - c_w), \qquad c_w = \tfrac{1}{2}\log\frac{w}{1-w}.$$

This expression behaves like the spring $-(x+m)$ far to the left, the spring $-(x-m)$ far to the right, and transitions between them near $x = c_w/m$. Notice that the mixing weight w enters *only* through the location of that transition. At $m = 5$, the transition sits at $x = 0.085$ for $w = 0.7$ and at $x = -0.085$ for $w = 0.3$, right in the middle of the empty valley between the two modes, where the density is $e^{-m^2/2}/\sqrt{2\pi} \approx 1.5\times10^{-6}$. The data almost never visit the one place where the two scores disagree. Let us compute both divergences by quadrature as m increases; we expect the KL divergence to stay near 0.339 nats while the Fisher divergence collapses roughly like $e^{-m^2/2}$.

```
xs = np.linspace(-14.0, 14.0, 14001)
w_a, w_b, unit = np.array([0.7, 0.3]), np.array([0.3, 0.7]), np.ones(2)
for m in (1.0, 3.0, 5.0):
    mus_m = np.array([-m, m])
    log_a, log_b = log_mix(xs, w_a, mus_m, unit), log_mix(xs, w_b, mus_m, unit)
    score_gap = mix_score(xs, w_a, mus_m, unit) - mix_score(xs, w_b, mus_m, unit)
    fisher = 0.5 * integrate.trapezoid(np.exp(log_a) * score_gap ** 2, xs)
    print(f'm = {m:.0f}: KL = {kl(log_a, log_b, xs):.3f} nats, '
          f'Fisher = {fisher:.1e}')
```

```
m = 1: KL = 0.184 nats, Fisher = 1.1e-01
m = 3: KL = 0.337 nats, Fisher = 7.1e-03
m = 5: KL = 0.339 nats, Fisher = 4.0e-06
```

As expected, the KL divergence settles at 0.339 nats while the Fisher divergence plunges by more than four orders of magnitude, from about 0.1 down to roughly 4×10^{-6}. At $m = 5$ it is on the order of $e^{-m^2/2} = e^{-12.5} \approx 4\times10^{-6}$, just as the nearly empty valley predicted. A loss built purely from scores cannot see how probability mass is divided between distant modes. That is why we insisted on a *connected* region earlier: on two disconnected regions, identical scores allow $\log p - \log q$ to take a different constant on each region. Consequently, a model trained by minimizing the Fisher divergence can get the location and shape of every mode right and still place 30% of its mass where the data place 70% (Wenliang and Kanagawa 2020).

Fortunately, adding noise fills the valley. If we add independent Gaussian noise of standard deviation 2 to both mixtures, each component's variance grows from 1 to 5. Modes that were ten standard deviations apart are now only 4.5 standard deviations apart, raising the density in the valley from 1.5×10^{-6} to roughly 0.015. Now the data actually visit the region where the two scores disagree (Exercise 2c quantifies the gain). This is a major reason why diffusion models perturb data with noise across many scales before matching scores; Section 17.1 measures this exact blind spot on trained neural networks.

37.5.2 Stein's Identity

The Fisher divergence suffers from a second, even more immediate obstacle: it depends on s_P, the score of the data distribution, yet we only observe data as samples. A finite set of points has no slope. To see where s_P enters, let us expand the square in one dimension:

$$D_{\mathrm{F}}(P\|Q) = \tfrac{1}{2}\,\mathbb{E}_P[s_P^2] - \mathbb{E}_P[s_P\, s_Q] + \tfrac{1}{2}\,\mathbb{E}_P[s_Q^2]. \tag{37.45}$$

The third term is an expectation over the data of a function supplied by the model, so we can estimate it directly from samples. The first term does not depend on Q at all, so it is an irrelevant constant when fitting or ranking models. All the trouble sits in the middle cross term, where the unknown data score multiplies the model score. Remarkably, a single identity satisfied by every smooth density eliminates that cross term completely. We state and prove it for a one-dimensional density, and then put it to work twice: first on the data distribution to remove the cross term, and second on the model distribution to test whether a sample came from Q.

Proposition (Stein's identity). *Let $p > 0$ be a differentiable density on $\mathbb{R}$ with score $s = (\log p)'$, and let g be differentiable with $g(x)\,p(x) \to 0$ as $x \to \pm\infty$, $\mathbb{E}_p|g'| < \infty$, and $\mathbb{E}_p|g\,s| < \infty$. Then*

$$\mathbb{E}_{x\sim p}[g'(x) + g(x)\,s(x)] = 0. \tag{37.46}$$

Proof. By definition $s = p'/p$, so $p' = s\,p$. Integrating by parts (Section 34.6.2.3) gives

$$\int g'\,p\,dx = [g\,p]_{-\infty}^{\infty} - \int g\,p'\,dx = -\int g\,s\,p\,dx,$$

where the boundary term vanishes and both integrals converge by the assumptions on g. ∎

For the standard normal distribution, $s(x) = -x$, and Stein's identity reads $\mathbb{E}[g'(X)] = \mathbb{E}[Xg(X)]$. Setting $g(x) = x$, for instance, immediately yields $\mathbb{E}[X^2] = 1$. This Gaussian case is Stein's lemma (Stein 1981), and the general form is due to (Stein 1986). What makes the identity so powerful is that it converts knowledge of a slope into an equation about expectations, and expectations are what samples can estimate.

37.5.2.1 Score Matching

Suppose the data distribution P has a smooth density, even though we only see samples from it. Applying Stein's identity to P (with $s = s_P$) and choosing the model's score as the test function ($g = s_Q$) gives $\mathbb{E}_P[s_Q' + s_Q\,s_P] = 0$. The cross term in Equation 37.45 therefore becomes $-\mathbb{E}_P[s_Q']$, eliminating s_P entirely:

$$D_{\mathrm{F}}(P\|Q) = \mathbb{E}_{x\sim P}\Big[\tfrac{1}{2}\,s_Q(x)^2 + s_Q'(x)\Big] + \tfrac{1}{2}\,\mathbb{E}_P[s_P^2]. \tag{37.47}$$

Integration by parts has shifted the derivative off the data, which have none, and onto the model. Since the trailing term $\frac{1}{2}\mathbb{E}_P[s_P^2]$ is the same for every model, minimizing the Fisher divergence is equivalent to minimizing the first expectation: an average over data points of quantities the model itself supplies, requiring neither Z nor s_P. This procedure

is called *score matching*; Hyvärinen (2005) introduced it specifically to fit unnormalized models, demonstrating it on natural image patches.

Our unnormalized quartic from Section 37.5.1 shows how cleanly this works in practice. Introduce a scale parameter $\theta > 0$ via $\tilde{q}_\theta(x) = e^{-\theta x^4/4}$, which gives $s_Q(x) = -\theta x^3$ and $s'_Q(x) = -3\theta x^2$. Writing a bar for the sample average over the data, the model-dependent part of Equation 37.47 is estimated by a simple quadratic in θ:

$$\overline{\tfrac{1}{2}\, s_Q^2 + s'_Q} = \tfrac{1}{2}\,\theta^2\,\overline{x^6} - 3\,\theta\,\overline{x^2}, \qquad \text{which is minimized at} \qquad \hat{\theta} = \frac{3\,\overline{x^2}}{\overline{x^6}}.$$

Just two sample moments fit the parameter, without ever integrating $e^{-\theta x^4/4}$. Maximum likelihood requires $Z(\theta)$ and leads to a different estimator (Exercise 4), though for a Gaussian model the two estimators coincide (Exercise 3).

In one dimension, s'_Q is just a single extra derivative. In $\mathbb{R}^n$, however, it becomes the trace of the Jacobian, a sum of n second derivatives (one per coordinate), turning the single backward pass that computed the score into n backward passes, which is far too expensive for images. Section 38.6.1 extends score matching to $\mathbb{R}^n$, shows how adding noise to the data avoids computing these second derivatives, and derives the training loss of diffusion models from the result.

37.5.2.2 A Test for Samples

Stein's identity also works in the opposite direction: to test whether a given sample came from a model Q. Applied to the model itself, the identity states that for every valid test function g, the quantity $g' + g\, s_Q$ has mean zero under Q. The linear operator mapping g to $g' + g\, s_Q$ is called the *Stein operator* of Q. If the sample average of $g' + g\, s_Q$ is far from zero for some g, we have evidence that the sample was not drawn from Q. Notice what this check needs: only the model's score evaluated at the sample points, with no normalizing constant Z and no samples from the model.

How much can a single test function catch? Let the model be $Q = \mathcal{N}(0, 1)$, whose score is $-x$, so the Stein operator maps g to $g'(x) - x\, g(x)$. Choosing $g(x) = 1$ yields $-x$, which checks the sample mean; choosing $g(x) = x$ yields $1 - x^2$, which checks the second moment against 1. As a third choice without a simple moment interpretation, let us try $g(x) = \sin x$. Below we draw 200,000 points from three sources: the true model $\mathcal{N}(0, 1)$, a shifted distribution $\mathcal{N}(0.5, 1)$, and a widened distribution $\mathcal{N}(0, 4)$, and print each test average alongside its standard error. From the first two moments alone, we expect -0.5 and $1 - 1.25 = -0.25$ on the shifted sample, and 0 and $1 - 4 = -3$ on the widened one.

```
rng = np.random.default_rng()
n = 200_000
# Stein operator of the model Q = N(0, 1), whose score is -x:  g'(x) - x g(x)
tests = [('g = 1', np.ones_like, np.zeros_like),
         ('g = x', lambda t: t, np.ones_like),
         ('g = sin x', np.sin, np.cos)]
samples = [('N(0, 1)', rng.normal(0.0, 1.0, n)),
           ('N(0.5, 1)', rng.normal(0.5, 1.0, n)),
           ('N(0, 4)', rng.normal(0.0, 2.0, n))]
```

```
    print(f'{"sample":10s}' + ''.join(f'{name:>18s}' for name, _, _ in tests))
    for s_name, x in samples:
        vals = [g_prime(x) - x * g(x) for _, g, g_prime in tests]
        print(f'{s_name:10s}' + ''.join(f'{v.mean():+10.3f} ± {v.std() / np.sqrt(n):.3f}'
                                        for v in vals))
```

```
sample                 g = 1              g = x          g = sin x
N(0, 1)       -0.001 ± 0.002    -0.002 ± 0.003    -0.001 ± 0.002
N(0.5, 1)     -0.503 ± 0.002    -0.249 ± 0.004    -0.145 ± 0.002
N(0, 4)       +0.002 ± 0.004    -3.006 ± 0.013    -0.410 ± 0.003
```

On the model's own sample, every average lands within a couple of standard errors of zero. All three test functions readily catch the shifted sample, reporting about -0.50, -0.25, and -0.15. The widened sample tells a different story: because its mean is still zero, $g = 1$ reports a value near zero (within a hundredth or so), completely missing a fourfold error in variance. Only $g(x) = x$ (near -3) and $g(x) = \sin x$ (near -0.4) flag the mismatch. Any individual test function probes only one aspect of the sample, and a wrong distribution can easily agree with the model on that aspect.

To see why $g = 1$ catches the shift but misses the variance, note that any smooth data distribution P satisfies its own Stein identity, $\mathbb{E}_P[g'] = -\mathbb{E}_P[g\, s_P]$. Substituting this into the model's Stein expectation gives

$$\mathbb{E}_P[g' + g\, s_Q] = \mathbb{E}_P[(s_Q - s_P)\, g]. \tag{37.48}$$

Thus the test average is simply the difference between the two scores, weighted by g. For the shifted sample, that score gap is the constant $-x + (x - 0.5) = -0.5$, which $g = 1$ captures completely. For the widened sample, the score gap is $-x + x/4 = -\frac{3}{4}x$, an odd function whose integral against any symmetric density is zero when weighted by $g = 1$. No single test function can be correlated with every possible score difference.

37.5.3 The Kernel Stein Discrepancy

Since one test function checks only one property, we do what Section 37.4.1 did for the maximum mean discrepancy (MMD): we test over an entire family of functions at once, namely all functions in the unit ball of a reproducing kernel feature space, and take the largest violation. Remarkably, this supremum over infinitely many test functions once again reduces to a closed form built from kernel evaluations on pairs of data points.

Let $\mathcal{A}_Q$ denote the Stein operator of Q, defined by $(\mathcal{A}_Q g)(x) = g'(x) + s_Q(x)\, g(x)$. Stein's identity states that $\mathbb{E}_Q[\mathcal{A}_Q g] = 0$, so we measure how badly P violates the model's identity by $\sup_g \mathbb{E}_P[\mathcal{A}_Q g]$.

Consider test functions of the form $g(x) = \langle \mathbf{w}, \varphi(x) \rangle$ with feature map φ and $\|\mathbf{w}\| \le 1$. By linearity, the Stein operator acts directly on the features: $(\mathcal{A}_Q g)(x) = \langle \mathbf{w}, \psi(x) \rangle$, where $\psi = \varphi' + s_Q\, \varphi$. By the Cauchy–Schwarz inequality, the supremum of $\langle \mathbf{w}, \mathbb{E}_P[\psi] \rangle$ over the unit ball is simply $\big\|\mathbb{E}_P[\psi]\big\|$. Squaring this norm gives $\mathbb{E}\langle \psi(x), \psi(x') \rangle$ for two independent draws $x, x' \sim P$. Expanding the inner product produces four terms, each expressed in terms of the kernel $k(x, x') = \langle \varphi(x), \varphi(x') \rangle$ and its partial derivatives:

$$\langle \varphi'(x), \varphi(x') \rangle = \partial_x k, \qquad \langle \varphi(x), \varphi'(x') \rangle = \partial_{x'} k,$$
$$\langle \varphi'(x), \varphi'(x') \rangle = \partial_x \partial_{x'} k.$$

Collecting these four terms with the score weights defines the **Stein kernel** of Q:

$$\begin{aligned} k_Q(x, x') = s_Q(x)\, s_Q(x')\, k(x, x') + s_Q(x)\, \partial_{x'} k(x, x') \\ + \partial_x k(x, x')\, s_Q(x') + \partial_x \partial_{x'} k(x, x'). \end{aligned} \tag{37.49}$$

The squared **kernel Stein discrepancy** (KSD) is then the expectation of the Stein kernel over two independent draws from the data:

$$\mathrm{KSD}^2(P, Q) = \mathbb{E}_{x, x' \sim P}\big[k_Q(x, x')\big]. \tag{37.50}$$

Because this expectation depends only on P, a sample $x_1, \dots, x_N$ from P yields the unbiased estimator

$$\widehat{\mathrm{KSD}}^2 = \frac{1}{N(N-1)} \sum_{i \neq j} k_Q(x_i, x_j),$$

the same off-diagonal pairwise average used for MMD. In statistics, an average of this form over distinct tuples is called a U-statistic, hence the function name `ksd2_ustat` below.

To understand what KSD actually measures, apply Equation 37.48 to each coordinate of φ. This yields $\mathbb{E}_P[\psi] = \mathbb{E}_P[(s_Q - s_P)\, \varphi]$, and squaring the norm gives

$$\mathrm{KSD}^2(P, Q) = \mathbb{E}_{x, x' \sim P}\big[(s_Q - s_P)(x)\, k(x, x')\, (s_Q - s_P)(x')\big]. \tag{37.51}$$

Thus KSD is a kernel-smoothed cousin of the Fisher divergence $D_{\mathrm{F}}(P \| Q)$: instead of squaring the score gap at each point individually, it multiplies the score gap across pairs of data points weighted by $k(x, x')$. Unfortunately, this means KSD inherits the mode-weight blind spot, and makes it even worse. For the separated mixtures of Section 37.5.1, *both* points in a pair must now land in the empty valley where the scores differ, so KSD^2 decays roughly like e^{-m^2}, the square of the Fisher divergence rate (Exercise 2e).

When both scores are known, Equation 37.51 lets us predict the exact population value. Let the data be $P = \mathcal{N}(0, 1)$, and test it against three candidate models. Against the true model $\mathcal{N}(0, 1)$, the score gap vanishes and $\mathrm{KSD}^2 = 0$. Against the shifted model $\mathcal{N}(1, 1)$, the score gap is $-(x-1) - (-x) = 1$ everywhere, so $\mathrm{KSD}^2 = \mathbb{E}[k(X, X')]$. For the RBF kernel $k(x, x') = e^{-(x-x')^2/(2\ell^2)}$ with length scale ℓ, this is the same Gaussian integral we met in Section 37.4.1: since $X - X' \sim \mathcal{N}(0, 2)$, the expectation equals $(1 + 2/\ell^2)^{-1/2}$, which at $\ell = 1$ gives $\mathrm{KSD}^2 = 1/\sqrt{3} \approx 0.577$.

Our third candidate is the unnormalized quartic $\tilde{q}(x) = e^{-x^4/4}$ from Section 37.5.1, whose normalizing constant we never need. Its score is $-x^3$, making the score gap $x - x^3$; the resulting double integral in Equation 37.51 has no elementary closed form, so `ksd2_population` below evaluates it by quadrature on a grid (which we can do here only because we know the true data density for our synthetic benchmark).

For the one-dimensional RBF kernel, the derivatives in Equation 37.49 are $\partial_{x'} k = \frac{x - x'}{\ell^2} k$, $\partial_x k = -\frac{x - x'}{\ell^2} k$, and $\partial_x \partial_{x'} k = \left(\frac{1}{\ell^2} - \frac{(x - x')^2}{\ell^4}\right) k$. In the code below, `dk_dy` is the derivative with respect to the second argument and `kappa` stores the matrix of values $k_Q(x_i, x_j)$. We test a single sample of $2{,}000$ points from $\mathcal{N}(0, 1)$ against all three models, passing `ksd2_ustat` nothing more than each model's score function.

```
def ksd2_ustat(x, score, ell=1.0):
    """U-statistic KSD^2 with the RBF kernel exp(-(a-b)^2 / (2 ell^2))."""
    d = x[:, None] - x[None, :]
    k = np.exp(-d ** 2 / (2 * ell ** 2))
    dk_dx = -d / ell ** 2 * k
    dk_dy = d / ell ** 2 * k
    d2k = (1 / ell ** 2 - d ** 2 / ell ** 4) * k
    s = score(x)
    kappa = (s[:, None] * s[None, :] * k + s[:, None] * dk_dy
             + dk_dx * s[None, :] + d2k)
    n = len(x)
    return (kappa.sum() - np.trace(kappa)) / (n * (n - 1))

def ksd2_population(score, ell=1.0):
    """KSD^2 of the data P = N(0, 1) against a model with this score, from
    the score difference s_Q - s_P weighted by the kernel, by quadrature."""
    t = np.linspace(-10.0, 10.0, 2001)
    h = np.exp(-t ** 2 / 2) / np.sqrt(2 * np.pi) * (score(t) + t)
    k = np.exp(-(t[:, None] - t[None, :]) ** 2 / (2 * ell ** 2))
    return h @ k @ h * (t[1] - t[0]) ** 2

x_p = rng.standard_normal(2000)          # the data sample, P = N(0, 1)
for name, s in [('true model  N(0,1)', lambda t: -t),
                ('wrong model N(1,1)', lambda t: -(t - 1)),
                ('quartic exp(-x^4/4)', lambda t: -t ** 3)]:
    print(f'KSD^2 vs the {name:19s}: {ksd2_ustat(x_p, s):+.4f}  '
          f'(population {ksd2_population(s):.4f})')
```

```
KSD^2 vs the true model  N(0,1) : +0.0010  (population 0.0000)
KSD^2 vs the wrong model N(1,1) : +0.5274  (population 0.5774)
KSD^2 vs the quartic exp(-x^4/4): +0.2835  (population 0.3208)
```

Against the true model, the estimate falls within a few thousandths of zero (and may even be slightly negative, as any unbiased estimator of zero can be). Against the shifted model, it lands close to our predicted $1/\sqrt{3} \approx 0.58$, typically within about 0.1. Against the quartic, it is several tenths, clearly far from zero. Here the population value is 0.32, though a single sample can easily miss it by 0.1 or more because the cubic score $-x^3$ grows rapidly in the tails, letting a handful of extreme points dominate the average.

Notice that across all three tests, only the score function changed. The data sample was fixed, we never sampled from any model, and the quartic's normalizing constant $Z \approx 2.56$ never appeared in the code.

With the RBF kernel, a population value of zero is trustworthy: for smooth data densities under mild regularity conditions on the scores, $\mathrm{KSD}^2(P, Q) = 0$ if and only if $P = Q$ (Q. Liu et al. 2016; Chwialkowski et al. 2016). Still, our two-mixture example is a useful caution: "nonzero in theory" does not prevent the value from being astronomically small

for distributions that are far apart.

Extending KSD to $\mathbb{R}^n$ changes only the notation: g becomes a vector field, the Stein operator becomes $s_Q^\top g + \nabla \cdot g$, the scalar products in Equation 37.49 become vector inner products, and the last term becomes $\sum_i \partial_{x_i} \partial_{x_i'} k$. In practice, KSD serves as a goodness-of-fit test for unnormalized models and as a diagnostic for MCMC samplers. Applying the Stein operator to the kernel itself also gives a deterministic update direction that nudges particles toward Q, an algorithm known as Stein variational gradient descent (Liu and Wang 2016). None of the training objectives in this book use KSD directly.

37.5.4 Training Objectives and Their Divergences

Across the previous two sections we built three major families of distribution comparisons, and this section added two score-based comparisons. Figure 37.12 maps out the three families and the divergences they share. It omits the Fisher divergence, which lies outside all three families, as well as KSD: although KSD has the form of an integral probability metric (a supremum over test functions), its test class depends on the model's score and therefore changes with Q.

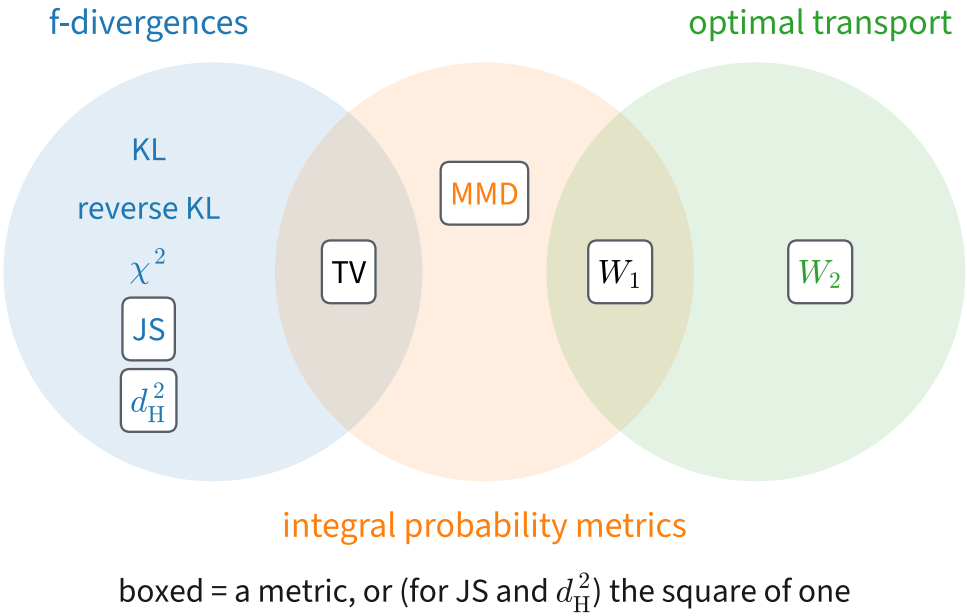


Figure 37.12: The ratio, test-function, and transport families and their shared members. f-divergences average a convex function of the density ratio, integral probability metrics maximize mean differences over a class of test functions, and optimal-transport distances measure the minimum cost of moving mass. Total variation belongs to the first two families and Wasserstein-1 to the last two. (Total variation is also a transport cost that charges 1 for moving mass by any nonzero distance, omitted from the diagram for clarity.) Boxed names are metrics; JS and the squared Hellinger divergence d_{H}^2 are squares of metrics.

Table 37.4 organizes the training objectives built on these comparisons. For each objective,

the table lists what it requires from the model and the data, along with the divergence it minimizes in the ideal limit of exact expectations and, for adversarial objectives, an optimal critic (the auxiliary network trained to tell real and generated samples apart in Section 37.3.4). The fourth column revisits the diagnostic test case that opened Section 37.3: data concentrated at a single point $P = \delta_0$ and model samples at $Q = \delta_d$ a distance d away. Any comparison that stays constant as d varies leaves such a model with no gradient pointing toward the data.

Table 37.4: Training objectives, ordered by what they need, and the divergences they minimize when averages are exact and the critic is optimal. The column on point masses is the value for δ_0 against δ_d with $d \neq 0$; $f(0)$ and $f'(\infty)$ are the limits of $f(u)$ at 0 and of $f(u)/u$ at infinity, and the MMD entry uses the RBF kernel with length scale ℓ.

What the objective needs	Objective	Divergence	Two point masses d apart	Treated in
model density at data points	maximum likelihood (autoregressive models, flows)	forward KL	∞	Section 37.3.2
samples of both, through a critic	original GAN	Jensen–Shannon	$\log 2$, constant	Section 37.3.4
samples of both, through a critic	f-GAN	the chosen D_f	$f(0) + f'(\infty)$, constant	Section 37.3.4
samples of both, no critic for a fixed kernel	moment matching, two-sample tests	MMD	$\mathrm{MMD}^2 = 2 - 2e^{-d^2/(2\ell^2)}$, flat beyond a few ℓ	Section 37.4.1
samples of both, a 1-Lipschitz critic	WGAN	W_1	d	Section 37.4.2
model samples, the model's density at them, the unnormalized target	variational inference, VAE posterior	reverse KL	∞	Section 37.3.2
model score, data samples	score matching, diffusion models	Fisher	undefined (a point mass has no score)	Section 37.5.1
model score, data samples	goodness-of-fit tests, sampler quality	KSD	undefined (a point mass has no score)	Section 37.5.3

Read from top to bottom, the table serves as a practical decision guide because its rows are ordered by what a model can provide. Whenever you can evaluate the model's normalized density, and it stays positive wherever data may appear, train by maximum likelihood: forward KL needs only the data, gives an unbiased sample estimate, and penalizes leaving any data mode uncovered. When a model can only generate samples, you have three options. Training a learned critic as in a GAN (Section 16.1) is the standard choice for sample-only models of complex data like images, where we do not know ahead of time which features distinguish real samples from generated ones. Today these are mostly one-step generators distilled from diffusion models and the decoders of latent autoencoders. The drawbacks of a learned critic are that the estimated divergence is biased downward and flattens out when the model is far from the data (Section 37.3.4). The optimal 1-Lipschitz critic has no such

flat stretch, because W_1 keeps a constant slope at every distance. Trained critics need not behave like their optimal versions, though: in the experiment of Section 16.2.4.2 a trained logistic critic still slopes at far generated samples, and a W_1 critic held near unit slope by the usual two-sided gradient penalty carries its generator to the data and then past it. A fixed kernel avoids training a second network altogether and achieves $O(N^{-1/2})$ estimation error in any dimension (Section 37.4.1), making it ideal for two-sample tests and sample evaluation. It is usually applied to learned features, because the MMD of a fixed kernel weakens as the dimension of raw data grows, and its gradient vanishes beyond a few length scales.

The final three rows cover settings where a normalizing constant is unknown. When the *target* density is unnormalized while the *model* provides both samples and a tractable density (as in variational inference), reverse KL applies, at the risk of collapsing onto a single mode. When the model itself is known only through its score, the Fisher divergence (via score matching) is the natural choice, provided we add noise across scales so it does not lose sight of distant mode weights, just as diffusion models do (Section 17.1).

This table tells us *which* comparison each objective targets, but not how reliably we can estimate it from finite data. Every sample-based estimator from the last three sections (the critic bound, MMD^2, and KSD^2) carries estimation error, and the critic bound is biased low to boot. The next two sections shift from comparing two distributions to measuring dependence between two random variables: Section 37.6 uses KL divergence to define how much one variable tells us about another, and Section 37.7 asks how much dependence a sample of n pairs can actually certify.

37.5.5 Summary

Even when a model's normalizing constant is intractable, we can compare it to data through its score, the gradient of the log-density with respect to the input, because the constant disappears upon differentiation. The Fisher divergence averages the squared gap between the data and model scores, and Stein's identity eliminates the unknown data score via a single integration by parts, yielding score matching. Because scores are purely local, however, they cannot see how probability mass is split between distant modes: for two mixtures separated by ten standard deviations, the Fisher divergence drops to 4×10^{-6} while the KL divergence remains 0.339 nats. Applying Stein's identity to the model instead gives the kernel Stein discrepancy, which tests samples using only the model's score. Ultimately, the right training objective is determined by what your model can supply: a normalized density, samples, or a score.

37.5.6 Exercises

1. **Fisher divergence and Stein moments.** Recall that the score of $\mathcal{N}(\mu, \sigma^2)$ is $-(x - \mu)/\sigma^2$.
   1. Compute the Fisher divergence Equation 37.44 between $\mathcal{N}(\mu_1, \sigma_1^2)$ and $\mathcal{N}(\mu_2, \sigma_2^2)$ in closed form, and verify that it reduces to $(\mu_1 - \mu_2)^2/(2\sigma^4)$ when the variances are equal. Is the Fisher divergence symmetric?
   2. Apply Stein's identity Equation 37.46 for $\mathcal{N}(0, 1)$ to $g(x) = x^k$ to derive the recursion $\mathbb{E}[X^{k+1}] = k\,\mathbb{E}[X^{k-1}]$, and use it to compute $\mathbb{E}[X^4]$ and $\mathbb{E}[X^6]$. What does $g(x) = 1$ yield for any density satisfying the conditions of the identity?
   3. Suppose a distribution P on $\mathbb{R}$ has finite moments of all orders and satisfies $\mathbb{E}_P[g'(X) - Xg(X)] = 0$ for every polynomial g. Using the fact that the normal distribution is uniquely determined by its moments, explain why $P = \mathcal{N}(0, 1)$.

Now let $P(X=1)=P(X=-1)=\frac{1}{2}$, and show that it satisfies the identity for every polynomial g of degree at most 2, yet fails for $g(x)=x^3$. What does this imply for a goodness-of-fit test that checks Stein's identity only on polynomials up to a fixed degree?

2. code **Score blindness to mixing weights.** ● Let $P_w = w\,\mathcal{N}(-m,1) + (1-w)\,\mathcal{N}(m,1)$.
   1. The text computes $D_{\mathrm{F}}(P_{0.7}\|P_{0.3})$ and $D_{\mathrm{KL}}(P_{0.7}\|P_{0.3})$ for $m\in\{1,3,5\}$. Extend the computation to $m\in\{1,\dots,6\}$. Show that the KL divergence approaches $0.4\log\frac{7}{3}\approx 0.339$, and determine the decay rate of the Fisher divergence by fitting $\log D_{\mathrm{F}}$ against m^2.
   2. Derive the score formula $s_w(x)=-x+m\tanh(mx-c_w)$ from the component responsibilities. Show that $s_{0.7}$ and $s_{0.3}$ differ most at $x=0$, by $0.8\,m$, and substitute $u=mx$ into the Fisher divergence integral to show that it scales as $C\,m\,e^{-m^2/2}$ for large m, where C depends only on the weights. Compare this with your fitted rate in (a). What does this imply for the mixing weights learned by score matching (Wenliang and Kanagawa 2020)?
   3. Add independent $\mathcal{N}(0,\sigma^2)$ noise to samples of both mixtures, turning each component into $\mathcal{N}(\pm m, 1+\sigma^2)$. Repeat (a) at $m=5$ for $\sigma\in\{0,0.5,1,2\}$. How does adding noise affect each divergence? Heavy noise reveals the mixing weights but blurs each mode's shape, whereas light noise does the reverse. Why can a model that matches scores across multiple noise levels recover both (Song and Ermon 2019)?
   4. Let P_t and Q_t denote two distributions after adding independent $\mathcal{N}(0,t)$ noise to both. For two Gaussians of equal variance σ^2 (raised to σ^2+t by the noise), verify from the closed forms in the text that $\frac{d}{dt}D_{\mathrm{KL}}(P_t\|Q_t) = -D_{\mathrm{F}}(P_t\|Q_t)$. This identity holds for every pair of smooth densities. Confirm it using a finite difference for the two mixtures at $m=5$ and $t\in\{1,4,9\}$, and conclude that $D_{\mathrm{KL}}(P\|Q)=\int_0^\infty D_{\mathrm{F}}(P_t\|Q_t)\,dt$. At which noise level is the Fisher divergence largest? What does this integral tell us about matching scores at a single noise level versus across all noise levels (Y. Song, Durkan, et al. 2021)?
   5. The kernel Stein discrepancy shares the same blind spot. Evaluate Equation 37.51 for the two mixtures by quadrature on a grid, using the RBF kernel with $\ell=1$ and $m\in\{1,3,5\}$, and compare with the Fisher divergences in the text (at $m=5$ you should find roughly 10^{-11}). Adapt `ksd2_population` from the text, replacing $\mathcal{N}(0,1)$ with $P_{0.7}$ as the data distribution. Where must both points of a pair fall to contribute, and how wide is that region relative to ℓ? Conclude that KSD^2 decays like e^{-m^2}. Would choosing a smaller length scale help?
3. **Score matching for a Gaussian model.** Let the data distribution P have mean a and variance v, take the model $Q=\mathcal{N}(\mu,\sigma^2)$, and write $J=\mathbb{E}_P[\frac{1}{2}s_Q^2+s_Q']$ for the model-dependent part of Equation 37.47.
   1. Show that $J=(v+(a-\mu)^2)/(2\sigma^4)-1/\sigma^2$, and verify that it is minimized at $\mu=a$ and $\sigma^2=v$, regardless of the rest of P. Compare this with the Gaussian that minimizes forward KL divergence (Section 37.3.2).
   2. At the minimum, $J=-1/(2v)$, so the Fisher divergence from P to its best Gaussian approximation is $\frac{1}{2}(\mathbb{E}_P[s_P^2]-1/v)$. Apply Stein's identity for P to $g(x)=x-a$, followed by the Cauchy–Schwarz inequality, to prove that $\mathbb{E}_P[s_P^2]\geq 1/v$, with equality if and only if P is Gaussian. Which density of a given variance has the smallest mean squared score?

3. Reparameterize the model as $\tilde{q}_\theta(x) = \exp(\theta_1 x + \theta_2 x^2)$. Show that J is a quadratic function of θ, so score matching reduces to solving two linear equations in the data moments. Show that the same holds for any exponential-family model $\tilde{q}_\theta(x) = \exp\left(\theta^\top \phi(x)\right)$. What would maximum likelihood need to compute instead?

4. code **An unnormalized quartic.** Let $\tilde{q}_\theta(x) = e^{-\theta x^4/4}$ with $\theta > 0$.
   1. Compute the score. Apply Stein's identity with $g(x) = x$ and with $g(x) = x^3$ to show that under the model $\mathbb{E}[X^4] = 1/\theta$ and $\mathbb{E}[X^6] = (3/\theta)\,\mathbb{E}[X^2]$, without computing the normalizing constant. Can Stein's identity with a polynomial g also determine $\mathbb{E}[X^2]$?
   2. The text derives the score-matching estimator $\widehat{\theta} = 3\,\overline{x^2}/\overline{x^6}$, where a bar denotes the sample average. Show that the normalizing constant scales as $Z(\theta) = \theta^{-1/4} Z(1)$ and that maximum likelihood yields $\widehat{\theta} = 1/\overline{x^4}$. Use part (a) to argue that both estimators converge to the true θ on a large sample from the model, even though they are different functions of the data.
   3. If U follows a Gamma distribution with shape $\frac{1}{4}$ and scale 1, then $(4U/\theta)^{1/4}$ with a random sign has density $\tilde{q}_\theta / Z(\theta)$. Draw $1{,}000$ points at $\theta = 1$, compute both estimates, and repeat $2{,}000$ times. Which estimator has the smaller standard deviation, and by what factor? What does score matching trade away in exchange for avoiding Z?
   4. Now suppose the data are standard normal (as in the KSD experiment), so no quartic model is exact. On a large sample, the two estimators in (b) converge to $3\,\mathbb{E}[X^2]/\mathbb{E}[X^6] = \frac{1}{5}$ and $1/\mathbb{E}[X^4] = \frac{1}{3}$. Evaluate `ksd2_population` at the scores of both fits, and find the θ that minimizes it. Three criteria have selected three different quartics. Why is this not a contradiction, and under what condition would all three agree?

5. code **The length scale of the Stein kernel.** Let the data be $P = \mathcal{N}(0, 1)$, as in the KSD experiment, and use the RBF kernel with length scale ℓ.
   1. For the model $\mathcal{N}(\mu, 1)$, use Equation 37.51 to show that $\mathrm{KSD}^2 = \mu^2\,(1 + 2/\ell^2)^{-1/2}$. What is its limit as $\ell \to \infty$? Which single test function g produces that same limit as the square of $\mathbb{E}_P[g' + g\,s_Q]$?
   2. For the model $\mathcal{N}(0, 4)$, whose score is $-x/4$, show that $\mathrm{KSD}^2 = \frac{9}{16}\,\ell\,(\ell^2+2)^{-3/2}$. (Hint: rewrite XX' in terms of the independent variables $X + X'$ and $X - X'$.) Show that this value is maximized at $\ell = 1$, the standard deviation of the data, and approaches zero as $\ell \to \infty$. Explain this limit using your answer to (a).
   3. Check both formulas with `ksd2_population` at $\ell \in \{0.25, 1, 4, 16\}$, and compute the corresponding four values for the quartic model with score $-x^3$. Why can no single length scale be optimal against every wrong model?

37.6 Mutual Information

The previous sections compared two distributions over the same variable, such as a data distribution and a model. Many questions in deep learning concern a pair of variables instead: an image and its caption, two crops of the same photograph, or a sentence and its translation. How much does one variable tell us about the other?

The standard measure of dependence, the correlation coefficient, detects only linear relationships. Consider $X \sim \mathcal{N}(0, 1)$ and let $Y = X^2 + 0.3\,\epsilon$, where $\epsilon \sim \mathcal{N}(0, 1)$ is inde-

pendent noise. Because X^3 is symmetric around zero, $\mathrm{Cov}(X, Y) = \mathbb{E}[X^3] = 0$, so the correlation between X and Y is zero. Yet X nearly determines Y: once we know X, Y varies only by noise of standard deviation 0.3, whereas on its own Y spreads roughly from 0 to 9. The quantity we define in this section captures this dependence readily—in fact, it credits this pair with more shared information than a bivariate Gaussian with correlation 0.9.

What we want is a measure that equals zero if and only if two variables are independent, grows as their dependence strengthens, and remains unchanged when either variable is re-expressed in different coordinates. We already have the exact tool needed to build it: the KL divergence $D_{\mathrm{KL}}(P\|Q)$ from Section 37.1, which by Gibbs' inequality is always nonnegative and vanishes only when $P = Q$. Taking P to be the true joint distribution of the pair and Q to be the product distribution the pair would follow if its components were independent gives the *mutual information* $I(X; Y)$.

Defining mutual information is straightforward; its real value lies in what follows from it. We establish three fundamental results. First, processing a variable cannot increase its mutual information with anything else, so no layer of a neural network carries more information about the input than the layer below it. Second, a classifier cannot reach a given error rate unless its input carries a minimum amount of information about the label. Third, the test loss of any probabilistic model directly certifies a lower bound on that information. We then turn to Shannon's two foundational questions (how much information a noisy channel can carry and how little a lossy compressor must keep) and a third question that grows out of compression: what should a learned representation keep? Throughout this section we assume the underlying distributions are known; estimating mutual information from samples alone is a harder problem that we take up in Section 37.7.

As before, we work in nats, where $1 \text{ bit} = \ln 2 \approx 0.693$ nats. All logarithms are natural; we write log in general formulas and ln when evaluating numerical values. We denote the binary entropy by $H_b(p) = -p \log p - (1 - p) \log(1 - p)$.

37.6.1 Measures of Dependence

To keep every definition concrete, we evaluate each one first on a single 2×2 probability table, which will return as a running example throughout the section.

37.6.1.1 Joint and Conditional Entropy

Let X be a fair bit, and let Y be a copy of X that is flipped with probability $\delta = 0.1$. We can read X as a bit sent over a noisy line and Y as the bit received, or read Y as a binary label and X as a noisy feature that agrees with it 90% of the time. Because the table is symmetric, nothing below depends on which interpretation we choose. With rows indexing $x \in \{0, 1\}$ and columns indexing $y \in \{0, 1\}$, the joint distribution is

$$p_{X,Y} = \begin{pmatrix} 0.45 & 0.05 \\ 0.05 & 0.45 \end{pmatrix},$$

and summing rows and columns gives the marginals $p_X = p_Y = (\frac{1}{2}, \frac{1}{2})$. We call this pair the *flipped bit*. Throughout, subscripts name the variables of a distribution, and we drop them when the arguments already make the distribution clear, as in the conditional $p(y \mid x)$.

Beyond the individual entropies $H(X)$ and $H(Y)$, a pair of random variables has two natural entropies, both introduced in Section 37.1.1.3. The *joint entropy* is the entropy of the pair viewed as a single random variable—here one with four outcomes:

$$H(X,Y) = -\mathbb{E}_{(x,y)\sim P_{X,Y}}[\log p_{X,Y}(x,y)]. \tag{37.52}$$

For our table, $H(X,Y) = -2(0.45\ln 0.45) - 2(0.05\ln 0.05) = 1.018$ nats. That is larger than the 0.693 nats of a single fair bit, but smaller than the 1.386 nats of two independent fair bits.

The *conditional entropy* measures the surprise remaining in Y once X is known. It is the average entropy of the conditional rows $p(y \mid x) = p_{X,Y}(x,y)/p_X(x)$:

$$H(Y \mid X) = -\mathbb{E}_{(x,y)\sim P_{X,Y}}[\log p(y \mid x)] = \sum_x p_X(x)\, H(Y \mid X = x). \tag{37.53}$$

For the flipped bit, the two rows are $(0.9, 0.1)$ and $(0.1, 0.9)$, each with entropy $H_b(0.1) = 0.325$ nats, so $H(Y \mid X) = 0.325$ nats. Even after observing X, Y still retains 0.325 of its original 0.693 nats of surprise.

Notice that the three numbers satisfy $1.018 = 0.693 + 0.325$. This is the chain rule $H(X,Y) = H(X) + H(Y \mid X)$ from Section 37.1.1.3: the surprise in the pair equals the surprise in X plus the additional surprise in Y once X is known.

37.6.1.2 Mutual Information

Conditional entropy captures the part of the surprise in Y that X cannot explain. What we want now is the part the two variables share. If X and Y were independent, their joint distribution would be the product of their marginals, $P_X \otimes P_Y$, placing $p_X(x)\,p_Y(y)$ in every cell. The *mutual information* is the KL divergence from the actual joint distribution to that independent product:

$$I(X;Y) = D_{\mathrm{KL}}(P_{X,Y} \,\|\, P_X \otimes P_Y) = \mathbb{E}_{(x,y)\sim P_{X,Y}}\left[\log \frac{p_{X,Y}(x,y)}{p_X(x)\,p_Y(y)}\right]. \tag{37.54}$$

For the flipped bit, the product of the marginals puts 0.25 in every cell. The ratio of the joint probability to the independent product is therefore $0.45/0.25 = 1.8$ on the diagonal, where X and Y agree, and $0.05/0.25 = 0.2$ off the diagonal, where they disagree. Averaging the log-ratios under the joint distribution gives

$$I(X;Y) = 0.9\ln 1.8 + 0.1\ln 0.2 = 0.529 - 0.161 = 0.368 \text{ nats}.$$

Agreements occur 1.8 times as often as independence predicts, and disagreements one fifth as often; mutual information is simply the expected logarithm of this factor. Because a KL divergence also measures the extra code length incurred by using the wrong model (Section 37.2), $I(X;Y)$ is the number of nats per pair wasted by a code that treats X and Y as independent. We encounter this exact penalty in Section 17.9, when a network predicts two masked tokens independently.

We can also read Equation 37.54 one row of the table at a time. Dividing numerator and denominator by $p_X(x)$ turns the ratio into $p(y \mid x)/p_Y(y)$, rewriting the expectation over pairs as an average over x of the KL divergence between the conditional and marginal distributions of Y:

$$I(X;Y) = \mathbb{E}_{x\sim P_X}\Big[D_{\mathrm{KL}}\big(p(y \mid x)\,\|\,p_Y(y)\big)\Big]. \tag{37.55}$$

In this form, mutual information measures how far, on average, observing x shifts the distribution of Y away from its marginal. For the flipped bit, each conditional row, $(0.9, 0.1)$ or $(0.1, 0.9)$, lies $0.9\ln 1.8 + 0.1\ln 0.2 = 0.368$ nats from the marginal $(0.5, 0.5)$. This asymmetric perspective is especially natural when the two variables play different roles, such as an input and the output of a noisy encoder, and we will return to it in Section 37.6.4.

Proposition (properties of mutual information). *For any pair (X, Y), $I(X;Y) = I(Y;X) \geq 0$, with equality if and only if X and Y are independent. When the entropies involved are finite,*

$$I(X;Y) = H(X) + H(Y) - H(X,Y) = H(X) - H(X \mid Y) = H(Y) - H(Y \mid X).$$

Proof. Swapping X and Y leaves the ratio in Equation 37.54 unchanged, which proves symmetry. Nonnegativity and the equality condition follow immediately from Gibbs' inequality applied to $P_{X,Y}$ and $P_X \otimes P_Y$: the KL divergence is nonnegative and vanishes if and only if $p_{X,Y}(x,y) = p_X(x)\,p_Y(y)$ everywhere, which is the definition of independence. To obtain the entropy identities, split the logarithm of the ratio into three terms and take expectations:

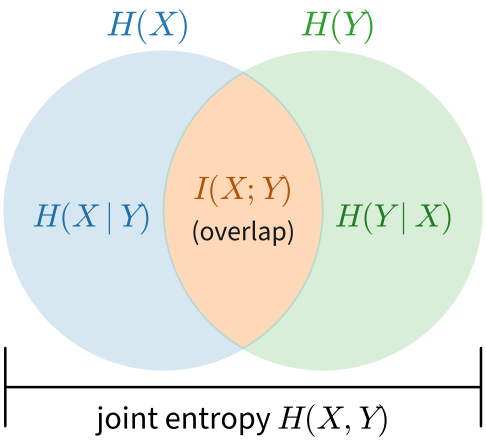
$$I(X;Y) = \mathbb{E}\left[\log p_{X,Y}(x,y)\right] - \mathbb{E}\left[\log p_X(x)\right] - \mathbb{E}\left[\log p_Y(y)\right] = -H(X,Y) + H(X) + H(Y).$$

Substituting the chain rule in its two orders, $H(X,Y) = H(Y) + H(X \mid Y) = H(X) + H(Y \mid X)$, yields the remaining two forms. ■

Thus X says exactly as much about Y as Y says about X. The final identity interprets mutual information as a reduction in uncertainty: observing X lowers the surprise in Y from $H(Y)$ to $H(Y \mid X)$—for the flipped bit, from 0.693 down to 0.325 nats. Nonnegativity also implies the subadditivity bound $H(X,Y) \leq H(X) + H(Y)$, with equality only at independence: a pair never carries more surprise than its components taken separately. Figure 37.13 illustrates these relationships visually; each identity in the proposition is simply a different way of reading off the area of the overlap.

The function `mutual_information` below implements the divergence form for any discrete joint table, and we will reuse it throughout the section. The cell also checks the identity $I(X;Y) = H(Y) - H(Y \mid X)$ by averaging the row entropies.

```
p_xy = np.array([[0.45, 0.05],    # rows x, columns y: a fair bit X and
                 [0.05, 0.45]])   # its copy Y, flipped with probability 0.1

def entropy(p):
    """Entropy of a p.m.f. of any shape, in nats (0 log 0 = 0)."""
```



$$I(X;Y) = H(X) - H(X \mid Y)$$
$$= H(Y) - H(Y \mid X)$$
$$= H(X) + H(Y) - H(X, Y)$$

Figure 37.13: Mutual information $I(X;Y)$ as the overlap of the entropies $H(X)$ and $H(Y)$: the crescents are the conditional entropies $H(X \mid Y)$ and $H(Y \mid X)$, the union is the joint entropy $H(X, Y)$, and the three equivalent expressions for $I(X;Y)$ follow by reading off the areas.

```
    p = p[p > 0]
    return float(-(p * np.log(p)).sum())

def mutual_information(p_xy):
    """I(X;Y) of a discrete joint p.m.f. (rows x, columns y), in nats."""
    p_x = p_xy.sum(axis=1, keepdims=True)
    p_y = p_xy.sum(axis=0, keepdims=True)
    m = p_xy > 0                                  # 0 log 0 = 0
    return float((p_xy[m] * np.log(p_xy[m] / (p_x * p_y)[m])).sum())

p_x, p_y = p_xy.sum(axis=1), p_xy.sum(axis=0)     # marginals from the joint
H_rows = sum(p_x[i] * entropy(p_xy[i] / p_x[i]) for i in range(2))   # H(Y|X)
I = mutual_information(p_xy)
print(f'I(X;Y) = {I:.4f} nats (KL form) = {entropy(p_y) - H_rows:.4f} '
      f'(H(Y) - H(Y|X)) = {I / np.log(2):.3f} bits')
```

```
I(X;Y) = 0.3681 nats (KL form) = 0.3681 (H(Y) - H(Y|X)) = 0.531 bits
```

Both expressions return 0.3681 nats, or 0.531 bits: a copy that flips one time in ten still conveys just over half a bit about the original.

37.6.1.3 The Gaussian Pair

So far we have worked with discrete tables, whereas most variables in deep learning (activations, gradients, and embeddings) are continuous. One continuous case can be computed completely by hand: the bivariate Gaussian. It will serve as our second running example and form the foundation for both the rate–distortion curve and the information bottleneck later in this section. For continuous densities, discrete entropies are replaced by the differential entropies h of Section 37.1.4.1, which can be negative and depend on

the units of measurement. The KL definition of mutual information suffers from neither drawback. Because the chain-rule algebra never relied on discreteness, $I(X;Y) = h(X) + h(Y) - h(X,Y) = h(Y) - h(Y \mid X)$ whenever these quantities are finite, where $h(Y \mid X)$ denotes the differential entropy of the conditional density $p(y \mid x)$, averaged over x.

Proposition (Gaussian mutual information). *If (X,Y) is bivariate Gaussian with variances $\sigma_X^2, \sigma_Y^2 > 0$ and correlation coefficient ρ, $\rho^2 < 1$, then*

$$I(X;Y) = -\tfrac{1}{2}\log\left(1-\rho^2\right). \tag{37.56}$$

Proof. We use $I(X;Y) = h(Y) - h(Y \mid X)$. A Gaussian variable with variance σ^2 has differential entropy $\frac{1}{2}\log(2\pi e\sigma^2)$ (Section 37.1.4.3), so $h(Y) = \frac{1}{2}\log(2\pi e\sigma_Y^2)$. Conditioned on $X = x$, Y remains Gaussian with variance $\sigma_Y^2(1-\rho^2)$, regardless of x. This is the variance of the residual left after subtracting the best linear prediction of Y from X, and for a bivariate Gaussian that residual is independent of X (Section 36.3.3.3). Hence $h(Y \mid X) = \frac{1}{2}\log\left(2\pi e\sigma_Y^2(1-\rho^2)\right)$, and subtracting gives $-\frac{1}{2}\log(1-\rho^2)$. ■

The result depends solely on $1-\rho^2$, the fraction of Y's variance left unexplained by X. Notice that the units of Y have cancelled out—our first glimpse of the coordinate invariance proved in Section 37.6.1.5.

When $\rho = 0$, the variables are independent and $I(X;Y) = 0$. As $\rho \to \pm 1$, the joint density collapses onto a straight line and $I(X;Y) \to \infty$. Between those extremes, $\rho = 0.5$ yields 0.144 nats, $\rho = 0.9$ yields 0.830 nats, and $\rho = 0.99$ yields 1.959 nats. The bivariate Gaussian with standard normal marginals and $\rho = 0.9$ will reappear throughout this section; we refer to it as the *Gaussian pair at $\rho = 0.9$*.

As the limit $\rho \to \pm 1$ shows, mutual information can indeed be infinite. This phenomenon will come up twice later—for deterministic layers of a neural network and for the encoder of the information bottleneck. If $Y = X$ for a continuous random variable X, knowing X pins down Y to infinitely many decimal places, and the same holds whenever Y is a deterministic function of X that takes no single value with positive probability. By contrast, infinite mutual information is impossible when Y is discrete with finite entropy: then $I(X;Y) = H(Y) - H(Y \mid X) \le H(Y)$ regardless of X, where $H(Y \mid X)$ averages the entropy of $p(\cdot \mid x)$ over x. For a standard normal X, for instance, the sign of X carries exactly $\ln 2$ nats. This same ceiling caps the Jensen–Shannon divergence of Section 16.1 at $\ln 2$: as shown there, it equals the mutual information between a sample and the binary label identifying its source.

37.6.1.4 Pointwise Mutual Information

Mutual information averages over all pairs of outcomes, which hides which specific pairs drive the dependence. Often a single pair is what we care about: does seeing the word *new* make the next word *york* much more likely than chance? The summand inside Equation 37.54 answers this directly. It is the *pointwise mutual information*,

$$\operatorname{pmi}(x,y) = \log \frac{p_{X,Y}(x,y)}{p_X(x)\,p_Y(y)}, \tag{37.57}$$

which compares how often the specific pair (x, y) occurs with how often it would occur if X and Y were independent. A positive pmi indicates that the two outcomes attract (they co-occur more often than chance predicts), a negative pmi indicates that they repel, and $I(X;Y) = \mathbb{E}_{(x,y)\sim P_{X,Y}}[\text{pmi}(x, y)]$ is their average attraction. For the flipped bit, the pmi is $\ln 1.8 = +0.588$ on the two agreeing cells and $\ln 0.2 = -1.609$ on the two disagreeing cells—the exact logarithms we averaged earlier.

Pointwise mutual information is a classic tool in natural language processing (Church and Hanks 1990), where it separates *collocations*, word pairs that naturally belong together, from words that co-occur simply because both are common. Suppose we count consecutive word pairs (bigrams) in a small corpus and tally how often each of *new*, *machine*, and *the* is followed by *york*, *learning*, or *day*:

count	york	learning	day
new	38	3	9
machine	1	46	3
the	11	21	168

Raw counts are misleading here. "The day" is by far the most frequent pair (168 occurrences versus 38 for "new york") simply because *the* and *day* are both high-frequency words. The code below divides each joint frequency by the product of its two marginal word frequencies, prints the resulting pmi for four pairs, and reports the mutual information of the full table.

```
counts = np.array([[38., 3., 9.],      # new york / new learning / new day
                   [1., 46., 3.],      # machine ...
                   [11., 21., 168.]])  # the ...
first, second = ['new', 'machine', 'the'], ['york', 'learning', 'day']

p = counts / counts.sum()
pmi = np.log(p / (p.sum(axis=1, keepdims=True) * p.sum(axis=0, keepdims=True)))
for (i, j) in [(0, 0), (1, 1), (2, 2), (2, 0)]:
    print(f'pmi({first[i]}, {second[j]}) = {pmi[i, j]:6.3f} nats '
          f'(count {counts[i, j]:.0f})')
print(f'I(first; second) = {float((p * pmi).sum()):.4f} nats')
```

```
pmi(new, york) =  1.517 nats (count 38)
pmi(machine, learning) =  1.372 nats (count 46)
pmi(the, day) =  0.336 nats (count 168)
pmi(the, york) = -1.109 nats (count 11)
I(first; second) = 0.4146 nats
```

Once we correct for the marginals, the true collocations rise to the top: "new york" scores 1.52 nats and "machine learning" scores 1.37, whereas the frequent pair "the day" reaches only 0.34. Meanwhile, "the york" is negative at -1.11 nats, occurring roughly three times less often than independence would predict. The mutual information of the table, 0.41 nats, summarizes how strongly the first word constrains the second on average. This same marginal correction reappears in learned word embeddings (Section 19.1): by training a skip-gram model to distinguish observed pairs of neighboring words from K random pairs, we implicitly factorize the matrix of shifted pointwise mutual informations, $\text{pmi}(x, y) - \log K$ (Levy and Goldberg 2014), as Exercise 7 of Section 37.7 derives.

37.6.1.5 Mutual Information and Correlation

The correlation coefficient $\rho(X,Y) = \mathrm{Cov}(X,Y)/(\sigma_X\sigma_Y)$ of Section 36.2 detects only linear dependence: having $\rho(X,Y) = 0$ does not imply independence. Mutual information, by contrast, vanishes if and only if the variables are independent. Returning to our opening example makes this difference concrete.

Let $X \sim \mathcal{N}(0,1)$ and $Y = X^2 + 0.3\,\epsilon$ with independent $\epsilon \sim \mathcal{N}(0,1)$, a pair whose correlation is $\rho = 0$. Writing $I(X;Y) = h(Y) - h(Y \mid X)$ lets us evaluate the mutual information using one-dimensional integrals. Conditioned on $X = x$, the only randomness left in Y is the additive noise, and shifting a distribution by the known constant x^2 leaves differential entropy unchanged, so $h(Y \mid X) = h(0.3\,\epsilon) = \frac{1}{2}\ln(2\pi e \cdot 0.09) = 0.215$ nats. Before computing $h(Y)$ numerically, we can bound it analytically: Y has variance $\mathrm{Var}(X^2) + 0.09 = 2.09$, and among all densities with a given variance the Gaussian maximizes differential entropy (Section 37.1.4.3), giving $h(Y) \le \frac{1}{2}\ln(2\pi e \cdot 2.09) = 1.79$ nats and thus $I(X;Y) \le 1.79 - 0.215 = 1.57$ nats. Because the marginal density of Y is a continuous mixture over x, $p_Y(y) = \int \mathcal{N}(x;0,1)\,\mathcal{N}(y;x^2,0.09)\,dx$, with no closed form, we integrate it numerically to obtain $h(Y)$.

```
normal = lambda t, var: np.exp(-t**2 / (2 * var)) / np.sqrt(2 * np.pi * var)
xs = np.linspace(-6, 6, 6001)              # quadrature grid for X
ys = np.linspace(-2.5, 30, 6501)           # grid for Y = X^2 + 0.3 eps
dx, dy = xs[1] - xs[0], ys[1] - ys[0]
p_Y = np.concatenate([                     # p_Y(y) = int N(x;0,1) N(y;x^2,0.09) dx
    (normal(xs, 1) * normal(c[:, None] - xs**2, 0.09)).sum(axis=1) * dx
    for c in np.array_split(ys, 13)])      # 13 chunks keep memory small
h_Y = float(-(p_Y[p_Y > 0] * np.log(p_Y[p_Y > 0])).sum() * dy)
h_noise = 0.5 * np.log(2 * np.pi * np.e * 0.09)   # h(Y|X) = h(0.3 eps)
print(f'h(Y) = {h_Y:.3f}, h(Y|X) = {h_noise:.3f}, '
      f'I(X;Y) = h(Y) - h(Y|X) = {h_Y - h_noise:.3f} nats')
```

```
h(Y) = 1.356, h(Y|X) = 0.215, I(X;Y) = h(Y) - h(Y|X) = 1.141 nats
```

The pair shares 1.141 nats. This sits well below our Gaussian upper bound of 1.57 nats because the density of Y—concentrated near zero with a long right tail—is far from Gaussian. At the same time, it exceeds the 0.830 nats of the Gaussian pair at $\rho = 0.9$, even though its linear correlation is zero. Figure 37.14 compares all three cases side by side.

Correlation and mutual information differ in a second fundamental way: their behavior under changes of coordinates. Measuring temperature in Fahrenheit instead of Celsius, or recording a price on a logarithmic scale, changes nothing about how informative that variable is about anything else, so a principled measure of dependence should remain invariant.

Proposition (reparameterization invariance). *Let ϕ and ψ be invertible maps of the values of X and of Y. Then $I(\phi(X);\psi(Y)) = I(X;Y)$.*

Proof. For discrete variables, invertible maps simply relabel the outcomes: the joint distribution of $(\phi(X),\psi(Y))$ assigns the same probabilities as that of (X,Y) to the relabeled outcomes, leaving the sum in Equation 37.54 unchanged. For continuous variables, we prove the smooth case here to show the mechanism; Section 37.6.2.2 removes the smoothness assumption. Suppose ϕ and ψ are continuously differentiable with continuously differentiable inverses, and change variables via $(x,y) \mapsto (\phi(x),\psi(y))$. The joint density of the

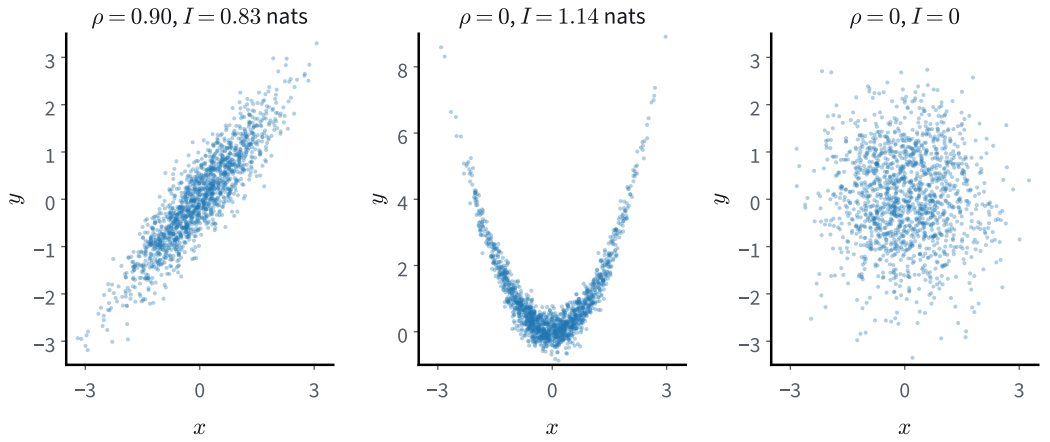


Figure 37.14: Three pairs compared by correlation and by mutual information. (a) A Gaussian pair with correlation 0.9. (b) $Y = X^2 + 0.3\epsilon$, whose correlation vanishes by symmetry even though X nearly determines Y; note the different vertical scale in (b). (c) An independent pair. Correlation cannot distinguish (b) from (c), whereas mutual information ranks (b) above (a).

transformed pair at $(\phi(x), \psi(y))$ is $p_{X,Y}(x, y)\,|\det D\phi(x)|^{-1}|\det D\psi(y)|^{-1}$, while the two marginal densities pick up the factors $|\det D\phi(x)|^{-1}$ and $|\det D\psi(y)|^{-1}$. Those Jacobian factors cancel in the density ratio, so the integrand of Equation 37.54 takes the same value at corresponding points, and its expectation is unchanged. A set of exceptional points of probability zero, such as the origin for $x \mapsto x^3$, does not affect the integral. ■

Differential entropy lacks this invariance because its Jacobian factor has nothing to cancel against, which is why h depends on units. Correlation is preserved only by affine transformations (up to sign). Applying the invertible map $x \mapsto x^3$ to the first coordinate of the Gaussian pair at $\rho = 0.9$, for example, leaves mutual information fixed at 0.830 nats, while the correlation drops from 0.9 to $3\rho/\sqrt{15} = 0.697$ (since $\mathbb{E}[X^3 Y] = 3\rho$ and $\mathrm{Var}(X^3) = 15$ for standard normal X). Mutual information asks whether two variables carry information about each other in *any* coordinates, not whether they happen to lie along a straight line.

37.6.2 Processing and Prediction

We now establish the three results promised in the introduction. The first two are fundamental limits: processing data cannot create information, and accurate prediction requires a minimum amount of it. Both follow from a single identity, the chain rule for mutual information. The third result turns the cross-entropy loss of any probabilistic model into a direct lower bound on the information carried by its input.

37.6.2.1 Conditional Mutual Information and the Chain Rule

Let us extend the flipped bit by one step. Suppose Z is produced by flipping Y once more, independently, with probability 0.2—a fair bit sent over two noisy lines in series, or a label followed by a noisy feature and a noisy copy of that feature (Figure 37.15). Two flips in series act as a single flip: the bit arrives inverted if and only if exactly one of the two lines flips it,

which happens with probability $0.1 \cdot 0.8 + 0.9 \cdot 0.2 = 0.26$. The pair (X, Z) is therefore a flipped bit with $\delta = 0.26$, giving $I(X;Z) = \ln 2 - H_b(0.26) = 0.120$ nats. The second noisy line has wiped out two thirds of the 0.368 nats that Y carried.

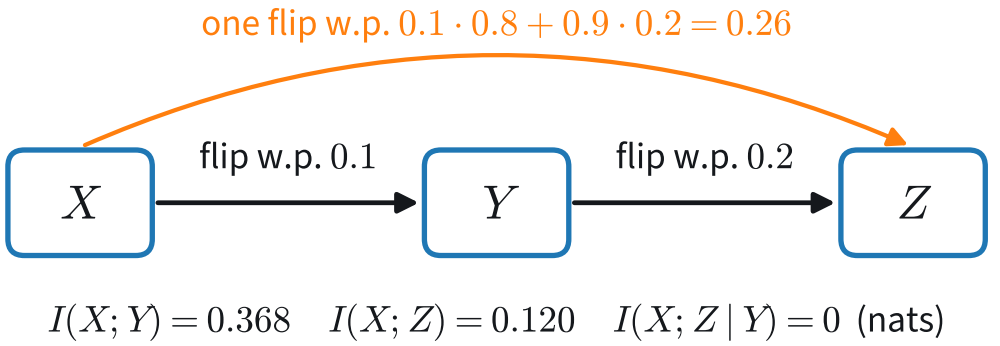


Figure 37.15: Two flips in series: a fair bit X is flipped with probability 0.1 to give Y, and Y is flipped with probability 0.2 to give Z; the two flips compose to a single flip with probability 0.26. The second flip removes two thirds of the information, and $I(X;Z \mid Y) = 0$ expresses that Z knows nothing about X that Y does not already know.

With three variables, a new question arises: how much information do X and Y share once Z is already known? Just as entropy has a conditional version, so does mutual information:

$$I(X;Y \mid Z) = H(X \mid Z) - H(X \mid Y, Z), \tag{37.58}$$

where $H(X \mid Y, Z)$ is the conditional entropy of X given the pair (Y, Z). Writing each conditional entropy as an expectation and combining the logarithms yields the divergence form $I(X;Y \mid Z) = \mathbb{E}_z\big[D_{\mathrm{KL}}(P_{X,Y|z} \,\|\, P_{X|z} \otimes P_{Y|z})\big]$, which is simply the ordinary mutual information averaged over the conditional distributions for each value of z. Because it is an expectation of KL divergences, $I(X;Y \mid Z)$ is always nonnegative, and it equals zero if and only if X and Y are conditionally independent given Z.

This definition lets us split the information that the pair (Y, Z) carries about X into two pieces. Subtracting and adding $H(X \mid Z)$ inside $I(X;Y,Z) = H(X) - H(X \mid Y, Z)$ gives

$$I(X;Y,Z) = \underbrace{H(X) - H(X \mid Z)}_{I(X;Z)} + \underbrace{H(X \mid Z) - H(X \mid Y, Z)}_{I(X;Y|Z)}.$$

This identity is the *chain rule for mutual information*, and it holds in full generality when each term is written in KL form. In words, what (Y, Z) tells us about X equals what Z tells us plus what Y adds once Z is known. We can equally well condition on Y first. For our two flips in series, the two orderings decompose the same total in two ways. Because Z is generated from Y and independent noise, once Y is known Z adds nothing further about X: $I(X;Z \mid Y) = 0$, so putting Y first gives $I(X;Y,Z) = I(X;Y) + 0 = 0.368$ nats. Putting Z first splits the same 0.368 nats as $I(X;Z) + I(X;Y \mid Z) = 0.120 + I(X;Y \mid Z)$, leaving $I(X;Y \mid Z) = 0.248$ nats. Here conditioning on Z reduced what Y reveals about

X, because Z already holds part of that information. As we will see in the next subsection, however, conditioning can also *increase* mutual information.

37.6.2.2 The Data-Processing Inequality

In a neural network, each hidden layer is computed from the layer below it, which in turn is computed from the input. Can such processing ever create information about the input that the earlier layer did not already contain? For the two noisy lines the answer was no, and the reason has nothing to do with bits in particular.

Three random variables X, Y, Z form a *Markov chain*, written $X \to Y \to Z$, when Z depends on (X, Y) solely through Y, meaning $p(z \mid x, y) = p(z \mid y)$. This captures the exact structure of sequential processing: Y is produced from X (a measurement, feature map, or hidden layer), and Z is then computed from Y alone, possibly with fresh randomness. Equivalently, X and Z are conditionally independent given Y, so $I(X; Z \mid Y) = 0$.

Proposition (data-processing inequality). *If $X \to Y \to Z$ is a Markov chain and the entropies involved are finite, then*

$$I(X; Z) \le I(X; Y),$$

with equality if and only if $I(X; Y \mid Z) = 0$, that is, if and only if $X \to Z \to Y$ is also a Markov chain (Cover and Thomas 1999).

Proof. Expand $I(X; Y, Z)$ by the chain rule in both orders:

$$I(X; Z) + I(X; Y \mid Z) = I(X; Y, Z) = I(X; Y) + I(X; Z \mid Y).$$

By the Markov property the last term vanishes, leaving $I(X; Z) = I(X; Y) - I(X; Y \mid Z) \le I(X; Y)$, since conditional mutual information is nonnegative. Equality holds if and only if $I(X; Y \mid Z) = 0$. ■

No processing of Y, whether deterministic or randomized, can increase the information it carries about X. For our two flips in series, the inequality simply reads $0.120 \le 0.368$.

Notice that this inequality relies strictly on the Markov chain structure, not on conditioning in general: outside a Markov chain, conditioning can create dependence where none existed. Suppose X and Y are independent fair bits and $Z = X \oplus Y$ is their exclusive or. Marginally X and Y share zero information, yet once Z is revealed, Y determines X completely. The code below computes the mutual informations for both the two flips in series and this XOR triple.

```
def cond_mi(p_abc):
    """I(A;B|C) of a joint p.m.f. indexed [a, b, c], in nats."""
    p_ac, p_bc, p_c = p_abc.sum(1), p_abc.sum(0), p_abc.sum((0, 1))
    m = p_abc > 0                                  # 0 log 0 = 0
    ratio = ((p_abc * p_c[None, None, :])[m]
             / (p_ac[:, None, :] * p_bc[None, :, :])[m])
    return float((p_abc[m] * np.log(ratio)).sum())

flip = lambda d: np.array([[1 - d, d], [d, 1 - d]])   # p(output | input)
p_xyz = 0.5 * flip(0.1)[:, :, None] * flip(0.2)[None, :, :]   # X -> Y -> Z
```

```
I_xy, I_xz = mutual_information(p_xyz.sum(2)), mutual_information(p_xyz.sum(1))
I_xz_y, I_xy_z = cond_mi(p_xyz.transpose(0, 2, 1)), cond_mi(p_xyz)
print(f'I(X;Y) = {I_xy:.4f}, I(X;Z) = {I_xz:.4f}, '
      f'I(X;Z|Y) = {I_xz_y:.4f}, I(X;Y|Z) = {I_xy_z:.4f}')
print(f'I(X;Y,Z) by the chain rule: {I_xz + I_xy_z:.4f} = {I_xy + I_xz_y:.4f}')
p_xor = np.zeros((2, 2, 2))
for a, b in np.ndindex(2, 2):
    p_xor[a, b, a ^ b] = 0.25          # independent fair bits, Z = X xor Y
print(f'XOR: I(X;Y) = {mutual_information(p_xor.sum(2)):.4f}, '
      f'I(X;Y|Z) = {cond_mi(p_xor):.4f} nats')
```

```
I(X;Y) = 0.3681, I(X;Z) = 0.1201, I(X;Z|Y) = 0.0000, I(X;Y|Z) = 0.2480
I(X;Y,Z) by the chain rule: 0.3681 = 0.3681
XOR: I(X;Y) = 0.0000, I(X;Y|Z) = 0.6931 nats
```

For the two flips, $I(X; Z \mid Y) = 0$ confirms the Markov property numerically, and both chain-rule decompositions of $I(X; Y, Z)$ yield 0.3681 nats. For the exclusive or, conditioning on Z boosts the mutual information between X and Y from 0 to the full $\ln 2 = 0.693$ nats of a bit.

When does processing lose no information at all? By the proposition, equality holds if and only if $X \to Z \to Y$ is also a Markov chain, meaning Z retains everything Y tells us about X. An invertible transformation $Z = g(Y)$ certainly achieves this, since Y can be recovered from Z. Yet a map can also discard part of Y without losing information about X, as long as whatever it discards is unrelated to X. For example, the total number of heads in a sequence of coin flips preserves everything the sequence says about the coin's bias, because the order of the flips carries no information about that bias. Any summary $T(Y)$ with this property (so that $p(x \mid y)$ depends on y only through $T(y)$, and therefore $I(X; T(Y)) = I(X; Y)$) is called a *sufficient statistic*. The sufficient statistics of an exponential family (Section 36.4) are precisely summaries of this kind, with Y playing the role of the data and X the parameter θ (equipped with a prior so that $I(\theta; Y)$ is well defined).

The data-processing inequality itself requires neither finite entropies nor densities. Because mutual information is a KL divergence, and no map of the data, random or deterministic, can increase a KL divergence (Section 37.3.3.3 proves this for countable sets, and it holds in full generality), Exercise 4 turns that fact into a general proof of the inequality. Applying the inequality once to an invertible map and once to its inverse also yields the reparameterization invariance of Section 37.6.1.5 without needing smoothness.

When applied to deep networks, the data-processing inequality is easy to misread. The layers of a network form a Markov chain $X \to Z_1 \to Z_2 \to \cdots$, so $I(X; Z_\ell)$ cannot increase with depth ℓ. From here to the end of this part, let Y denote a prediction label rather than the middle variable of a chain. Whatever a network knows about the label must pass through the input, so $Y \to X \to Z_1 \to Z_2 \to \cdots$ is also a Markov chain, and $I(Y; Z_\ell)$ cannot increase with ℓ either. No architecture can conjure up label information that was absent from the input, and no later layer can recover information that an earlier layer threw away.

This does *not* mean depth is unhelpful. Hidden layers do not add information about the label; rather, they reorganize existing information so that a simple final layer can read it out. Suppose the label Y is a fair bit, and the input $X \in \mathbb{R}^2$ is uniform on the circle of radius 1 when $Y = 0$ and on the circle of radius 2 when $Y = 1$. The input determines the label exactly, so $I(Y; X) = \ln 2$, yet no straight line separates two concentric circles. The

best linear classifier places its decision boundary tangent to the inner circle: it classifies the entire inner circle correctly, along with the one third of the outer circle that lies beyond the line, achieving an accuracy of $2/3$. Now apply a single nonlinear feature $Z = \|X\|^2$, which takes only the values 1 and 4. A simple threshold between them classifies every point perfectly, even though $I(Y;Z) = \ln 2$ is no larger than $I(Y;X)$. The transformation added no information, yet made all of it linearly readable. Exercise 2 formalizes a notion of usable information that counts only what a restricted family of classifiers can extract.

One final caution applies to the input chain $X \to Z_1 \to Z_2 \to \cdots$. For a deterministic layer, $H(Z_\ell \mid X) = 0$. If the representation is discrete or quantized, $I(X;Z_\ell)$ simply reduces to the entropy $H(Z_\ell)$; if it is continuous, $I(X;Z_\ell)$ is typically infinite (Section 37.6.1.3). As a measure of how much a layer retains about its input, $I(X;Z_\ell)$ is therefore meaningful only when the layer injects noise or produces a discrete output.

37.6.2.3 Fano's Inequality

Accurate prediction demands mutual information. A classifier that achieves 5% error across $1{,}000$ classes must receive an input carrying most of the label's entropy, and Fano's inequality tells us exactly how much.

Proposition (Fano's inequality). *Let Y take values in a set of k elements, let $\hat{Y} = g(X)$ be any estimate of Y computed from X and taking values in the same set, and let $P_e = P(\hat{Y} \neq Y)$ be its error probability. Then*

$$H_b(P_e) + P_e \log(k-1) \geq H(Y \mid X) = H(Y) - I(X;Y). \tag{37.59}$$

The left-hand side has a clean coding interpretation as the cost of repairing the estimate $\hat{Y}$: it takes $H_b(P_e)$ nats to specify whether $\hat{Y}$ is wrong, and whenever it is wrong (with probability P_e), at most $\log(k-1)$ nats to specify which of the remaining $k-1$ labels is correct. Anyone who has both the estimate and the repair knows Y exactly, so the repair cannot cost less than the uncertainty about Y left by $\hat{Y}$—which, by data processing, is at least the uncertainty left by X.

The proof turns this intuition into an exact bound using a single device: an indicator variable for whether an error occurred. The rest is just the chain rule.

Proof. Let $W = \mathbf{1}\{\hat{Y} \neq Y\}$ be the error indicator. We expand $H(W, Y \mid \hat{Y})$, the remaining uncertainty about both the error and the true label given the estimate, in two ways. First, W is completely determined by $(Y, \hat{Y})$ and therefore adds no entropy, so $H(W, Y \mid \hat{Y}) = H(Y \mid \hat{Y})$. Second, the chain rule gives $H(W, Y \mid \hat{Y}) = H(W \mid \hat{Y}) + H(Y \mid W, \hat{Y})$, and we bound each term separately:

- Conditioning cannot increase entropy, since $H(W) - H(W \mid \hat{Y}) = I(W;\hat{Y}) \geq 0$. Therefore $H(W \mid \hat{Y}) \leq H(W) = H_b(P_e)$.
- When $W = 0$, the label equals $\hat{Y}$ and has zero uncertainty. When $W = 1$ (which happens with probability P_e), the label must be one of the $k-1$ values other than $\hat{Y}$, so its entropy is at most $\log(k-1)$. Averaging over W gives $H(Y \mid W, \hat{Y}) \leq P_e \log(k-1)$.

Combining both bounds yields $H(Y \mid \hat{Y}) \le H_b(P_e) + P_e \log(k-1)$. Finally, because $\hat{Y}$ is computed solely from X, the triple $Y \to X \to \hat{Y}$ forms a Markov chain, and the data-processing inequality gives $I(Y;\hat{Y}) \le I(Y;X)$, or equivalently $H(Y \mid \hat{Y}) \ge H(Y \mid X)$. ■

Notice that the count of $k-1$ remaining labels in the second step relies on $\hat{Y}$ taking values inside the same label set. If the classifier is allowed to answer "abstain", then $\log k$ must replace $\log(k-1)$, and the stated inequality breaks down: for a fair bit Y independent of X and a classifier that always abstains, $P_e = 1$ and the left-hand side becomes $H_b(1) + \ln 1 = 0$, which lies below $H(Y \mid X) = \ln 2$.

In the simplest case, Fano's bound is exact. For the flipped bit with the estimate $\hat{Y} = X$, we have $P_e = 0.1$ and $k = 2$, so the left-hand side equals $H_b(0.1) + 0.1 \ln 1 = 0.325$, matching $H(Y \mid X)$ exactly.

No classifier, regardless of architecture or training procedure, can beat Fano's bound. For $1{,}000$ balanced classes, $H(Y) = \ln 1000 = 6.91$ nats; rearranging Equation 37.59 for $I(X;Y)$ shows the minimum mutual information required to achieve each target error rate.

```
H_b = lambda p: -p * np.log(p) - (1 - p) * np.log(1 - p)   # binary entropy
k = 1000
for p_e in (0.01, 0.05, 0.2, 0.5):
    need = np.log(k) - H_b(p_e) - p_e * np.log(k - 1)
    print(f'P_e = {p_e:4.2f}: I(X;Y) >= {need:.3f} of log k = {np.log(k):.3f} nats')
```

```
P_e = 0.01: I(X;Y) >= 6.783 of log k = 6.908 nats
P_e = 0.05: I(X;Y) >= 6.364 of log k = 6.908 nats
P_e = 0.20: I(X;Y) >= 5.026 of log k = 6.908 nats
P_e = 0.50: I(X;Y) >= 2.761 of log k = 6.908 nats
```

Reaching 5% error requires at least 6.36 of the 6.91 nats of label entropy (92% of the total), and even a classifier that is wrong half the time needs 2.76 nats, or two fifths of the total.

37.6.2.4 The Loss Certificate

Fano's inequality turns a classification error rate into a lower bound on mutual information. A language model, however, is not evaluated by hard error rate; like almost every model trained by maximum likelihood, it reports a cross-entropy loss instead. That loss yields a lower bound on mutual information as well, and the proof takes only three lines.

Proposition (a loss certifies information). *Let $q(y \mid x)$ be any model of the label given the input, with cross-entropy loss $L = \mathbb{E}[-\log q(Y \mid X)]$ under $P_{X,Y}$. Then*

$$I(X;Y) \;\ge\; H(Y) - L, \tag{37.60}$$

and for k balanced classes, $I(X;Y) \ge \log k - L$.

Proof. By definition, $I(X;Y) = H(Y) - H(Y \mid X)$. For each x, the cross-entropy of $q(\cdot \mid x)$ under $p(\cdot \mid x)$ exceeds the true conditional entropy of $p(\cdot \mid x)$ by $D_{\mathrm{KL}}(p(\cdot \mid x) \,\|\, q(\cdot \mid x)) \ge 0$ (Section 37.1). Averaging over x yields $H(Y \mid X) \le L$, and subtracting from $H(Y)$ completes the proof. ■

Because held-out test loss estimates L, the test loss of any classifier or language model directly certifies a lower bound on the information its input carries. For example, a 10-class classifier on balanced labels with a test loss of 0.3 nats certifies that its input carries at least $\ln 10 - 0.3 = 2.0$ of the label's 2.30 nats. The bound holds even if the model is poorly calibrated or badly trained; a weaker model simply certifies a smaller lower bound. This is the variational bound of Barber and Agakov (2003) with a classifier acting as the decoder, and Section 37.7.3 develops it into a broader family of mutual information estimators.

The certificate is tight if and only if $q(y \mid x)$ matches the true conditional distribution $p(y \mid x)$. On the flipped bit, a model that predicts $\hat{Y} = X$ with confidence c has expected loss $L = -0.9 \log c - 0.1 \log(1 - c)$, which we evaluate below across four confidence levels.

```
for c in (0.6, 0.8, 0.9, 0.99):
    L = -(0.9 * np.log(c) + 0.1 * np.log(1 - c))   # cross-entropy of q
    print(f'confidence {c:4.2f}: loss {L:.4f}, '
          f'certifies I(X;Y) >= {np.log(2) - L:.4f} nats')
```

```
confidence 0.60: loss 0.5514, certifies I(X;Y) >= 0.1418 nats
confidence 0.80: loss 0.3618, certifies I(X;Y) >= 0.3314 nats
confidence 0.90: loss 0.3251, certifies I(X;Y) >= 0.3681 nats
confidence 0.99: loss 0.4696, certifies I(X;Y) >= 0.2236 nats
```

Only the calibrated model ($c = 0.9$) certifies the full 0.368 nats. An overconfident model ($c = 0.99$) pays a steep penalty whenever it errs and certifies only 0.22 nats, while an underconfident one ($c = 0.6$) certifies 0.14 nats. Yet all four models make identical hard predictions and err 10% of the time, so Fano's inequality, which sees only the error rate, certifies the full 0.368 nats for every one of them. Cross-entropy loss sees the predicted probabilities as well, and penalizes miscalibration. Both results upper-bound the same conditional entropy $H(Y \mid X)$ (Fano via an error rate and the certificate via a loss), turning each upper bound into a lower bound on $I(X; Y)$. In fact, the two are even more closely connected than they appear: Exercise 6 shows that Fano's bound is itself the loss certificate of one specific model, and that a well-calibrated classifier that knows when it is uncertain can certify strictly more.

37.6.3 Communication and Compression

Shannon posed two foundational engineering questions whose answers turn out to be a maximum and a minimum of mutual information: how much information a noisy channel can reliably carry, and how little information a lossy compressor must keep. Readers focused on representation learning may skim channel capacity on a first pass, whereas rate–distortion theory is essential because the final part of this section builds directly on it.

37.6.3.1 Channel Capacity

How much information can survive a communication line that flips each bit independently with probability δ? A memoryless *channel* is specified by a conditional distribution $p(y \mid x)$ applied independently to each input symbol, with each transmitted symbol counting as one use of the channel. Our flipped bit is exactly such a channel, with input X and output Y. Although the channel noise is fixed, the sender is free to choose the input distribution $p(x)$. The *capacity* of the channel is the maximum mutual information achievable over all choices of input distribution, measured in nats per use:

$$C = \max_{p(x)} I(X;Y). \tag{37.61}$$

A sender who transmits mostly zeros underuses the channel, since such an input carries little entropy to begin with. Shannon's channel coding theorem gives this quantity its operational meaning. Defining the *rate* of a code as the number of nats of message it delivers per channel use, Shannon showed that by encoding long blocks of symbols one can transmit reliably—with error probability as small as desired—at any rate below C, whereas reliable communication is impossible at any rate above C (Shannon 1948; Cover and Thomas 1999). We state this theorem without proof.

For the bit-flipping channel, known as the *binary symmetric channel*, computing the capacity takes a single line. Write $I(X;Y) = H(Y) - H(Y \mid X)$. Every row of $p(y \mid x)$ is $(1-\delta, \delta)$ up to ordering, so $H(Y \mid X) = H_b(\delta)$ regardless of the input distribution. Meanwhile, $H(Y) \le \ln 2$, with equality if and only if Y is uniform. Because $P(Y=1) = \delta + P(X=1)(1-2\delta)$, whenever $\delta \neq \tfrac{1}{2}$ the output is uniform if and only if the input is uniform. Hence

$$C = \ln 2 - H_b(\delta) \quad \text{nats per use}, \tag{37.62}$$

achieved by the uniform input. At $\delta = 0.1$, this recovers the 0.368 nats of our flipped bit (whose input was already uniform); for the two lines in series ($\delta = 0.26$), capacity drops to 0.120 nats. At $\delta = \tfrac{1}{2}$, the output is independent of the input and capacity vanishes. At $\delta = 1$, capacity returns to the full $\ln 2$ nats, since a receiver who knows every bit is inverted can simply flip it back: noise destroys information only to the extent that the receiver cannot predict it. The code below first sweeps the input distribution for the $\delta = 0.1$ channel, and then sweeps the flip probability δ under a uniform input.

```
inputs = (0.1, 0.3, 0.5, 0.7, 0.9)
I_in = [mutual_information(np.array([[1 - a], [a]]) * flip(0.1)) for a in inputs]
print('P(X=1):', '  '.join(f'{a:6.2f}' for a in inputs))
print('I(X;Y):', '  '.join(f'{v:6.4f}' for v in I_in))
deltas = (0.0, 0.1, 0.26, 0.5, 1.0)
C = [np.log(2) - entropy(np.array([d, 1 - d])) for d in deltas]
print('delta: ', '  '.join(f'{d:6.2f}' for d in deltas))
print('C:     ', '  '.join(f'{c:6.4f}' for c in C))
```

```
P(X=1):   0.10    0.30    0.50    0.70    0.90
I(X;Y): 0.1463  0.3160  0.3681  0.3160  0.1463
delta:    0.00    0.10    0.26    0.50    1.00
C:      0.6931  0.3681  0.1201  0.0000  0.6931
```

An input that sends a 1 only 10% of the time has only $H_b(0.1) = 0.325$ nats to offer and gets 0.146 of them through—two fifths of what the uniform input achieves. The left panel of Figure 37.16 plots the capacity as a function of δ. Neural networks contain channels of their own: dropout erases a feature with probability α and passes it through otherwise, and Exercise 7 shows that a binary feature can push at most $(1-\alpha)\ln 2$ nats through it.

37.6.3.2 Rate–Distortion

Lossless compression requires $H(X)$ nats per symbol (Section 37.2), which means an exact lossless code for a continuous source would require infinitely many nats. In practice,

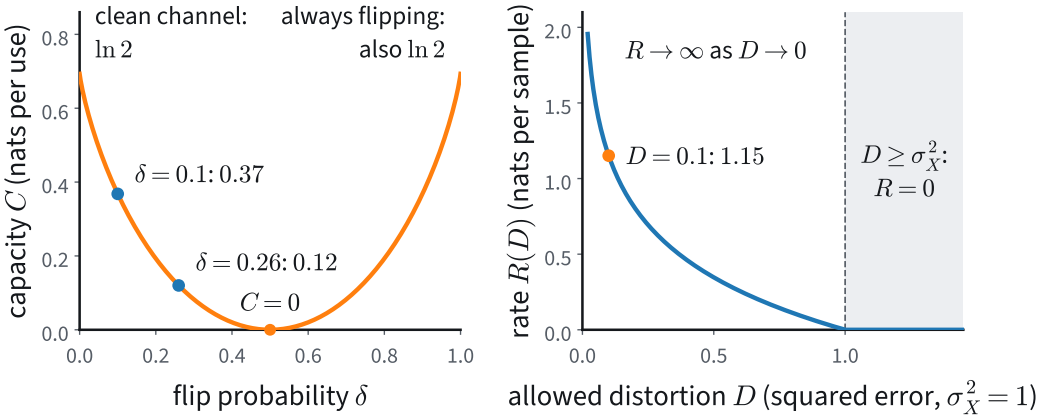


Figure 37.16: Shannon's two operational curves, in nats. Left: the capacity $C = \ln 2 - H_b(\delta)$ of the binary symmetric channel, with the flipped bit ($\delta = 0.1$) and the two flips in series ($\delta = 0.26$) marked; a channel that flips with probability $1/2$ carries nothing, while one that always flips carries the full $\ln 2$. Right: the Gaussian rate–distortion function for a source of variance $\sigma_X^2 = 1$; the rate diverges as $D \to 0$ and reaches zero once D meets the source variance.

an approximate reconstruction is usually all we need: an image codec may perturb pixels slightly, and a learned representation may deliberately discard irrelevant detail. Suppose we choose a nonnegative *distortion* measure $d(x, \hat{x}) \geq 0$ scoring how well $\hat{x}$ reconstructs x, and allow an average distortion of at most D. How few nats per sample are required? The *rate–distortion function* answers this as the minimum mutual information an encoder $p(\hat{x} \mid x)$ must retain to stay within the distortion budget:

$$R(D) = \inf_{p(\hat{x}|x):\ \mathbb{E}[d(X,\hat{X})] \leq D} I(X; \hat{X}). \tag{37.63}$$

A tighter distortion budget demands more information. Shannon's rate–distortion theorem mirrors the channel coding theorem, with the rate now measuring the nats a code spends per source sample: block codes operating at any rate above $R(D)$ can achieve average distortion D, whereas no code below $R(D)$ can (Shannon 1959).

In practice, it is usually easier to replace the hard distortion constraint with a penalty, minimizing $I(X; \hat{X}) + \lambda\, \mathbb{E}[d(X, \hat{X})]$ over all encoders for a multiplier $\lambda > 0$. Every encoder corresponds to a point in the distortion–rate plane lying on or above the curve $R(D)$, and the penalized objective is constant along straight lines of slope $-\lambda$. Because $R(D)$ is convex, the minimum occurs where a line of slope $-\lambda$ touches the curve from below: larger λ selects lower distortion at a higher rate. The information bottleneck of Section 37.6.4 is formulated in exactly this Lagrangian form.

Proposition (Gaussian rate–distortion). *For a source $X \sim \mathcal{N}(0, \sigma_X^2)$ under squared error $d(x, \hat{x}) = (x - \hat{x})^2$,*

$$R(D) = \frac{1}{2} \log \frac{\sigma_X^2}{D} \quad (0 < D < \sigma_X^2), \qquad R(D) = 0 \quad (D \geq \sigma_X^2). \tag{37.64}$$

Inverting this formula for distortion gives $D = \sigma_X^2 e^{-2R}$: every additional bit of rate ($\ln 2$ nats) cuts the squared error by a factor of four.

Proof. The lower bound is the main step, and it rests on a single fact from Section 37.1.4.3: among all densities with variance s^2, the Gaussian maximizes differential entropy at $\frac{1}{2}\log(2\pi e s^2)$, which increases with s^2. Take any encoder satisfying $\mathbb{E}[(X - \hat{X})^2] \le D$. Then

$$\begin{aligned} I(X;\hat{X}) &= h(X) - h(X \mid \hat{X}) = h(X) - h(X - \hat{X} \mid \hat{X}) \\ &\ge h(X) - h(X - \hat{X}) \\ &\ge \tfrac{1}{2}\log(2\pi e \sigma_X^2) - \tfrac{1}{2}\log(2\pi e D) = \tfrac{1}{2}\log\frac{\sigma_X^2}{D}. \end{aligned}$$

The second equality holds because subtracting the known value $\hat{X}$ merely shifts the conditional density without changing its entropy. The first inequality uses the fact that conditioning cannot increase entropy, and the second applies the Gaussian maximum-entropy bound, since the reconstruction error $X - \hat{X}$ has variance at most $\mathbb{E}[(X - \hat{X})^2] \le D$. When $D \ge \sigma_X^2$, a constant encoder that transmits nothing and always outputs the mean 0 already meets the budget, so $R(D) = 0$.

It remains to construct an encoder that achieves this bound for $D < \sigma_X^2$. Set $\sigma_X^2 = 1$ and add independent Gaussian noise: $Z = X + \sigma\epsilon$ with $\epsilon \sim \mathcal{N}(0,1)$ and $\sigma > 0$. Then $\mathrm{Cov}(X, Z) = 1$ and $\mathrm{Var}(Z) = 1 + \sigma^2$, giving squared correlation $\rho_{XZ}^2 = 1/(1+\sigma^2)$. Applying Equation 37.56 gives the mutual information retained by this encoder:

$$I(X; X + \sigma\epsilon) = \tfrac{1}{2}\log\left(1 + \sigma^{-2}\right). \tag{37.65}$$

Now reconstruct X using the conditional mean $\hat{X} = \mathbb{E}[X \mid Z] = Z/(1+\sigma^2)$, the optimal linear regression of X on Z with slope $\mathrm{Cov}(X,Z)/\mathrm{Var}(Z)$. Its mean squared error is the conditional variance $D = 1 - 1/(1+\sigma^2) = \sigma^2/(1+\sigma^2)$, which sweeps across the entire interval $(0, 1)$ as σ varies. Because $\hat{X}$ is an invertible rescaling of Z, we have $I(X;\hat{X}) = \frac{1}{2}\log(1+\sigma^{-2}) = \frac{1}{2}\log(1/D)$. For a general variance σ_X^2, simply rescale X and $\hat{X}$. ■

The encoder used in this proof—a signal corrupted by independent Gaussian noise—is known as a *Gaussian channel* with signal-to-noise ratio σ^{-2}, and Equation 37.65 is its capacity (Exercise 8). The forward noising process of a diffusion model (Chapter 17) is precisely such a channel: regardless of the data distribution, a coordinate noised to signal-to-noise ratio s carries at most $\frac{1}{2}\log(1+s)$ nats about its clean value. The code below simulates this encoder on one million samples at $\sigma^2 = 1/9$, 1, and 3 (corresponding to $D = 0.1$, 0.5, and 0.75) and verifies its squared error.

```
rng = np.random.default_rng()
x = rng.standard_normal(1_000_000)
for s2 in (1 / 9, 1.0, 3.0):                          # sigma^2
    z = x + np.sqrt(s2) * rng.standard_normal(x.size)  # encoder
    x_hat = z / (1 + s2)                               # E[X | Z]
    D = s2 / (1 + s2)
    mse = np.mean((x - x_hat)**2)
```

```
    print(f'sigma^2 = {s2:5.3f}: D = {D:.3f}, MSE {mse:.3f}, '
          f'I(X;Z) = R(D) = {0.5 * np.log(1 + 1 / s2):.3f} nats')
```

```
sigma^2 = 0.111: D = 0.100, MSE 0.100, I(X;Z) = R(D) = 1.151 nats
sigma^2 = 1.000: D = 0.500, MSE 0.500, I(X;Z) = R(D) = 0.347 nats
sigma^2 = 3.000: D = 0.750, MSE 0.750, I(X;Z) = R(D) = 0.144 nats
```

The empirical squared errors match D to within a few thousandths: a distortion of $D = 0.1$ costs 1.15 nats per sample, $D = 0.5$ costs 0.35 nats (half a bit), and $D = 0.75$ costs 0.14 nats.

Keep in mind that this noisy encoder achieves $R(D)$ as a mutual information, not yet as an actual file size. A practical code must map samples to discrete bit strings, and it approaches the rate–distortion curve only by coding long blocks of samples jointly. Any scalar code that quantizes one sample at a time stays strictly above the curve. The simplest example keeps only the sign of X: it spends $\ln 2 = 0.693$ nats and reconstructs $\pm\sqrt{2/\pi}$ (the mean of $|X|$), incurring squared error $1 - 2/\pi = 0.363$, whereas the rate–distortion curve asks only 0.506 nats for that distortion.

The right panel of Figure 37.16 shows the full curve. As $D \to 0$, the required rate climbs without bound—reflecting the infinite information in an exact copy of a continuous variable—and drops to 0 once D reaches σ_X^2. That steep climb near $D = 0$ explains why modern image generators compress images before modeling them: the finest, nearly imperceptible pixel details consume the vast majority of the bits, so Section 17.7 delegates them to an autoencoder.

37.6.4 The Information Bottleneck

A good representation should keep what predicts the label and discard the rest. At first glance, discarding anything might seem counterproductive, since the data-processing inequality tells us that throwing away information can never improve the Bayes-optimal predictor. In practice, however, irrelevant details carry a real cost. A photograph of a dog also encodes the lighting, the lawn, and where in the frame the dog happens to stand. Storing all of those nuances costs extra nats, and a classifier trained on finitely many images can easily mistake them for genuine cues to the label. The information bottleneck formalizes this trade-off using two mutual informations: it is a rate–distortion problem in which the label itself defines the distortion.

37.6.4.1 The Bottleneck Objective

Let Z be a representation of an input X generated by a stochastic encoder $p(z \mid x)$, and let Y be the target label. When Z is continuous, the encoder must be stochastic, because a deterministic map would typically yield $I(X; Z) = \infty$ (Section 37.6.1.3). The *information bottleneck* (Tishby et al. 1999) seeks a representation Z that retains as little of X as possible while preserving as much about Y as possible:

$$\min_{p(z|x)} \; I(X; Z) - \beta\, I(Y; Z). \tag{37.66}$$

Here $I(X; Z)$ is the rate—how much information the representation keeps about the input—while $I(Y; Z)$, weighted by $\beta > 0$, is the information it keeps about the label. Because the

encoder sees only X, the variables form a Markov chain $Y \to X \to Z$, and the data-processing inequality constrains $I(Y;Z) \le \min\{I(X;Y),\, I(X;Z)\}$. A representation can never know more about the label than the input does, nor can it predict well without keeping input information.

The multiplier β governs the trade-off, and its two extremes are easy to understand. Whenever $\beta \le 1$, keeping nothing is optimal: by the inequality above, $I(X;Z) - \beta I(Y;Z) \ge (1-\beta)\, I(X;Z) \ge 0$, and a constant encoder that ignores X completely achieves $I(X;Z) = I(Y;Z) = 0$. When the constant encoder is optimal, we say the bottleneck *collapses*. Conversely, as β grows large, the label term dominates and the encoder preserves whatever helps predict Y. In that limit, $I(Y;Z)$ approaches its ceiling $I(X;Y)$, and the representation approaches a *minimal sufficient statistic*: the coarsest summary of X that still predicts Y as well as X itself. Of course, when the encoder is restricted to a parametric family—as every neural network is—it may stop short of that ceiling.

Proposition (the bottleneck as a rate–distortion problem). *If $Y \to X \to Z$ is a Markov chain, then*

$$I(Y;Z) = I(X;Y) - \mathbb{E}_{(x,z)\sim P_{X,Z}}\Big[D_{\mathrm{KL}}\big(p(y \mid x) \,\|\, p(y \mid z)\big)\Big]. \tag{37.67}$$

Proof. Expand $I(Y;X,Z)$ via the chain rule in both orders: $I(Y;Z) + I(Y;X \mid Z) = I(Y;X) + I(Y;Z \mid X)$. The Markov property gives $I(Y;Z \mid X) = 0$. Writing the remaining conditional mutual information in KL form yields $I(Y;X \mid Z) = \mathbb{E}_{(x,z)}\big[D_{\mathrm{KL}}(p(y \mid x,z) \,\|\, p(y \mid z))\big]$, and the Markov property simplifies $p(y \mid x,z)$ to $p(y \mid x)$. ∎

Substituting Equation 37.67 into the objective turns Equation 37.66 into $I(X;Z) + \beta\, \mathbb{E}[d(X,Z)] - \beta I(X;Y)$, with distortion $d(x,z) = D_{\mathrm{KL}}(p(y \mid x) \,\|\, p(y \mid z))$. Since the final term $\beta I(X;Y)$ does not depend on the encoder, the information bottleneck is precisely the Lagrangian form of a rate–distortion problem. Instead of asking a designer to hand-craft a distortion metric, the label supplies it automatically: a code z is a faithful representation of x whenever it predicts Y just as accurately as x does. Unlike ordinary rate–distortion, however, this distortion depends on the encoder itself through $p(y \mid z)$, making optimization considerably harder.

37.6.4.2 The Gaussian Information Bottleneck

In one special setting, every quantity in the information bottleneck can be solved in closed form (Chechik et al. 2005). Take (X,Y) to be our Gaussian pair at $\rho = 0.9$, for which $I(X;Y) = 0.830$ nats, and use the additive-noise encoder from Section 37.6.3.2:

$$Z = X + \sigma\, \epsilon, \qquad \epsilon \sim \mathcal{N}(0,1),$$

parameterized solely by the noise scale σ. The rate $I(X;Z)$ is already given by Equation 37.65. For the label, $\mathrm{Cov}(Y,Z) = \mathrm{Cov}(Y,X) = \rho$ and $\mathrm{Var}(Z) = 1 + \sigma^2$, so $\rho_{YZ}^2 = \rho^2/(1+\sigma^2)$. Applying Equation 37.56 gives both mutual informations:

$$I(X;Z) = \tfrac{1}{2}\log\left(1+\sigma^{-2}\right), \qquad I(Y;Z) = -\tfrac{1}{2}\log\left(1 - \frac{\rho^2}{1+\sigma^2}\right). \tag{37.68}$$

As σ decreases from ∞ to 0, the representation moves smoothly from knowing nothing about X to knowing X exactly, and its label information rises from 0 to the data-processing ceiling of 0.830 nats.

To visualize this trade-off directly, write $u = I(X;Z)$ for the rate (reserving R for the rate–distortion function $R(D)$). Inverting Equation 37.68 gives $\sigma^{-2} = e^{2u} - 1$, hence $1/(1+\sigma^2) = 1 - e^{-2u}$, expressing label information directly as a function of rate:

$$I(Y;Z) = -\tfrac{1}{2}\log\left(1 - \rho^2(1 - e^{-2u})\right), \qquad \frac{d\,I(Y;Z)}{du} = \frac{\rho^2}{\rho^2 + (1-\rho^2)\,e^{2u}}.$$

The derivative decreases monotonically from ρ^2 at $u = 0$ toward 0, so the curve is strictly concave. Leaving the origin with slope $\rho^2 = 0.81$, the first nat of input information buys at most 0.81 nats of label information, and each subsequent nat yields diminishing returns as $I(Y;Z)$ approaches $I(X;Y)$. Chechik et al. (2005) proved that for a Gaussian pair no encoder, linear or nonlinear, can surpass this curve, making it the exact optimal frontier; we use that result without proof. For any chosen β, the minimizer of the Lagrangian selects a single *operating point* along this frontier.

Proposition (operating point and collapse). *In the Gaussian bottleneck with $\rho^2 < 1$, the Lagrangian $u - \beta\, I(Y;Z)$ is convex in u, with derivative $1 - \beta\rho^2$ at $u = 0$. For $\beta \le 1/\rho^2$ its minimum is $u = 0$, the constant encoder. For $\beta > 1/\rho^2$ the minimizer and its label information are*

$$u^* = \frac{1}{2}\log\frac{\rho^2(\beta-1)}{1-\rho^2}, \qquad I(Y;Z)^* = \frac{1}{2}\log\frac{\beta-1}{\beta(1-\rho^2)}. \tag{37.69}$$

Proof. Because $I(Y;Z)$ is concave in u, the Lagrangian $u - \beta I(Y;Z)$ is convex, with derivative $1 - \beta\rho^2/(\rho^2 + (1-\rho^2)e^{2u})$. At $u = 0$, this derivative equals $1 - \beta\rho^2$. Whenever $\beta\rho^2 \le 1$, the derivative is nonnegative at $u = 0$ and strictly increasing, so the minimum over $u \ge 0$ occurs at $u = 0$. Otherwise, setting the derivative to zero gives $\rho^2 + (1-\rho^2)e^{2u} = \beta\rho^2$, or $e^{2u} = \rho^2(\beta-1)/(1-\rho^2)$. Substituting $\rho^2 e^{-2u} = (1-\rho^2)/(\beta-1)$ into $I(Y;Z) = -\tfrac{1}{2}\log(1 - \rho^2 + \rho^2 e^{-2u})$ yields the second expression. ■

Earlier, the general data-processing argument guaranteed collapse for $\beta \le 1$. For the Gaussian pair at $\rho = 0.9$, collapse actually extends all the way to $\beta \le 1/\rho^2 = 1.235$. Below that threshold, every nat of rate adds 1 to the objective while reducing it by at most $\beta\rho^2 \le 1$, because the frontier's slope never exceeds its initial value ρ^2. Above the threshold, increasing β slides the operating point up the frontier toward the ceiling: $\beta = 2$ gives $u^* = \tfrac{1}{2}\ln(0.81/0.19) = 0.725$ nats of rate and 0.484 nats of label information, while $\beta = 16$ reaches 2.079 and 0.798 nats. Figure 37.17 plots this frontier and its operating points in the *information plane*, with rate $I(X;Z)$ on the horizontal axis and label information $I(Y;Z)$ on the vertical axis.

A phase transition this sharp is worth checking numerically: the proposition predicts that the optimal encoder keeps nothing at $\beta = 1.2$ yet retains information at $\beta = 1.25$. Below, we minimize the Lagrangian directly over a grid of $4{,}001$ noise scales without invoking the closed-form solution.

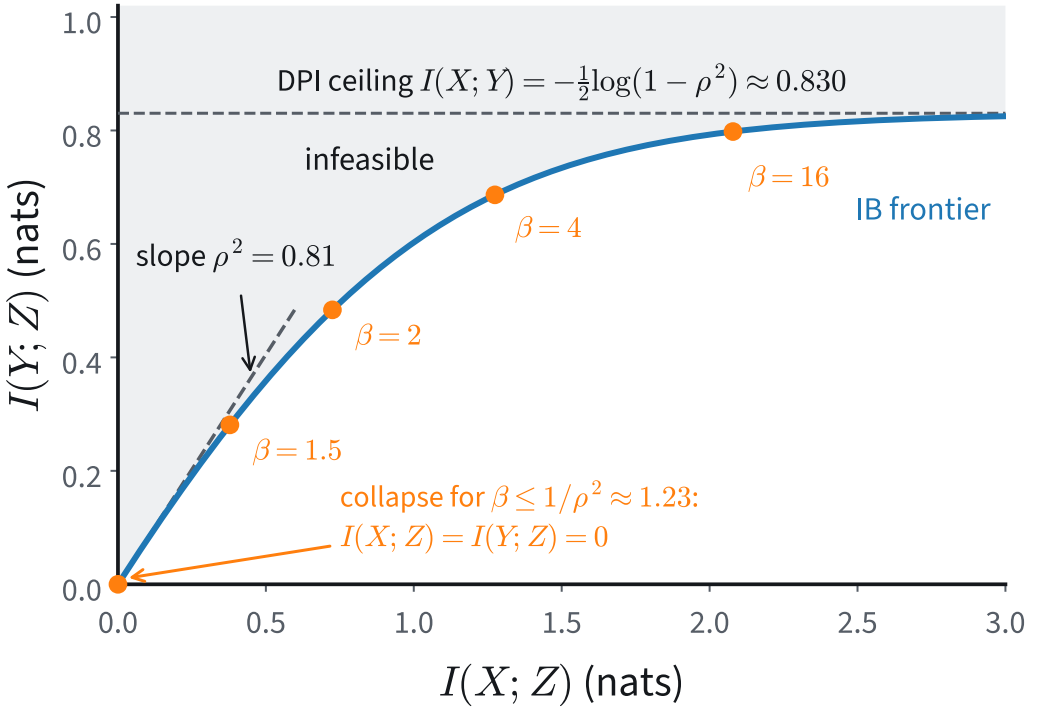


Figure 37.17: The information plane of the Gaussian bottleneck with $\rho = 0.9$. The concave frontier traces the largest attainable $I(Y;Z)$ at each rate $I(X;Z)$; the shaded region above it is infeasible, and the dashed ceiling is the data-processing limit $I(X;Y) = -\frac{1}{2}\log(1-\rho^2) \approx 0.830$ nats. The frontier leaves the origin with slope ρ^2, and the multiplier selects the point where the slope equals $1/\beta$; for $\beta \le 1/\rho^2 \approx 1.23$ that point collapses to the origin.

```
rho = 0.9
sigma = np.logspace(-2, 2, 4001)                  # encoder Z = X + sigma eps
I_xz = 0.5 * np.log(1 + sigma**-2)
I_yz = -0.5 * np.log(1 - rho**2 / (1 + sigma**2))
for beta in (1.2, 1.25, 1.5, 2, 4, 16):
    i = np.argmin(I_xz - beta * I_yz)             # Lagrangian on the grid
    u = (0.5 * np.log(rho**2 * (beta - 1) / (1 - rho**2))
         if beta * rho**2 > 1 else 0.0)           # closed-form optimum
    print(f'beta = {beta:5.2f}: I(X;Z) grid {I_xz[i]:.3f}, closed form {u:.3f};'
          f' I(Y;Z) = {I_yz[i]:.3f}')
```

```
beta =  1.20: I(X;Z) grid 0.000, closed form 0.000; I(Y;Z) = 0.000
beta =  1.25: I(X;Z) grid 0.032, closed form 0.032; I(Y;Z) = 0.026
beta =  1.50: I(X;Z) grid 0.379, closed form 0.378; I(Y;Z) = 0.281
beta =  2.00: I(X;Z) grid 0.725, closed form 0.725; I(Y;Z) = 0.484
beta =  4.00: I(X;Z) grid 1.275, closed form 1.274; I(Y;Z) = 0.687
beta = 16.00: I(X;Z) grid 2.078, closed form 2.079; I(Y;Z) = 0.798
```

The grid search and the closed-form solution agree to a thousandth of a nat, confirming the collapse threshold right where the proposition predicts: at $\beta = 1.2$ the constant encoder is optimal, whereas by $\beta = 1.25$ the encoder already retains 0.032 nats. Notice that this exact same encoder with noise scale σ also achieves the rate–distortion curve $R(D)$ for squared error at $D = \sigma^2/(1+\sigma^2)$. Rate–distortion judges the encoder by how well it reconstructs X, whereas the information bottleneck judges it by how well it predicts Y.

37.6.4.3 The Variational Information Bottleneck

For a Gaussian pair we could compute both mutual informations in the bottleneck objective analytically. On real data we can compute neither, because the true joint distribution is unknown: all we observe are samples (x, y) paired with an encoder $p(z \mid x)$ that we choose. The *variational information bottleneck* (Alemi et al. 2017) makes the objective trainable by replacing each mutual information with a tractable bound that can be estimated from samples.

For the label term, we already have the needed bound. Introduce a decoder $q(y \mid z)$ (a classifier predicting the label from the representation) with cross-entropy loss $L_Z = \mathbb{E}[-\log q(Y \mid Z)]$. Applying the loss certificate Equation 37.60 to the pair (Z, Y) immediately gives $I(Y;Z) \geq H(Y) - L_Z$.

Because we want to *minimize* the rate $I(X;Z)$, we need an upper bound on it instead. Write the rate in the conditional form Equation 37.55, $I(X;Z) = \mathbb{E}_x\big[D_{\mathrm{KL}}(p(z \mid x) \,\|\, p(z))\big]$. While the encoder $p(z \mid x)$ is known, the aggregated marginal $p(z)$ is not, since it averages $p(z \mid x)$ over the unknown data distribution. Replace $p(z)$ by any fixed prior density $r(z)$ of our choice, such as a standard normal. Using the identity $\log \frac{p(z|x)}{r(z)} = \log \frac{p(z|x)}{p(z)} + \log \frac{p(z)}{r(z)}$ and taking expectations over x and z gives

$$\mathbb{E}_x\Big[D_{\mathrm{KL}}\big(p(z \mid x) \,\|\, r(z)\big)\Big] = I(X;Z) + D_{\mathrm{KL}}\big(p(z) \,\|\, r(z)\big) \;\geq\; I(X;Z). \tag{37.70}$$

Any mismatch between our chosen prior r and the true marginal $p(z)$ can only inflate the rate bound; Exercise 9 calculates this gap explicitly for a Gaussian encoder. Combining both

bounds shows that the KL term on the left-hand side of Equation 37.70 plus βL_Z upper-bounds the bottleneck objective Equation 37.66 up to the constant $\beta H(Y)$. Because both terms are simple expectations over training examples, a neural network can minimize this bound directly with stochastic gradient descent.

Anyone who has trained a variational autoencoder will recognize the left-hand side of Equation 37.70 immediately. A variational autoencoder (Section 17.7) is precisely this same construction with the input serving as its own label: an encoder $p(z \mid x)$, a fixed prior $r(z)$, and a decoder that reconstructs x from z instead of predicting a separate class label. Its training loss is the reconstruction error plus a weighted KL penalty. Viewed through our lens, that KL penalty is an upper bound on the rate, capping how many nats the latent code can store about the input, while the reconstruction error plays the role of distortion. Only the placement of the trade-off weight differs: Equation 37.66 multiplies the preserved information by β, whereas a variational autoencoder places its weight on the KL rate term, so a large KL weight there corresponds to a small β here.

The decoder's cross-entropy loss and the KL divergence to a prior are the only quantities in this section that can be evaluated from samples alone—giving a lower bound on one mutual information and an upper bound on another. Every other number we computed relied on knowing the joint distribution, which we never have for real images or text. In Section 37.7, we ask how much about mutual information can be estimated from samples alone.

37.6.5 Summary

Mutual information is the KL divergence between a joint distribution and the product of its marginals, $I(X;Y) = D_{\mathrm{KL}}(P_{X,Y} \,\|\, P_X \otimes P_Y)$; it vanishes if and only if X and Y are independent and is invariant under invertible reparameterizations of either variable. Unlike linear correlation, it detects nonlinear dependence—assigning 1.14 nats to X and $Y = X^2 + 0.3\epsilon$ even though their correlation is zero. By the data-processing inequality, sequential processing cannot create information, though neural network layers can rearrange existing information to make it easier for a classifier to read. Fano's inequality turns any target error rate into a minimum requirement on mutual information, and the cross-entropy loss L of any probabilistic model certifies $I(X;Y) \geq H(Y) - L$. Shannon's channel capacity and rate–distortion function arise as the maximum of mutual information over inputs and its minimum over lossy encoders. Finally, the information bottleneck $\min I(X;Z) - \beta I(Y;Z)$ casts representation learning as a rate–distortion problem whose distortion is supplied by the label: for a Gaussian pair it collapses until β exceeds $1/\rho^2$, and in its variational form its rate penalty becomes the familiar KL regularizer of a variational autoencoder.

37.6.6 Exercises

1. code **Rare pairs and PMI.** Pointwise mutual information Equation 37.57 corrects co-occurrence counts for the frequencies of the two words (Section 37.6.1.4).
    1. Show that $\mathrm{pmi}(x,y) \leq \min\{-\log p_X(x), -\log p_Y(y)\}$. When is each of the two bounds attained?
    2. Extend `counts` by a fourth first word *sri* and a fourth second word *lanka* that occur only in the bigram "sri lanka", with count 2. Recompute the pmi of "new york", "machine learning", "the day", and "sri lanka", and repeat with count 1. Zero counts give pmi $= -\infty$ and make `(p * pmi).sum()` return `nan`, so restrict sums to the nonzero cells, as `mutual_information` does.

3. Compare the contributions $p_{X,Y}(x,y)\,\mathrm{pmi}(x,y)$ of "new york" and "sri lanka" to $I(X;Y)$. Then add 1 to every count and recompute both pmi values. Which of these scores would you use to rank collocations, and why?

2. **Information versus usable information.** A label Y is a fair bit. Given $Y=0$ the input $X \in \mathbb{R}^2$ is uniform on the circle of radius 1, and given $Y=1$ it is uniform on the circle of radius 2. A representation computes $Z = \|X\|^2$.

   1. Show that $I(Y;Z) = I(Y;X) = \ln 2$, although $x \mapsto \|x\|^2$ is not invertible. Which condition for equality in the data-processing inequality of Section 37.6.2.2 holds?

   2. Show that no linear classifier $\hat{y} = \mathbf{1}(\mathbf{w}^\top \mathbf{x} + b > 0)$ classifies every input correctly, whereas a threshold on Z does.

   3. The layers of a network form a Markov chain $X \to Z_1 \to Z_2 \to \cdots$. Show that the error of the best classifier of any kind that reads Z_ℓ cannot fall as ℓ grows. The accuracy of a linear classifier fit to Z_ℓ often rises with ℓ all the same. Why is there no contradiction?

   4. Xu et al. (2020) measure the information that a family $\mathcal{V}$ of predictors $q(y \mid x)$ can extract. Let $H_{\mathcal{V}}(Y \mid X) = \inf_{q \in \mathcal{V}} \mathbb{E}[-\log q(Y \mid X)]$, and let $H_{\mathcal{V}}(Y)$ be the same infimum over the predictors in $\mathcal{V}$ that ignore their input. The *usable information* is

      $$I_{\mathcal{V}}(X \to Y) = H_{\mathcal{V}}(Y) - H_{\mathcal{V}}(Y \mid X).$$

      Let $\mathcal{V}$ be logistic regression on the given input. Show that $I_{\mathcal{V}}(X \to Y) = 0$ and $I_{\mathcal{V}}(Z \to Y) = \ln 2$. For X, use that the expected logistic loss is convex in $(\mathbf{w}, b)$ and unchanged when $\mathbf{w}$ is rotated. Which inequality of Section 37.6.2.2 does $I_{\mathcal{V}}$ violate?

3. **Synergy and the Venn diagram.** Let X and Y be independent fair bits and $Z = X \oplus Y$ their exclusive or, for which the section computed $I(X;Y) = 0$ and $I(X;Y \mid Z) = \ln 2$.

   1. Which of the Markov chains $X \to Z \to Y$, $X \to Y \to Z$, and $Y \to X \to Z$ hold? What does the data-processing inequality therefore say about this triple?
   2. In a three-variable version of Figure 37.13, the area common to all three disks would be the *interaction information* $I(X;Y;Z) = I(X;Y) - I(X;Y \mid Z)$. Prove that $I(X;Y;Z) = H(X) + H(Y) + H(Z) - H(X,Y) - H(X,Z) - H(Y,Z) + H(X,Y,Z)$, and conclude that $I(X;Y;Z)$ is symmetric in its three arguments.
   3. Evaluate $I(X;Y;Z)$ for the triple of (a) and for three copies $X = Y = Z$ of one fair bit. What do the two values imply for a three-disk diagram of entropies?

   Adapted from Thomas M. Cover and Joy A. Thomas, Elements of Information Theory, 2nd ed., Exercise 2.25, and David J. C. MacKay, Information Theory, Inference, and Learning Algorithms, Exercise 8.8.

4. code **Data processing from f-divergences.** For any channel κ that maps an input to a random output, and any two distributions P and Q of the input, the output distributions satisfy $D_f(P\kappa \,\|\, Q\kappa) \le D_f(P \,\|\, Q)$ for every f-divergence (Section 37.3.3.3). The KL divergence and $\chi^2(P\|Q) = \sum_x p(x)^2/q(x) - 1$ are two of them.

1. Let $X \to Y \to Z$ be a Markov chain. Take $P = P_{X,Y}$, $Q = P_X \otimes P_Y$, and the channel κ that maps (x, y) to (x, z) with $z \sim p(z \mid y)$. Show that $P\kappa = P_{X,Z}$ and $Q\kappa = P_X \otimes P_Z$, and conclude that $I(X;Z) \le I(X;Y)$.
2. Which part of the section's data-processing proposition does this route not deliver?
3. For the two flips in series, with probabilities 0.1 and 0.2, compute the KL divergence and the χ^2 divergence of each joint table from the product of its marginals, before and after the second flip.
4. Show that for a fair input through the flip channel with probability δ, $\chi^2(P_{X,Y} \,\|\, P_X \otimes P_Y) = (1-2\delta)^2$, and check your numbers from (c). Conclude that a further flip with probability δ' multiplies this divergence by $(1-2\delta')^2$, here by 0.36. Is the ratio $I(X;Z)/I(X;Y)$ from
   (c) the same factor?

5. **Fano and a converse.** ● Fano's inequality Equation 37.59 bounds from below the error of any estimate $\hat{Y} = g(X)$ of a label Y with k values.
   1. Let $\hat{Y}(x) = \arg\max_y p(y \mid x)$ be the most probable label. Show that its error satisfies $P_e \le 1 - e^{-H(Y|X)}$. Bound $\max_y p(y \mid x)$ from below by $\sum_y p(y \mid x)^2 = \mathbb{E}[p(Y \mid X) \mid X = x]$, then apply Jensen's inequality once, to $\mathbb{E}[p(Y \mid X)] = \mathbb{E}\big[e^{\log p(Y|X)}\big]$.
   2. An observation X carries $I(X;Y) = 9$ nats about a balanced $10{,}000$-class label Y. Use Equation 37.59 and (a) to bracket the error of the best classifier. The inequality Equation 37.59 has no closed-form solution for P_e, so solve it numerically. What does the width of this interval say about predicting a downstream error rate from a mutual information?

6. **The loss certificate and Fano.** A classifier of $k = 10$ balanced classes has test error $P_e = 0.08$ and test loss $L = 0.30$ nats.
   1. How many nats of $I(X;Y)$ does Fano's inequality Equation 37.59 certify, and how many does the loss certificate Equation 37.60?
   2. For any estimate $\hat{Y} = g(X)$ with error probability P_e, let the model $q(y \mid x)$ put probability $1 - P_e$ on $y = g(x)$ and spread P_e evenly over the other $k-1$ labels. Show that its cross-entropy loss is $H_b(P_e) + P_e \log(k-1)$, and conclude that Fano's inequality is the loss certificate of this model. Which of the four models evaluated in Section 37.6.2.4 is it?
   3. The classifier of (a) reports a smaller loss than the model that (b) builds from its decisions. What must its confidences do for that to happen? Give a pair (X, Y) with a binary label and a model with error rate P_e that certifies more than Fano's inequality does.

7. **Binary channels.** ● Work in nats with the binary entropy H_b.
   1. A *Z-channel* passes a 0 unchanged and turns a 1 into a 0 with probability $\frac{1}{2}$. Compute $I(X;Y) = H(Y) - H(Y \mid X)$ as a function of $a = P(X=1)$, and show that it is largest at $a = \frac{2}{5}$, where it equals $\ln \frac{5}{4}$ nats. Why does the best input send the unreliable symbol less than half the time, and why not never?
   2. A feature takes the values ± 1 and passes through dropout with rate α, which replaces it by 0 with probability α and rescales it otherwise. Show that this *binary erasure channel* has capacity $(1-\alpha)\ln 2$ nats.

3. Find the flip probability δ at which the binary symmetric channel has the same capacity as dropout with $\alpha = 0.5$. Why does a flipped bit cost more capacity than an erased one?

8. **The Gaussian channel.** Let $Y = X + N$ with noise $N \sim \mathcal{N}(0, \sigma_N^2)$ independent of X, and let the input obey the power constraint $\mathbb{E}[X^2] \le P$.
   1. Show that $I(X;Y) = h(Y) - h(N)$ for any input with a density.
   2. Among densities with variance s^2, the Gaussian has the largest differential entropy, $\frac{1}{2}\log(2\pi e s^2)$. Use this to show $h(Y) \le \frac{1}{2}\log\left(2\pi e(P+\sigma_N^2)\right)$, and conclude that the capacity is $C = \frac{1}{2}\log(1 + P/\sigma_N^2)$, attained by a Gaussian input.
   3. Evaluate C at signal-to-noise ratios $P/\sigma_N^2 = 1, 10, 100$, in nats and in bits. Show that at high signal-to-noise ratio each doubling of P adds about $\frac{1}{2}\ln 2$ nats, half a bit.
   4. A diffusion model replaces a coordinate X of the data, scaled so that $\mathbb{E}[X^2] = 1$, by $\sqrt{a}\,X + \sqrt{1-a}\,\epsilon$ with $0 < a < 1$ and independent $\epsilon \sim \mathcal{N}(0,1)$. Show that the noised coordinate carries at most $\frac{1}{2}\log\frac{1}{1-a}$ nats about X, whatever the distribution of X, and evaluate the bound at $a = 0.5$ and $a = 0.01$. For which data is it attained?

9. **The rate bound of a variational autoencoder.** Let $X \sim \mathcal{N}(0,1)$, let the encoder be $Z = X + \sigma\epsilon$ as in Section 37.6.4.2, so that $p(z \mid x) = \mathcal{N}(z; x, \sigma^2)$, and let the prior be $r = \mathcal{N}(0,1)$. Use $D_{\mathrm{KL}}(\mathcal{N}(\mu, s^2) \,\|\, \mathcal{N}(0,1)) = \frac{1}{2}(\mu^2 + s^2 - 1 - \log s^2)$.
   1. Compute the three terms of Equation 37.70 and check the identity. By how many nats does the KL penalty overstate the rate at $\sigma = 1$?
   2. Rescale the code to $Z' = Z/\sqrt{1+\sigma^2}$. Show that $I(X;Z') = I(X;Z)$ and that the bound is now an equality.
   3. The rate does not depend on the units of the code, and the bound does. Which prior r makes the bound an equality for a given encoder? What does minimizing the KL penalty therefore do to the code, besides lowering its rate?

37.7 Estimating Mutual Information

Every mutual information we have computed so far came from a closed-form joint density. In practice, data never come with a density attached. Instead, we are given a collection of pairs, such as spoken sentences with their transcripts or search queries with the documents that answered them, and from those samples alone we want to quantify how much one member of a pair reveals about the other. Representation learning pushes this one step further: it trains models to make that shared information large, turning the estimate itself into a training objective.

A familiar contrastive loss makes the challenge concrete. Systems such as SimCLR and CLIP train on paired *views* of the same example, such as two random crops of a photograph or an image paired with its caption. Given one view, the model learns to identify its true partner among a batch of candidates, say 128 of them, by minimizing a cross-entropy loss known as the InfoNCE loss. Papers frequently subtract this loss from $\ln 128 = 4.85$ and report the difference as the mutual information between the two views. Yet that difference can never exceed 4.85 nats, no matter how much information the views actually share: a model that picks the right partner every time has only proved that it can distinguish 128 candidates. Because we both train models on such quantities and interpret them as measurements, we

need to distinguish a genuine measurement from a lower bound, and to separate the slack of a bound from the error of a learned model.

There are two main ways to estimate mutual information from samples. The direct approach counts samples, either inside histogram bins or within nearest-neighbor regions; it works well for small amounts of information between low-dimensional variables, but breaks down for vectors. The second approach replaces the unknown densities with a learned model arranged so that the resulting expectation is a lower bound on $I(X;Y)$, and then maximizes that bound. Two results anchor the rest of this section: the fundamental limit on what any estimator can guarantee without distributional assumptions (Section 37.7.2.4), and the variational bound behind the InfoNCE loss together with its logarithmic ceiling (Section 37.7.4).

```
from scipy.spatial.distance import cdist
from scipy.special import digamma, logsumexp
```

37.7.1 A Pair with Known Information

To judge an estimator fairly, we need a benchmark whose exact mutual information we already know. Throughout this section we use the standard bivariate Gaussian pair from Section 37.6.1.3 with correlation ρ, which shares

$$I(X;Y) = -\tfrac{1}{2}\log\left(1-\rho^2\right) \tag{37.71}$$

nats: 0.83 nats at $\rho = 0.9$ and 1.96 nats at $\rho = 0.99$. All logarithms in this section are natural.

The variational bounds we will derive rely on the *pointwise mutual information* (Section 37.6.1.4), $\mathrm{pmi}(x,y) = \log \frac{p(x,y)}{p(x)\,p(y)}$, which measures how much more likely a specific pair (x,y) is under the joint distribution than under independence. For our standard Gaussian pair, the pmi is a simple quadratic function:

$$\mathrm{pmi}(x,y) = -\tfrac{1}{2}\log(1-\rho^2) - \frac{x^2 - 2\rho xy + y^2}{2(1-\rho^2)} + \frac{x^2+y^2}{2}. \tag{37.72}$$

Two identities follow immediately from this definition. First, averaging $\mathrm{pmi}(x,y)$ over pairs drawn from the joint distribution gives $I(X;Y)$. Second, averaging its exponential $e^{\mathrm{pmi}(x,y)}$ over *independent* draws of x and y gives 1, because $p(x)\,p(y)\,e^{\mathrm{pmi}(x,y)} = p(x,y)$ integrates to 1. Let us verify both identities on samples. For the second identity, we break the dependence by pairing each x with the y from the next sample in the array. We call these *mismatched pairs*; each one is an independent draw from $P_X \otimes P_Y$, and we will reuse this simple trick whenever a bound requires independent pairs.

```
def gauss_pair(rng, rho, n):
    """Samples of the standard Gaussian pair; n is a count or a shape."""
    x = rng.standard_normal(n)
    y = rho * x + np.sqrt(1 - rho**2) * rng.standard_normal(n)
    return x, y
```

```python
def mi_gauss(rho):
    """I(X;Y) of the pair, in nats."""
    return -0.5 * np.log(1 - rho**2)

def pmi_gauss(x, y, rho):
    """Pointwise mutual information pmi(x, y) of the pair, in nats."""
    c = 1 - rho**2
    return (-0.5 * np.log(c) - (x**2 - 2 * rho * x * y + y**2) / (2 * c)
            + (x**2 + y**2) / 2)

rng = np.random.default_rng()
x, y = gauss_pair(rng, 0.9, 200_000)
print(f'I(X;Y) = {mi_gauss(0.9):.3f} nats, mean pmi over pairs = '
      f'{pmi_gauss(x, y, 0.9).mean():.3f}')
print(f'mean exp(pmi) over independent pairs = '
      f'{np.exp(pmi_gauss(x, np.roll(y, 1), 0.9)).mean():.3f}')
```

```
I(X;Y) = 0.830 nats, mean pmi over pairs = 0.833
mean exp(pmi) over independent pairs = 0.994
```

Both sample averages match their theoretical values to within about 0.01. Every variational bound in this section builds on these two identities, and it is the second one that will cause almost all of our statistical trouble. At $\rho = 0.9$, e^{pmi} is well behaved: its variance over independent pairs is $e^{2I} - 1$ (Exercise 1), which is only 4.3 here. At $I = 6$, however, that variance explodes to $160{,}000$, leaving the same $200{,}000$-sample average uncertain by roughly 0.9.

37.7.2 Nonparametric Estimation

The most direct way to estimate a probability distribution from data is to count how many samples fall into each region of space. The key design choice is how to define those regions. We begin with the simplest choice: a fixed grid.

37.7.2.1 Histograms

The histogram estimator partitions the range of each variable into b equal-width intervals, counts how many samples land in each of the $b \times b$ cells, and plugs the resulting empirical frequencies into the discrete formula for mutual information. Despite its simplicity, this estimator suffers from two competing biases that push in opposite directions.

The first bias pulls the estimate *down*. Replacing a continuous value by its bin index applies a deterministic function to X, and by the data-processing inequality (Section 37.6.2.2) any function of X shares at most as much information with Y as X itself does. Binning can therefore only destroy information, and the loss is severe whenever the dependence is narrower than a bin. At $\rho = 0.99$, for example, the conditional standard deviation of y given x is $\sqrt{1 - \rho^2} = 0.14$. Dividing a sample that spans roughly $[-4.5, 4.5]$ into 10 bins produces cells of width 0.9, more than six times wider than the conditional spread.

The second bias pulls the estimate *up*. The plug-in estimate is the KL divergence between the empirical joint table and the product of its row and column marginals. It is therefore always nonnegative, and equals zero only if the table factorizes exactly. Even when X and Y are completely independent, random sampling fluctuations prevent exact factorization,

so noise masquerades as dependence. The fewer samples each cell contains, the worse this upward bias becomes.

On top of these two errors, a histogram has a hard ceiling. Its estimate cannot exceed the entropy of the binned X, and b bins can carry at most $\log b$ nats: just 2.30 nats for ten bins, no matter how tightly X and Y are linked. We will encounter a version of this ceiling in every estimator in this section: no method can report more information than the logarithm of the number of states it can distinguish.

Let us implement this estimator as `binned_mi` and test it on $200{,}000$ samples from four distributions using $b = 10$, 40, and 160: an independent pair, the Gaussian pair at $\rho = 0.9$ and $\rho = 0.99$, and the parabolic pair $Y = X^2 + 0.3\,\epsilon$ from Section 37.6.1.5, which has zero linear correlation yet shares 1.141 nats.

```
def binned_mi(x, y, bins):
    """Plug-in MI of the bins x bins histogram of (x, y), in nats."""
    p = np.histogram2d(x, y, bins=bins)[0] / len(x)       # cell frequencies
    q = p.sum(axis=1, keepdims=True) * p.sum(axis=0, keepdims=True)
    nz = p > 0                                            # occupied cells
    return float((p[nz] * np.log(p[nz] / q[nz])).sum())

mi_parabola = 1.141  # by quadrature, sec_mdl-mi-nonlinear-correlation
n = 200_000
x = rng.standard_normal(n)
pairs = {'rho = 0': (gauss_pair(rng, 0.0, n), 0.0),
         'rho = 0.9': (gauss_pair(rng, 0.9, n), mi_gauss(0.9)),
         'rho = 0.99': (gauss_pair(rng, 0.99, n), mi_gauss(0.99)),
         'parabola': ((x, x**2 + 0.3 * rng.standard_normal(n)), mi_parabola)}
print('pair        truth  b = 10  b = 40  b = 160')
for name, ((u, v), truth) in pairs.items():
    print(f'{name:10}  {truth:5.3f}'
          + ''.join(f'  {binned_mi(u, v, b):6.3f}' for b in (10, 40, 160)))
```

```
pair        truth  b = 10  b = 40  b = 160
rho = 0     0.000   0.000   0.003   0.035
rho = 0.9   0.830   0.629   0.813   0.845
rho = 0.99  1.959   1.090   1.800   1.955
parabola    1.141   0.551   0.989   1.136
```

Both biases show up clearly in the table. With $b = 10$, every dependent pair reads low: roughly 0.6 instead of 0.83 nats at $\rho = 0.9$, and only about half the true value at $\rho = 0.99$ and for the parabola. With $b = 160$, the independent pair is credited with a few hundredths of a nat, and the pair at $\rho = 0.9$ actually overshoots its true value, something coarse binning alone could never cause. At $\rho = 0.99$ with $b = 160$, the estimate lands within about 0.01 of the truth, but only through a fortunate cancellation of opposite errors; the row directly above shows the same grid overshooting. When the true value is unknown, nothing in the table tells us which column to trust.

A third weakness is sensitivity to coordinates. Applying an invertible transformation to either variable leaves mutual information unchanged (Section 37.6.1.5), yet it distorts a fixed-width grid. Suppose we replace x with x^3 in the Gaussian pair at $\rho = 0.9$. The largest of $200{,}000$ standard normal draws is around ± 4.5, so x^3 spans roughly ± 90. Each of 40 equal-width bins is then about 4.5 wide, which packs roughly nine out of ten samples into

the two or three bins around zero. A clean fix is to bin the *ranks*, meaning each value's position in the sorted sample. Any strictly increasing map preserves ranks, and every rank bin automatically holds the same number of samples.

```
x, y = gauss_pair(rng, 0.9, n)
rank = lambda a: np.argsort(np.argsort(a))       # position in sorted order
print(f'true I = {mi_gauss(0.9):.3f} nats; histograms with b = 40:')
for name, u in (('x', x), ('x**3', x**3)):
    print(f'{name:4}: correlation {np.corrcoef(u, y)[0, 1]:.3f}, '
          f'raw {binned_mi(u, y, 40):.3f}, '
          f'ranks {binned_mi(rank(u), rank(y), 40):.3f} nats')
```

```
true I = 0.830 nats; histograms with b = 40:
x   : correlation 0.901, raw 0.819, ranks 0.819 nats
x**3: correlation 0.699, raw 0.439, ranks 0.819 nats
```

Even though the true mutual information remains 0.830 nats, cubing x drops the linear correlation from 0.90 to about 0.7 and slashes the raw histogram estimate to between 0.3 and 0.5 nats (with the exact value depending on the few extreme draws that set the bin span). Binning the ranks, by contrast, gives the exact same estimate for x and x^3, digit for digit, and stays within 0.02 nats of the truth.

Ranks fix the coordinate problem, but nothing can save histograms from high dimension. A pair of five-dimensional vectors requires a ten-dimensional grid; putting just ten bins along each axis creates 10^{10} cells, almost all of which will be empty for any realistic sample size. To handle vectors, an estimator must let the data themselves set the resolution, using small regions where samples are dense and larger ones where they are sparse.

37.7.2.2 The KSG Estimator

The estimator of Kraskov et al. (2004), known as KSG, replaces fixed histogram cells with adaptive nearest-neighbor regions around each data point. It is the standard nonparametric tool for pairs of low-dimensional variables. (This subsection and the next can be skipped on a first reading.)

KSG measures distances using the largest coordinate difference (the Chebyshev, or ℓ_∞, distance), so that a ball of radius r is a square in two dimensions and a hypercube in higher dimensions. Around each sample point i, we draw the smallest box in the joint (x, y) space that reaches its m-th nearest neighbor, and let r_i denote the half-width of that box. By construction, this box contains a fraction m/n of the n samples, giving a local estimate of the joint probability mass near (x_i, y_i).

Next, we measure the two marginal probabilities at the exact same scale r_i. Let $n_x^{(i)}$ be the number of other samples falling strictly inside the vertical strip $|x - x_i| < r_i$, and let $n_y^{(i)}$ be the number falling strictly inside the horizontal strip $|y - y_i| < r_i$. Sharing the half-width r_i is what lets raw counts stand in for probability densities: if the densities are roughly constant across the box, the box holds probability mass about $p(x_i, y_i)\,(2r_i)^2$ while the two strips hold about $p(x_i)\,2r_i$ and $p(y_i)\,2r_i$. When we divide the box fraction by the product of the two strip fractions, the geometric factors $2r_i$ cancel out. Taking the logarithm produces a local estimate of the pointwise mutual information computed purely by counting. Figure 37.18 illustrates these three counts at a single point.

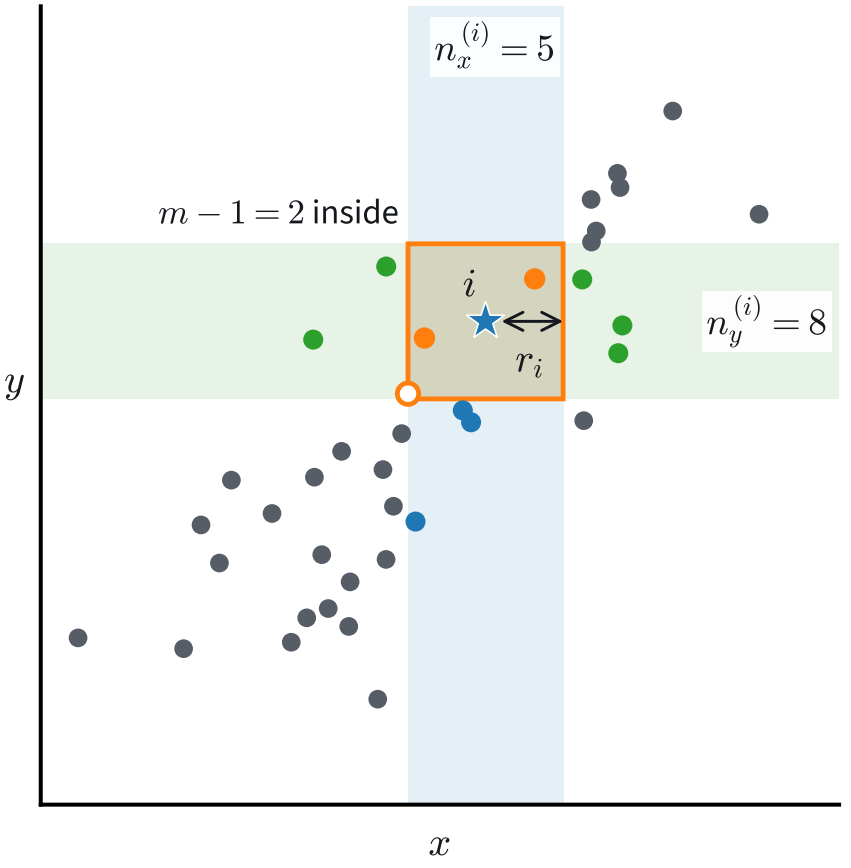


Figure 37.18: The KSG count at one sample point i (the star), with $m = 3$. The box of half-width r_i reaches the third nearest neighbor in the joint plane (the ring on its left edge, which lies in the horizontal strip and counts toward $n_y^{(i)}$ only), while two neighbors lie strictly inside. The vertical strip of the same half-width holds $n_x^{(i)}$ other samples and the horizontal strip holds $n_y^{(i)}$ (blue and green dots lie in one strip only). Each summand of the estimator is approximately $\log \frac{m/n}{(n_x/n)(n_y/n)}$, a pointwise mutual information computed from counts.

Averaging these local log ratios across all n points estimates $I(X;Y)$, with one finite-sample correction. Taking the logarithm of a small count is a biased estimate of a log probability: for the box reaching the m-th of $n-1$ neighbors, the expected log probability mass is $\mathbb{E}[\log p] = \psi(m) - \psi(n)$ rather than $\log(m/n)$, where $\psi = (\log\Gamma)'$ is the digamma function (we omit the derivation). Replacing logarithms with ψ gives the KSG estimator:

$$\hat{I}_{\text{KSG}} = \psi(m) + \psi(n) - \frac{1}{n}\sum_{i=1}^{n}\left(\psi\left(n_x^{(i)}+1\right) + \psi\left(n_y^{(i)}+1\right)\right). \tag{37.73}$$

Because $\psi(j) \approx \log j$ for large j, each summand is approximately $\log \frac{m\,n}{n_x^{(i)}\,n_y^{(i)}}$, matching the local log ratio of the box fraction to the product of the strip fractions. The $+1$ inside the strip terms aligns them with the box: $\psi(m)$ corresponds to a box whose boundary passes through its m-th neighbor, so a strip containing $n_x^{(i)}$ points strictly inside is treated as if its boundary passed through the $(n_x^{(i)}+1)$-th point. Both strip counts use strict inequalities and exclude point i itself, so a neighbor lying right on a strip boundary is not counted. Because every radius r_i adapts automatically to the local sample density, there is no bin width or kernel bandwidth to tune.

We can compute all neighbor counts at once from the $n \times n$ matrices of Chebyshev distances in x and y using `cdist`, taking their elementwise maximum for the joint distance.

```
def ksg_counts(x, y, m=5):
    """Strip counts n_x, n_y of every sample point; x, y are (n, d)."""
    dx, dy = cdist(x, x, 'chebyshev'), cdist(y, y, 'chebyshev')
    d = np.maximum(dx, dy)                        # joint Chebyshev distance
    np.fill_diagonal(d, np.inf)
    r = np.partition(d, m - 1, axis=1)[:, m - 1]  # half-width: m-th neighbor
    n_x = (dx < r[:, None]).sum(axis=1) - 1       # strict, self excluded
    n_y = (dy < r[:, None]).sum(axis=1) - 1
    return n_x, n_y

def ksg_mi(x, y, m=5):
    """KSG nearest-neighbor estimate of I(X;Y) in nats; x, y are (n, d)."""
    n_x, n_y = ksg_counts(x, y, m)
    return float(digamma(m) + digamma(len(x))
                 - (digamma(n_x + 1) + digamma(n_y + 1)).mean())
```

Because every summand in Equation 37.73 depends solely on neighbor counts, counting alone places a hard ceiling on what KSG can report.

Proposition (sample-size ceiling of KSG). *For any sample of n points in which no two joint distances from the same point are equal,*

$$\hat{I}_{\text{KSG}} \le \psi(n) - \psi(m), \tag{37.74}$$

with equality when $Y = X$.

Proof. Without distance ties, exactly $m-1$ samples lie strictly closer to point i than its m-th neighbor. Since the joint Chebyshev distance is the larger of the two coordinate distances, each of these $m-1$ points satisfies both $|x_j - x_i| < r_i$ and $|y_j - y_i| < r_i$, so it lies in both

strips: $n_x^{(i)} \geq m-1$ and $n_y^{(i)} \geq m-1$. Because ψ is strictly increasing, every summand is at most $\psi(m)+\psi(n)-2\psi(m) = \psi(n)-\psi(m)$. When $Y = X$, the joint and marginal distances coincide, so both strips contain only the $m-1$ points strictly inside the box and every summand attains the bound. ■

For $n = 2{,}000$ and $m = 5$, this ceiling is $\psi(2000)-\psi(5) = 6.09$ nats, close to $\log(n/m) = 5.99$. Even for $Y = X$ with continuous X, where the true mutual information is infinite, KSG reports 6.1 nats, and doubling the sample size raises that ceiling by only about $\log 2$. We are seeing the histogram's $\log b$ ceiling again, with adaptive neighborhoods in place of bins: a neighbor-counting estimator can report at most the logarithm of the number of distinct neighborhoods the sample affords, roughly n/m.

37.7.2.3 KSG in High Dimension

Unfortunately, in higher dimensions KSG breaks down well below its sample-size ceiling, and for a geometric reason. Consider d independent copies of our Gaussian pair, each with $\rho = 0.9$, so that X and Y are d-dimensional vectors. Because the log density ratio splits into a sum of d independent terms, mutual information is additive across independent coordinates, making the true value $d \times 0.83$ nats.

KSG relies on the joint and marginal densities being nearly constant across each neighbor box, an assumption that collapses as dimension grows. With $n = 2{,}000$ and $m = 5$, each box captures a fraction $m/n = 0.0025$ of the sample. If points were spread uniformly across $2d$ dimensions, such a box would span a fraction $(m/n)^{1/(2d)}$ of the data's width along every axis: 0.05 at $d = 1$, jumping to 0.47 at $d = 4$ and 0.83 at $d = 16$. Yet the dependence lives inside a thin band around $y = \rho x$: given x, each coordinate of y has standard deviation $\sqrt{1-\rho^2} = 0.44$, roughly 0.07 of the total span (about -3.3 to 3.3) of the sample. A box many times wider than that band smears out the dependence just like an overly coarse histogram bin. Let us run KSG on $2{,}000$ samples for $d = 1, 2, 4, 8, 16$, recording both the estimate and the average strip count, along with the $Y = X$ case that attains the ceiling.

```
n, rho = 2000, 0.9
print(' d  true I    KSG  mean strip count')
for dim in (1, 2, 4, 8, 16):
    x, y = gauss_pair(rng, rho, (n, dim))        # dim independent pairs
    n_x, n_y = ksg_counts(x, y)
    print(f'{dim:2d}  {dim * mi_gauss(rho):6.2f}  {ksg_mi(x, y):5.2f}  '
          f'{(n_x.mean() + n_y.mean()) / 2:16.1f}')
x = rng.standard_normal((n, 1))
print(f'Y = X:  KSG = {ksg_mi(x, x):.2f}, '
      f'ceiling psi(n) - psi(m) = {digamma(n) - digamma(5):.2f} nats')
```

```
 d  true I    KSG  mean strip count
 1    0.83   0.83              71.3
 2    1.66   1.60              50.0
 4    3.32   2.51              31.3
 8    6.64   2.85              26.0
16   13.29   2.75              26.8
Y = X:  KSG = 6.09, ceiling psi(n) - psi(m) = 6.09 nats
```

In one and two dimensions, KSG stays within about 0.1 nat of the truth. By $d = 4$, it drops

to roughly 2.5 nats against a true value of 3.3, nearly a full nat low. After that it stalls: at both $d = 8$ and $d = 16$, KSG reports just under 3 nats even as the true mutual information doubles from 6.6 to 13.3. Notice that the sample-size ceiling is not what stops it here, since 3 nats is less than half of the 6.1-nat ceiling printed for $Y = X$.

The average strip counts expose why the estimate stalls. Each summand is approximately $\log \frac{m\,n}{n_x n_y}$, so the estimate can climb only if the marginal strips empty out. They drop from about 70 points at $d = 1$ to about 31 at $d = 4$, and then level off near 26. To keep boxes as local at $d = 16$ as they were at $d = 1$, we would need $m/n = 0.05^{32}$, requiring a sample of roughly 10^{42} points. The sample size needed to maintain local neighborhoods grows exponentially with the dimension.

37.7.2.4 Limits of Distribution-Free Estimation

Perhaps a more clever estimator could bypass this bottleneck. To test that hope, let us ask for something weaker than an exact measurement: a *lower bound*. Specifically, we seek a function B of n samples such that $B \le I(X;Y)$ with probability at least 0.95 *for every distribution* of the pair (X, Y). (Any confidence level above $1/2$ would work just as well.) Lower bounds are also what representation learning wants, since we want to be confident that two views share a large amount of information.

The simplest possible case already reveals the fundamental barrier. Set $Y = X$ for a discrete variable X, so that $I(X;Y) = H(X)$. If an estimator treats the discrete symbols as arbitrary labels with no built-in metric or ordering, it cannot exploit *which* symbols appeared. All it can inspect is the pattern of repetitions: how many distinct values appeared once, how many appeared twice, and so on. This summary is called the *type* of the sample, and we assume B depends only on it. Four short steps establish the limit, with the third step doing the real work:

1. Let X be uniform over M symbols and draw n samples. Any given pair of draws collides with probability $1/M$, so by the union bound the probability that *some* two of the $n(n-1)/2$ pairs coincide is at most $n(n-1)/(2M)$.
2. Apply the 0.95 guarantee to the uniform distribution on $M = n^2$ symbols, whose entropy is $2 \log n$: on a sample drawn from *this* distribution, $B \le 2 \log n$ with probability at least 0.95.
3. By the first step, a sample of size n from $M = n^2$ symbols consists of n distinct values with probability greater than $\frac{1}{2}$. Because two events whose probabilities sum to more than 1 must overlap, there exists at least one all-distinct sample on which $B \le 2 \log n$.
4. Every sample of n distinct values has the exact same type, so B must return the same value (at most $2 \log n$) on *every* all-distinct sample, no matter what distribution actually generated it.

Now test this bound on a uniform distribution over $M = 2^{256}$ symbols, whose entropy is $256 \ln 2 = 177$ nats. With $n = 10^6$ samples, all draws are distinct with probability at least $1 - 4.3 \times 10^{-66}$, and on that sample B can certify at most $2 \log 10^6 = 27.6$ nats. A million samples cannot guarantee 177 nats; in fact, they cannot even guarantee 30. Exercise 4 sharpens the constant and asks how many samples would be needed.

McAllester and Stratos (2020) proved the general theorem without restricting the estimator to depend only on the sample type:

> **Theorem (McAllester–Stratos, informal).** Any distribution-free lower bound on $I(X;Y)$ that holds with fixed confidence from n samples is $O(\log n)$.

The invariance of mutual information lies behind this barrier. Because any invertible relabelling of X or Y leaves $I(X;Y)$ unchanged (Section 37.6.1.5), an estimator that makes no assumptions about the coordinates must behave consistently under every possible relabelling, leaving only the pattern of repetitions to work with. To break past $O(\log n)$, an estimator must assume something about the data. KSG assumes that coordinate distances are meaningful and that densities are smooth, yielding an estimate with no finite-sample guarantee. The variational bounds of Section 37.7.3 can report numbers larger than $\log n$, and in Section 37.7.4.4 we will see how much we can actually trust them when they do.

37.7.2.5 The Compression-Phase Debate

The two limitations we have just seen, coordinate-dependent histograms and the logarithmic sample ceiling, shed light on a famous debate in deep learning. Shwartz-Ziv and Tishby (2017) trained a small classifier on the $4{,}096$ binary patterns of 12 bits, binned the activations of each hidden layer Z throughout training, and plotted estimates of $I(X;Z)$ and $I(Y;Z)$ for input X and label Y. They observed a brief *fitting* phase in which both quantities rose, followed by a long *compression* phase in which $I(X;Z)$ dropped while $I(Y;Z)$ remained high, and proposed that this compression explains why deep networks generalize. Later, Saxe et al. (2018) showed that this compression phase appears with saturating activations like tanh, but disappears with ReLU activations, even though ReLU networks generalize just as well.

What was the histogram actually measuring? In a deterministic network, the binned activation Z_{bin} is a deterministic function of the input X. Consequently, $H(Z_{\text{bin}} \mid X) = 0$, and the binned estimate simplifies to $I(X;Z_{\text{bin}}) = H(Z_{\text{bin}})$, simply the entropy of the binned activations. That quantity can never exceed the logarithm of the number of inputs, $\ln 4096 = 8.3$ nats, and it drops whenever distinct inputs collapse into the same histogram cell, exactly what happens when training drives saturating tanh units into the two outer bins near ± 1. The true mutual information $I(X;Z)$ itself never moves: as long as distinct inputs produce distinct activations, $I(X;Z) = H(X)$ for a finite input set, and for continuous inputs $I(X;Z)$ is infinite (Section 37.6.1.3). The plotted curves track how activations cluster across fixed bins, a property of the estimator rather than of the network's true mutual information (Exercise 8).

37.7.3 Variational Lower Bounds

Because counting cannot measure large mutual information in high dimensions, variational methods change the goal. Rather than estimating the unknown densities in $\log \frac{p(x,y)}{p(x)\,p(y)}$ directly, we introduce an auxiliary model arranged so that its expected score is guaranteed to be a *lower bound* on $I(X;Y)$, no matter how imperfect the model is. Training the model to maximize that expectation tightens the bound without ever overshooting $I(X;Y)$. Estimation thus becomes optimization. Its output carries two distinct sources of error: approximation error from the auxiliary model, which may fall short of the optimal one, and sampling error, because we must replace population expectations with finite batch averages.

We examine four variational bounds in turn, unified in a single framework by Poole et al. (2019) . The first uses a normalized decoder (Barber–Agakov). The next two use an unnormalized scoring function, yielding the Donsker–Varadhan (DV) bound and the bound of Nguyen, Wainwright, and Jordan (NWJ). The fourth, InfoNCE, is important enough to

merit its own section. Every proof relies on a single tool, Gibbs' inequality (Section 37.1): a KL divergence is always nonnegative.

37.7.3.1 The Barber–Agakov Bound

The simplest variational bound replaces the unknown posterior $p(x \mid y)$ with a tractable model $q(x \mid y)$.

Proposition (Barber–Agakov bound). *For any conditional density $q(x \mid y)$,*

$$I(X;Y) \geq H(X) + \mathbb{E}_{(x,y)\sim P_{X,Y}} \left[\log q(x \mid y)\right], \tag{37.75}$$

with equality if and only if $q(x \mid y) = p(x \mid y)$. For continuous X, the differential entropy $h(X)$ replaces $H(X)$.

Proof. Starting from $I(X;Y) = H(X) - H(X \mid Y)$, we subtract the right-hand side of Equation 37.75:

$$\begin{aligned} I(X;Y) - H(X) - \mathbb{E}\left[\log q(x \mid y)\right] &= \mathbb{E}\left[\log \frac{p(x \mid y)}{q(x \mid y)}\right] \\ &= \mathbb{E}_y \left[D_{\mathrm{KL}}(p(\cdot \mid y) \,\|\, q(\cdot \mid y))\right] \geq 0. \end{aligned}$$

The inequality follows from Gibbs' inequality applied for each y and averaged over P_Y. Equality holds if and only if the KL divergence vanishes for every y, meaning $q(x \mid y) = p(x \mid y)$. ■

The Barber–Agakov bound (Barber and Agakov 2003) gives an information-theoretic reading of an encoder–decoder: if a decoder $q(x \mid y)$ reconstructs X from Y with low cross-entropy, then X and Y must share a large amount of information. Its practical limitation is the term $H(X)$, which is usually unknown. Fortunately, $H(X)$ does not depend on q, so *maximizing* the bound over q never requires knowing it. Writing $H(X) = -\mathbb{E}[\log p(x)]$ yields the equivalent ratio form $I(X;Y) \geq \mathbb{E}\left[\log\left(q(x \mid y)/p(x)\right)\right]$, which will be the key step in proving the InfoNCE bound.

Throughout this section, we say that a model *certifies* the value of its bound. Keep in mind that this certification holds at the level of population expectations; a finite batch average only estimates the certified value and can fluctuate above or below it.

A standard classifier is a familiar special case. By the symmetry of mutual information, we can swap X and Y and treat a classifier $q(y \mid x)$ as a decoder of the label. This recovers the loss certificate of Section 37.6.2.4: for k balanced classes and held-out cross-entropy L, $I(X;Y) \geq \log k - L$. Because $L \geq 0$, a 10-class classifier can never certify more than $\ln 10 = 2.30$ nats. InfoNCE will take the exact same form.

Our Gaussian pair illustrates how much a mismatched decoder can lose. Training a predictor by squared error alone corresponds to a unit-variance Gaussian decoder $q(x \mid y) = \mathcal{N}(x; ay, 1)$. Since $h(X) = \frac{1}{2}\log(2\pi e)$, the Barber–Agakov bound evaluates to $\frac{1}{2} - \frac{1}{2}\mathbb{E}[(X - aY)^2] = \frac{1}{2} - \frac{1}{2}(1 - 2a\rho + a^2)$, which peaks at $a = \rho$ with value $\rho^2/2$. At $\rho = 0.9$, this decoder certifies only 0.41 nats out of the true 0.83, and no matter how close ρ gets to 1, it can never certify more than $\frac{1}{2}$ nat. The missing information is the penalty for

fixing the decoder's variance at 1 instead of matching the true conditional variance $1 - \rho^2$ (Exercise 2).

37.7.3.2 The Donsker–Varadhan Bound

Our second bound removes both requirements of Barber–Agakov: it needs neither $H(X)$ nor a normalized density q. Because mutual information is the KL divergence from the joint distribution to the product of its marginals (Section 37.6), any lower bound on KL divergence immediately bounds $I(X;Y)$. The Donsker–Varadhan bound achieves this using an arbitrary scoring function T, called a *critic* (as in Section 37.3.4), with no normalization constraint whatsoever.

Proposition (Donsker–Varadhan bound). *For distributions P and Q and any function T for which $\mathbb{E}_P[T]$ is defined and $\mathbb{E}_Q[e^T] < \infty$,*

$$D_{\mathrm{KL}}(P\|Q) \geq \mathbb{E}_P[T] - \log \mathbb{E}_Q\left[e^T\right], \tag{37.76}$$

with equality if and only if $T = \log \frac{dP}{dQ} + c$ for some constant c. When $P \ll Q$ and the divergence is finite, the supremum over T attains $D_{KL}(P\|Q)$, giving the Donsker–Varadhan representation of KL divergence (Donsker and Varadhan 1983).

Proof. If P is not absolutely continuous with respect to Q, the left-hand side is infinite and the inequality is trivial. Otherwise, use the critic to reweight Q: define the *tilted distribution* $\tilde{Q}$ with density $\tilde{q}(x) = q(x)\, e^{T(x)}/Z$, where $Z = \mathbb{E}_Q[e^T]$ is the normalizing constant. Then

$$\begin{aligned} D_{\mathrm{KL}}(P\|Q) - (\mathbb{E}_P[T] - \log Z) &= \mathbb{E}_P\left[\log \frac{p}{q} - T + \log Z\right] \\ &= \mathbb{E}_P\left[\log \frac{p}{\tilde{q}}\right] = D_{\mathrm{KL}}(P\|\tilde{Q}) \geq 0 \end{aligned}$$

by Gibbs' inequality. Equality holds if and only if $P = \tilde{Q}$, which means $p/q = e^T/Z$, or equivalently $T = \log(p/q) + \log Z$. $\blacksquare$

The slack in the bound is simply $D_{\mathrm{KL}}(P\|\tilde{Q})$, so a critic is accurate precisely when tilting Q by e^T reproduces P. Notice also that shifting T by any constant c cancels between the two terms. Setting $P = P_{X,Y}$ and $Q = P_X \otimes P_Y$ gives a lower bound on mutual information for any critic $T(x, y)$:

$$I(X;Y) \geq \mathbb{E}_{P_{X,Y}}[T] - \log \mathbb{E}_{P_X \otimes P_Y}\left[e^T\right],$$

whose optimal critic is $\mathrm{pmi}(x, y)$ up to an arbitrary additive constant. In practice, we estimate the first expectation by averaging T over observed pairs and the second by averaging e^T over mismatched pairs from the same minibatch. Parameterizing T with a neural network yields *MINE* (mutual information neural estimation) (Belghazi et al. 2018).

37.7.3.3 The NWJ Bound

Although the Donsker–Varadhan bound is tight in expectation, the logarithm outside $\mathbb{E}_Q[e^T]$ creates a subtle bias on minibatches. Because log is concave, Jensen's inequality implies that the logarithm of a batch average is on average smaller than the logarithm of the true expectation. Subtracting a smaller number pushes the batch estimate $\overline{T} - \log \overline{e^T}$ *upward* on average, an undesirable property for an estimator of a lower bound.

We can eliminate the logarithm using the elementary tangent inequality $\log z \le z/e$, which holds for all $z > 0$ with equality at $z = e$. Applying it to $Z = \mathbb{E}_Q[e^T]$ in Equation 37.76 gives, for every critic T,

$$I(X;Y) \ge \mathbb{E}_{P_{X,Y}}[T] - e^{-1}\,\mathbb{E}_{P_X \otimes P_Y}\left[e^T\right]. \tag{37.77}$$

This is the *NWJ bound* (Nguyen et al. 2010). For any fixed critic T, NWJ is never larger than DV, matching it only when $Z = e$. It is tight at $T^* = 1 + \text{pmi}$: whereas DV is invariant to adding a constant to the critic, NWJ pins that constant to 1. Writing $e^{-1}\,\mathbb{E}_Q[e^T] = \mathbb{E}_Q[e^{T-1}]$ shows that NWJ is simply the general f-divergence dual bound of Section 37.3.4 specialized to the KL divergence, $D_{\mathrm{KL}}(P\|Q) \ge \mathbb{E}_P[T] - \mathbb{E}_Q[e^{T-1}]$, derived there from the convex conjugate of $u \log u$. For KL divergence, Donsker–Varadhan strictly sharpens that general bound.

Because no logarithm wraps the second expectation, evaluating Equation 37.77 on a minibatch yields an *unbiased* estimate of the lower bound. Unbiased, however, does not mean reliable. At the optimal critic, $e^{T^*-1} = e^{\text{pmi}}$ is the density ratio $p(x,y)/(p(x)p(y))$. Its expectation over independent pairs is 1, but when X and Y are strongly dependent, that expectation is dominated by a tiny fraction of huge values mixed among many values near zero. Any batch that happens to miss those rare spikes severely underestimates the second term and therefore overshoots the bound. We will quantify and measure this behavior in Section 37.7.4.4.

37.7.4 InfoNCE

37.7.4.1 The Contrastive Loss

Because the exponential term e^T in the DV and NWJ bounds is unnormalized, a single extreme pair can dominate an entire batch. InfoNCE avoids this instability by normalizing *within* the batch, forcing the candidates to compete against one another. Specifically, we draw one *positive* pair $(x, y^+) \sim P_{X,Y}$, calling x the *anchor*, together with $N-1$ *negatives* $y_2, \dots, y_N$ drawn independently from the marginal P_Y. We score all N candidates with a critic $T(x,y)$ and pass the scores through a softmax to predict which candidate is the true partner. The *InfoNCE loss* (Oord et al. 2018) is the resulting cross-entropy:

$$\mathcal{L}_{\text{NCE}} = -\mathbb{E}\left[\log \frac{e^{T(x,y^+)}}{e^{T(x,y^+)} + \sum_{j=2}^{N} e^{T(x,y_j)}}\right], \tag{37.78}$$

where the expectation runs over both the positive pair and the $N-1$ negatives. The corresponding mutual information estimate subtracts this loss from $\log N$:

$$\hat{I}_{\text{NCE}} = \log N - \mathcal{L}_{\text{NCE}}. \tag{37.79}$$

Some papers write the denominator as an average $\frac{1}{N}\sum_{j=1}^{N} e^{T(x,y_j)}$, in which case the expected log ratio directly equals $\hat{I}_{\mathrm{NCE}}$. Conceptually, $\mathcal{L}_{\mathrm{NCE}}$ is simply an N-class cross-entropy whose classes are the other examples in the batch rather than fixed labels. This is the training objective behind CPC, SimCLR, and CLIP (Oord et al. 2018; Chen et al. 2020; Radford et al. 2021), where x and y are two views or two modalities of the same example and the remaining examples in the batch serve as negatives.

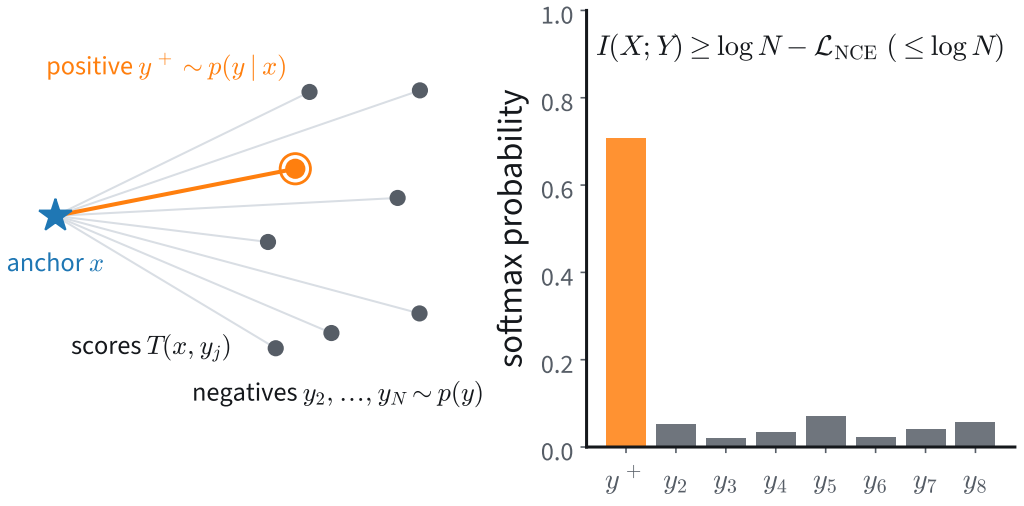


Figure 37.19: The InfoNCE task. Left: an anchor x is scored by the critic against its positive partner y^+, drawn jointly with it, and $N-1$ negatives drawn independently from the marginal; the rays carry the scores $T(x, y_j)$. Right: a softmax over the N scores must identify the positive. The resulting cross-entropy certifies $I(X;Y) \geq \log N - \mathcal{L}_{\mathrm{NCE}}$, a bound that can never exceed $\log N$.

Figure 37.19 illustrates the setup. In the smallest nontrivial case, $N = 2$, the model faces a single negative and the loss reduces to binary cross-entropy: telling the true partner from one distractor. Even a perfect model can then demonstrate only that it chose the right option out of two, so the estimate is capped at $\ln 2 = 0.69$ nats.

37.7.4.2 The InfoNCE Bound

The following proposition confirms that minimizing the InfoNCE loss certifies a lower bound on mutual information, and shows where the $\log N$ ceiling comes from (Poole et al. 2019).

Proposition (InfoNCE bound). *For any measurable critic T and any $N \geq 1$,*

$$I(X;Y) \geq \log N - \mathcal{L}_{\mathrm{NCE}}. \tag{37.80}$$

Moreover, the right-hand side is at most $\log N$.

Proof. Label the candidates $y_1 = y^+, y_2, \ldots, y_N$ and collect them into a single random bag $V = (Y_1, \ldots, Y_N)$, where the negatives $Y_{2:N}$ are independent of (X, Y_1). The strategy is to

replace Y with the bag V, construct a normalized conditional distribution over bags given X, and invoke the Barber–Agakov bound.

Step 1: independent negatives add no information. By the chain rule for mutual information (Section 37.6), $I(X; Y_1, Y_{2:N}) = I(X; Y_1) + I(X; Y_{2:N} \mid Y_1)$. Because $Y_{2:N}$ is independent of (X, Y_1), it remains independent of X given Y_1. The second term is therefore 0, giving $I(X; V) = I(X; Y)$.

Step 2: the softmax ratio defines a normalized model. Here is the key calculation. Define the rescaled softmax score

$$\gamma(x, v) = \frac{e^{T(x, y_1)}}{\frac{1}{N} \sum_{j=1}^{N} e^{T(x, y_j)}},$$

which takes values in $(0, N]$. Let $p(v)$ be the distribution of a bag of N independent draws from P_Y, including the first slot. Because all N entries are i.i.d., permuting them leaves $p(v)$ unchanged. Consequently, all N ratios $e^{T(x, Y_i)} / \left(\frac{1}{N} \sum_j e^{T(x, Y_j)}\right)$ have the same expectation under $p(v)$, and for every fixed x we have

$$\begin{aligned} \mathbb{E}_{p(v)}\left[\gamma(x, V)\right] &= \frac{1}{N} \sum_{i=1}^{N} \mathbb{E}_{p(v)} \left[\frac{e^{T(x, Y_i)}}{\frac{1}{N} \sum_j e^{T(x, Y_j)}} \right] \\ &= \mathbb{E}_{p(v)} \left[\frac{\frac{1}{N} \sum_i e^{T(x, Y_i)}}{\frac{1}{N} \sum_j e^{T(x, Y_j)}} \right] = 1. \end{aligned}$$

Thus $q(v \mid x) = p(v)\, \gamma(x, v)$ is a valid conditional probability distribution over bags.

Step 3: apply Barber–Agakov to the bag. Because Y_1 has marginal P_Y and is independent of $Y_{2:N}$, the marginal distribution of V is also $p(v)$. Applying the ratio form of Equation 37.75 to (X, V) with the roles of the two variables swapped (using the symmetry of I) gives

$$I(X; V) \geq \mathbb{E}_{p(x,v)} \left[\log \frac{q(v \mid x)}{p(v)} \right] = \mathbb{E}\left[\log \gamma(X, V)\right] = \log N - \mathcal{L}_{\text{NCE}}.$$

The final equality holds because $\log \gamma$ equals $\log N$ plus the log-softmax probability assigned to the positive. Combining this with Step 1 proves the bound. Finally, since the softmax probability of the positive cannot exceed 1, we have $\mathcal{L}_{\text{NCE}} \geq 0$, and hence the right-hand side is at most $\log N$. ■

Intuitively, picking the true partner out of N candidates, even with zero error, demonstrates at most $\log N$ nats: the information contained in a single choice among N alternatives. No critic can squeeze more information out of that task, just as a 10-class classifier could never certify more than $\ln 10$ nats. Increasing N raises the ceiling, which is one motivation given for the large batches of contrastive methods such as CPC, SimCLR, and CLIP. It is not the whole story: as we will see at the end of the section, a tighter bound is not what makes a representation good. The loss is minimized by any critic of the form $T^*(x, y) = \text{pmi}(x, y) + c(x)$, where $c(x)$ is an arbitrary function of the anchor alone (Exercise 5).

37.7.4.3 The Ceiling with an Exact Critic

One might naturally expect that with the exact critic $T = \text{pmi}$, $\hat{I}_{\text{NCE}}$ tracks the true $I(X;Y)$ all the way up to $\log N$ and then flattens out at a sharp corner. Substituting $T = \text{pmi}$ into Equation 37.79 reveals a softer, and more sobering, behavior:

$$\hat{I}_{\text{NCE}} = I(X;Y) - \mathbb{E}\left[\log \frac{1}{N}\sum_{j=1}^{N} e^{\text{pmi}(x,y_j)}\right]. \tag{37.81}$$

The estimate equals the true mutual information minus a slack term, which is nonnegative by the proposition. We can estimate the size of this slack with a quick back-of-the-envelope calculation. Inside the logarithm sits an average of N terms: the $N-1$ negatives are independent of x, so each has mean 1 (the second identity we checked at the start of the section), contributing roughly 1 to the average; meanwhile, the positive contributes $e^{\text{pmi}(x,y^+)}/N$, which is roughly e^I/N for a typical pair. Replacing the random average by this typical value yields

$$\hat{I}_{\text{NCE}} \approx I - \log\left(1 + \frac{e^I}{N}\right) = -\log\left(e^{-I} + \frac{1}{N}\right). \tag{37.82}$$

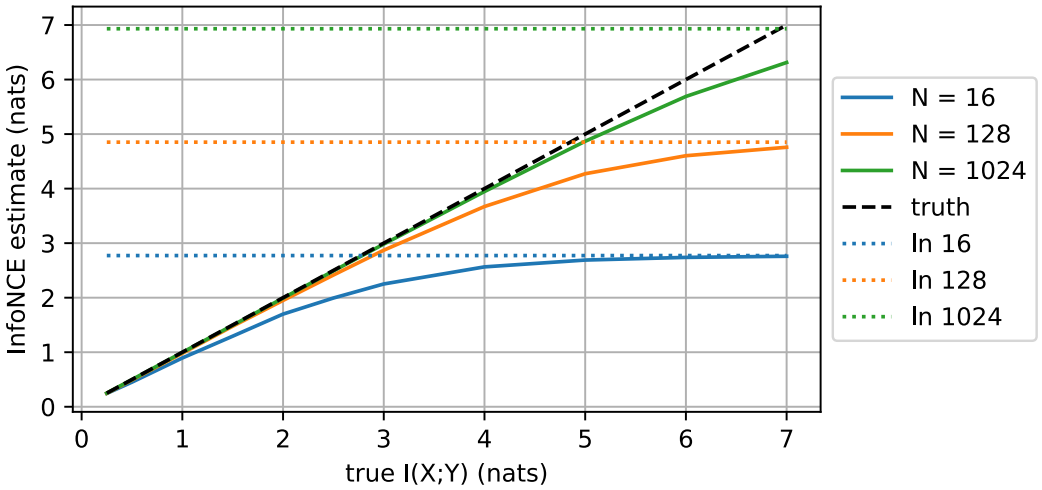
The right-hand side is a smooth minimum of I and $\log N$. It tracks I closely when $e^I \ll N$, approaches $\log N$ when $e^I \gg N$, and bends well below both in the transition region: right at $I = \log N$, it reads $\log N - \log 2$, falling 0.69 nats short. Our initial guess forgot that the positive sits inside its own denominator alongside negatives that average 1; the positive dominates the denominator only once e^I greatly exceeds N.

Because replacing a heavy-tailed average with its mean is a rough heuristic, we should check Equation 37.82 empirically. For $N = 128$ and $I = 4$, it predicts 3.6 nats even though the ceiling $\ln 128 = 4.85$ still looks comfortably far away. Below, we score all $N \times N$ pairs in a batch simultaneously: row i pairs anchor x_i with every y_j, so its diagonal entry is the positive and the $N-1$ off-diagonal entries are its negatives. Although the rows share candidates, the bound remains valid (Exercise 6). Let us sweep $I(X;Y)$ for $N = 16$, 128, and 1024 (ceilings 2.77, 4.85, and 6.93 nats) and compare the empirical estimates against the truth.

```
def infonce_bound(f):
    """log N minus the InfoNCE loss of an N x N score matrix f[i, j]."""
    return np.log(len(f)) + np.mean(np.diag(f) - logsumexp(f, axis=1))

def exact_scores(x, y, rho):
    """The exact critic T = pmi on all N x N pairs (x_i, y_j) of a batch."""
    return pmi_gauss(x[:, None], y[None, :], rho)

true_mi = np.array([0.25, 0.5, 1, 1.5, 2, 2.5, 3, 4, 5, 6, 7])
rhos = np.sqrt(1 - np.exp(-2 * true_mi))                # invert mi_gauss
Ns = (16, 128, 1024)
ests = np.array([[np.mean([infonce_bound(exact_scores(*gauss_pair(rng, r, N), r))
                           for _ in range(200 if N <= 128 else 50)])
                  for r in rhos] for N in Ns])             # (N, level)
```

```
ceilings = [[np.log(N)] * len(true_mi) for N in Ns]
d2l.plot(true_mi, [*ests, true_mi, *ceilings], 'true I(X;Y) (nats)',
         'InfoNCE estimate (nats)',
         legend=[f'N = {N}' for N in Ns] + ['truth'] + [f'ln {N}' for N in Ns],
         fmts=('C0-', 'C1-', 'C2-', 'k--', 'C0:', 'C1:', 'C2:'),
         figsize=(5.5, 3), legend_loc='outside')
for I in (1, 4, 7):
    k = list(true_mi).index(I)
    print(f'I = {I}:  ' + '   '.join(f'N = {N}: {e:.2f}'
                                      for N, e in zip(Ns, ests[:, k])))
```

```
I = 1:  N = 16: 0.89   N = 128: 0.98   N = 1024: 1.00
I = 4:  N = 16: 2.56   N = 128: 3.67   N = 1024: 3.95
I = 7:  N = 16: 2.76   N = 128: 4.76   N = 1024: 6.31
```



None of the curves has a sharp corner. Each peels away from the diagonal early and bends smoothly toward its ceiling, with all nine printed values falling within 0.1 nat of Equation 37.82. At $I = 1$, every batch size lands within about 0.1 nat of the truth. By $I = 4$, a batch of 128 reports only 3.7 nats, and even $N = 1024$ begins to lag slightly behind. At $I = 7$, just past its ceiling of 6.93, $N = 1024$ reports only 6.3.

Two exact facts explain why this happens. First, the average inside the logarithm is at least the positive's own term $e^{\text{pmi}(x,y^+)}/N$, so the slack is at least $\mathbb{E}[\text{pmi}(x, y^+)] - \log N = I - \log N$, recovering the $\log N$ ceiling. Second, the slack disappears only in the limit $N \to \infty$ (Exercise 5). The hard ceiling $\log N$ is merely the crudest symptom of this finite-batch slack.

This smooth compression also makes the estimate difficult to invert. With $N = 128$, a reading of 3.7 corresponds to 4 nats and a reading near 4.75 corresponds to 7 nats, squeezing every true value from 4 nats to infinity into the top 1.2 nats of the scale.

Image classification puts these numbers in perspective. Suppose two views of an image each determine its class among $1{,}000$ balanced categories. Because the label is a function of either view, the data-processing inequality (Section 37.6.2.2) implies that the two views share at least the entropy of the label, $\ln 1000 = 6.91$ nats. An InfoNCE loss with $N = 128$ candidates can never certify anything close to that, and even with $N = 1024$ the estimate

reads only 6.3 when the true shared information is 7. Whether two views retain enough information for fine-grained classification simply cannot be read off such a number.

37.7.4.4 Exact Critics Compared

We can now compare DV, NWJ, and InfoNCE on equal footing: batch size $N = 128$, each given the exact optimal critic $T^* = 1 + \text{pmi}$ (adding a constant leaves InfoNCE unchanged). DV and NWJ draw their independent pairs from the N mismatched pairs of the batch, and the heavy-tailed behavior we noted earlier lets us predict what will happen before running a single line of code.

A typical joint pair has pmi close to I. Under the product distribution $P_X \otimes P_Y$, such a pair occurs with probability roughly e^{-I}, since $p(x)\,p(y) = p(x,y)\,e^{-\text{pmi}}$. In other words, roughly one mismatched pair in e^I looks plausible enough to pass for a genuine pair. Each of these *rare pairs* carries a huge weight $e^{\text{pmi}} \approx e^I$, and together they account for the entire expectation $\mathbb{E}[e^{\text{pmi}}] = 1$. A batch of size N contains roughly Ne^{-I} such pairs on average: about six at $I = 3$, two at $I = 4$, and only 0.3 at $I = 6$.

As long as a batch contains several rare pairs, its sample average of e^{pmi} stays near 1 and all three bounds should agree. Once the expected count Ne^{-I} drops below 1, most batches contain no rare pairs at all. Call such a batch *starved*. On a starved batch, the average of e^{pmi} is close to zero: NWJ then reports the batch mean of T^* minus almost nothing (about $I + 1$, one nat too high), while DV subtracts the logarithm of a tiny number and overshoots by many nats.

Because NWJ is unbiased, something must balance those starved batches. That counterweight is a *hit* batch, one that happens to contain a mismatched pair with $e^{\text{pmi}} \geq N$. A single such pair adds at least 1 to the batch mean of e^{pmi} and drags NWJ down by at least a nat, leaving it near I or occasionally far below. At $I = 6$, a rare pair has $e^{\text{pmi}} \approx e^6 = 403 > N$, so every batch that contains one is a hit batch. With an average of 0.3 rare pairs per batch, roughly one batch in four should be hit, while most of the remaining batches are starved.

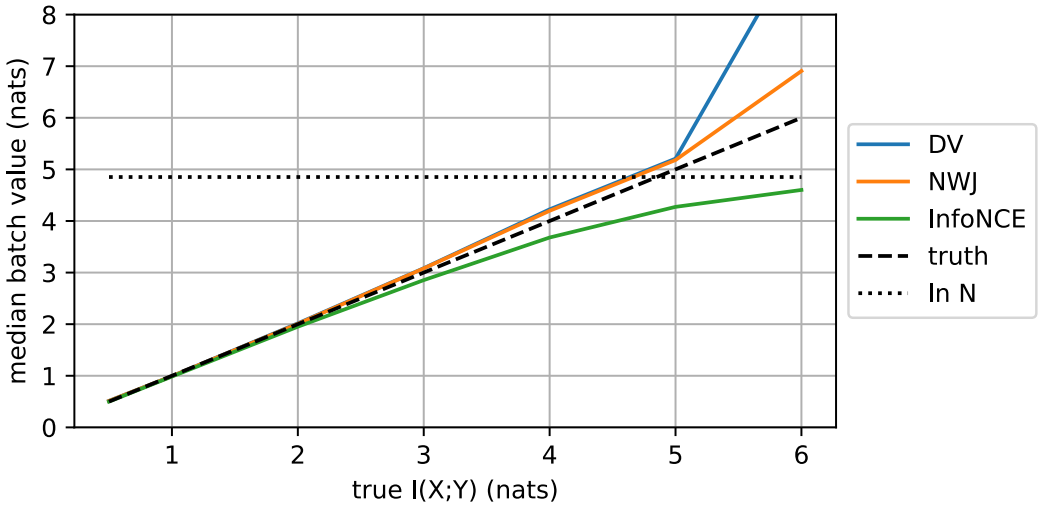
InfoNCE behaves completely differently. Because it normalizes within the batch, a convincing negative affects at most one row of the softmax. It should stay steady right where Equation 37.82 predicts, around 4.6 nats at $I = 6$. Notice that the crossover $Ne^{-I} \approx 1$ occurs at $I \approx \ln N$. The logarithm of the batch size limits DV and NWJ just as much as it limits InfoNCE; DV and NWJ simply hide that limit behind exploding variance instead of a visible ceiling.

Let us test these predictions by drawing 200 batches at each level of I. Because heavy-tailed distributions distort sample means and standard deviations, we report the median and interquartile range for each bound, along with the sample mean for NWJ. The final two columns track the fraction of starved batches (mean $e^{\text{pmi}} < 0.1$) and the fraction of hit batches (max $e^{\text{pmi}} \geq N$), and the plot compares the three medians.

```
def quartiles(a):
    """Median and quartiles of a list of batch values, as text."""
    q1, med, q3 = np.percentile(a, [25, 50, 75])
    return f'{med:5.2f} [{q1:5.2f}, {q3:5.2f}]'

N, true_mi = 128, np.array([0.5, 1, 2, 3, 4, 5, 6])
medians = []
```

```
# Last two columns: share of batches whose mean e^pmi over mismatched pairs
# is below 0.1, and share holding a mismatched pair with e^pmi >= N.
print(f'{"I":>3}  {"DV median [IQR]":20}  {"NWJ median [IQR]":20}  '
      f'{"NWJ mean":>8}  {"InfoNCE median [IQR]":20}  {"<0.1":>4}  {">=N":>4}')
for I in true_mi:
    rho, batches = np.sqrt(1 - np.exp(-2 * I)), []
    for _ in range(200):
        x, y = gauss_pair(rng, rho, N)
        T_joint = 1 + pmi_gauss(x, y, rho)              # T* on true pairs
        T_prod = 1 + pmi_gauss(x, np.roll(y, 1), rho)   # T* on mismatched
        e_pmi = np.exp(T_prod - 1)                      # e^pmi, mismatched
        batches.append((T_joint.mean() - (logsumexp(T_prod) - np.log(N)),  # DV
                        T_joint.mean() - e_pmi.mean(),                     # NWJ
                        infonce_bound(exact_scores(x, y, rho)),
                        e_pmi.mean() < 0.1, e_pmi.max() >= N))
    dv, nwj, nce, low, rare = np.array(batches).T
    medians.append([np.median(a) for a in (dv, nwj, nce)])
    print(f'{I:3.1f}  {quartiles(dv)}  {quartiles(nwj)}  {nwj.mean():8.2f}'
          f'  {quartiles(nce)}  {low.mean():4.0%}  {rare.mean():4.0%}')
d2l.plot(true_mi, [*np.array(medians).T, true_mi, [np.log(N)] * len(true_mi)],
         'true I(X;Y) (nats)', 'median batch value (nats)',
         legend=['DV', 'NWJ', 'InfoNCE', 'truth', 'ln N'], ylim=[0, 8],
         fmts=('C0-', 'C1-', 'C2-', 'k--', 'k:'), figsize=(5.5, 3),
         legend_loc='outside')
```

```
  I  DV median [IQR]       NWJ median [IQR]      NWJ mean  InfoNCE median [IQR]  <0.1   >=N
0.5   0.51 [ 0.43,  0.58]   0.51 [ 0.43,  0.58]      0.50   0.50 [ 0.45,  0.54]    0%    0%
1.0   0.99 [ 0.88,  1.13]   0.99 [ 0.88,  1.12]      0.99   0.99 [ 0.92,  1.04]    0%    0%
2.0   2.02 [ 1.81,  2.17]   2.01 [ 1.80,  2.16]      1.97   1.95 [ 1.90,  2.01]    0%    1%
3.0   3.08 [ 2.75,  3.47]   3.08 [ 2.70,  3.36]      3.02   2.85 [ 2.79,  2.92]    0%    2%
4.0   4.23 [ 3.74,  4.95]   4.20 [ 3.71,  4.57]      3.76   3.68 [ 3.62,  3.74]    8%    6%
5.0   5.21 [ 4.54,  8.28]   5.18 [ 4.44,  5.94]      4.83   4.27 [ 4.22,  4.31]   32%   36%
6.0   9.47 [ 5.33, 29.04]   6.91 [ 5.10,  7.00]      5.93   4.60 [ 4.58,  4.64]   57%   28%
```



Up to $I = 3$, all three medians stay within about 0.2 nat of the truth, almost no batch is

starved or hit, and InfoNCE displays the tightest spread. Beyond that point, starved batches appear right on schedule: at most about one in ten at $I = 4$, roughly a third at $I = 5$, and more than half at $I = 6$.

In the $I = 6$ row, NWJ's median climbs to about $7 = I+1$ with its upper quartile sitting right on top of it, meaning that half of all batches overshoot by roughly a full nat. The quarter or so of hit batches drag the sample mean back to within about half a nat of 6. DV fares even worse: its median sits several nats above the truth, and its upper quartile reaches into the tens, sending its curve right off the top of the plot. InfoNCE, by contrast, reads a rock-steady 4.6 nats with an interquartile range under 0.1.

We can also calculate NWJ's spread analytically. Its mismatched-pair term is a batch average of e^{pmi}, whose variance is $(e^{2I} - 1)/N$. At $I = 6$ and $N = 128$, that works out to a standard deviation of 36 nats, driven by extreme hit batches so rare that 200 draws rarely contain one (Exercise 1).

None of this contradicts the variational inequalities, which hold for population expectations. Rather, it shows how variational bounds reconcile with the distribution-free limit of Section 37.7.2.4. InfoNCE cannot exceed $\log N$ (with N at most the sample size), so it obeys the logarithmic limit by construction. DV and NWJ are free to report larger numbers, but whenever they do, their batch estimates cease to be reliable lower bounds.

37.7.4.5 Training a Critic

In real applications we do not know the true pmi, so the critic must be learned from data. When a contrastive loss reports less than the true mutual information, how do we tell whether the shortfall comes from an imperfect critic or from the slack of the bound itself? To answer this, let us train a dual-encoder critic of the exact form used by CLIP and SimCLR: two small networks $g, h : \mathbb{R} \to \mathbb{R}^{16}$ embed x and y, and the critic is their inner product $T(x, y) = g(x)^\top h(y)$. Computing all $N \times N$ scores for a batch then takes just one matrix multiplication of the two embedding matrices, reducing N^2 network evaluations to $2N$. We generate two disjoint pools of $4{,}096$ pairs at $\rho = 0.99$ (true information 1.96 nats), one for training and one for evaluation.

```
rho, n_pool = 0.99, 4096
x_tr, y_tr = (a.astype(np.float32) for a in gauss_pair(rng, rho, n_pool))
x_ev, y_ev = (a.astype(np.float32) for a in gauss_pair(rng, rho, n_pool))
```

Each encoder is a multilayer perceptron with two hidden layers of 64 ReLU units and Glorot-uniform initialization. On the resulting $N \times N$ score matrix, the InfoNCE loss is simply standard multi-class cross-entropy where the target class for row i is i, the index of its diagonal positive pair.

```
def encoder(d_out=16, hidden=64):
    """An MLP that embeds a scalar in d_out dimensions."""
    net = torch.nn.Sequential(torch.nn.Linear(1, hidden), torch.nn.ReLU(),
                              torch.nn.Linear(hidden, hidden), torch.nn.ReLU(),
                              torch.nn.Linear(hidden, d_out))
    for layer in net[::2]:                          # the Linear layers
        torch.nn.init.xavier_uniform_(layer.weight)
        torch.nn.init.zeros_(layer.bias)
```

```
    return net

def scores(g, h, x, y):
    """All N x N critic values T(x_i, y_j) = g(x_i) . h(y_j)."""
    return g(x[:, None]) @ h(y[:, None]).T

def infonce_loss(f):
    """Cross-entropy of each row's softmax at its positive, the diagonal."""
    return torch.nn.functional.cross_entropy(f, torch.arange(len(f)))

g, h = encoder(), encoder()
```

We train for $1{,}000$ Adam steps with learning rate 10^{-3} on minibatches of $N = 128$ pairs sampled from the training pool, then wrap the trained model as `critic_np` so it accepts NumPy arrays just like `exact_scores`.

```
opt = torch.optim.Adam([*g.parameters(), *h.parameters()], lr=1e-3)
N = 128
for step in range(1000):
    idx = rng.choice(n_pool, N, replace=False)
    loss = infonce_loss(scores(g, h, torch.tensor(x_tr[idx]),
                               torch.tensor(y_tr[idx])))
    opt.zero_grad()
    loss.backward()
    opt.step()

def critic_np(a, b):
    """The trained critic on NumPy inputs, as an N x N NumPy array."""
    with torch.inference_mode():
        return scores(g, h, torch.tensor(a), torch.tensor(b)).numpy()
```

To separate critic error from bound slack cleanly, we evaluate both the learned critic and the exact critic on the *same* held-out batches. Below, we partition the evaluation pool into disjoint batches of size $N \in \{2, 8, 32, 128\}$ and average the InfoNCE bound across them. Notice that a critic trained at $N = 128$ can be evaluated at any other batch size, because the optimal critic $\mathrm{pmi}(x, y) + c(x)$ does not depend on N.

```
def infonce_eval(score, x, y, N):
    """Mean InfoNCE bound over the disjoint batches of N pairs in (x, y)."""
    return np.mean([infonce_bound(score(x[i:i + N], y[i:i + N]))
                    for i in range(0, len(x), N)])

exact_np = lambda a, b: exact_scores(a, b, rho)   # the exact critic T = pmi

print('   N   ln N   learned   exact')
for N_ev in (2, 8, 32, 128):
    print(f'{N_ev:4d}  {np.log(N_ev):5.2f}  '
          f'{infonce_eval(critic_np, x_ev, y_ev, N_ev):7.2f}  '
          f'{infonce_eval(exact_np, x_ev, y_ev, N_ev):6.2f}')
print(f'true I(X;Y) = {mi_gauss(rho):.2f} nats')
```

```
   N   ln N   learned   exact
   2   0.69     0.58    0.58
   8   2.08     1.42    1.42
  32   3.47     1.78    1.80
 128   4.85     1.89    1.91
true I(X;Y) = 1.96 nats
```

Across every batch size, the learned critic matches the exact critic to within a few hundredths of a nat. The remaining gap to the true 1.96 nats therefore belongs entirely to the bound, closing gradually as N increases until both critics reach about 1.9 nats at $N = 128$. Look closely at $N = 8$: the ceiling $\ln 8 = 2.08$ already sits above the true value of 1.96, so the shortfall of roughly 0.55 nat is not caused by hitting the ceiling. It is the smooth-minimum slack of the bound, which Equation 37.82 predicts at about 0.6 nat. Of course, a scalar Gaussian pair gives the critic an easy task; in high dimensions, critic approximation error is no longer negligible, and without a known ground truth the two sources of error cannot be disentangled.

Contrastive systems such as SimCLR modify the critic slightly: they normalize the embeddings z_x and z_y to unit length and divide their cosine similarity by a *temperature* τ, setting $T(x, y) = \text{sim}(z_x, z_y)/\tau$ (Chen et al. 2020). The temperature controls how sharply the softmax separates candidates, and it introduces a second ceiling of its own. Because cosine similarity lies in $[-1, 1]$, no two scores can differ by more than $2/\tau$, so a cosine critic can never certify $2/\tau$ nats or more regardless of how large N is (Exercise 3): that caps the bound below 4 nats at $\tau = 0.5$ and below 20 nats at $\tau = 0.1$. A large temperature can therefore bottle up the bound well before the batch size does.

In practice, which estimator should we choose? For a pair of one- or two-dimensional variables sharing at most a few nats, nearest-neighbor counting is the right tool: KSG landed within about 0.1 nat of the truth in that regime. For high-dimensional vectors, none of the estimators in this section can accurately measure a large mutual information, so the question shifts from *how much* information two views share to *at least how much*. InfoNCE answers that question reliably: train it with as many negatives as memory allows and interpret $\log N - \mathcal{L}_{\text{NCE}}$ strictly as a lower bound. A reading within a nat or so of $\log N$ tells us only that the views share at least that much, while a low reading could simply reflect a weak critic. As for DV and NWJ, they have no sweet spot of their own: several nats below $\log N$, where their batch estimates remain trustworthy, InfoNCE is nearly as accurate and far steadier; higher up, their typical minibatch overshoots.

Finally, a tighter lower bound does not automatically produce a better representation. Tschannen et al. (2020) showed that downstream task performance correlates poorly with the tightness of the mutual information bound being maximized: looser bounds often yield stronger representations, and architectural choices for the encoder and critic matter at least as much as the estimated information. Two fundamental questions thus remain open: how to estimate large mutual information reliably in high dimensions, and why maximizing loose variational bounds still produces such effective representations.

37.7.5 Summary

From finite samples alone, mutual information can be measured directly only when the variables are low-dimensional and the shared information is modest. Beyond that regime, every estimator is constrained by the logarithm of the number of outcomes it can distinguish: $\log b$ for a histogram with b bins, roughly $\log(n/m)$ for KSG nearest-neighbor counts (which

saturate far below even that ceiling in high dimensions), and $O(\log n)$ for any distribution-free lower bound. Variational methods sidestep density estimation by maximizing a lower bound parameterized by a decoder or a critic. InfoNCE, the cross-entropy objective of contrastive learning, is low-variance and well behaved, yielding roughly $-\log(e^{-I}+1/N)$ with an exact critic and never exceeding $\log N$ for N candidates. Donsker–Varadhan and NWJ have no hard ceiling, yet their minibatch estimates remain reliable only a couple of nats below $\log N$. On our Gaussian benchmark, a learned dual-encoder critic matched the exact pmi critic to within a few hundredths of a nat, confirming that the observed gap belonged to the bound itself. Even so, the tightness of an information bound does not by itself predict the quality of a learned representation.

37.7.6 Exercises

1. **Heavy tails of the exponentiated pmi.** ● Let (X, Y) be the Gaussian pair with correlation ρ, $I = -\frac{1}{2}\log(1-\rho^2)$, and $Q = P_X \otimes P_Y$. Under Q the mean of e^{pmi} is 1 for every ρ, although a typical independent pair has e^{pmi} near 0 when ρ is close to 1.
   1. Show that $\mathbb{E}_Q[e^{2\,\text{pmi}}] = \int p^2/q = 1/(1-\rho^2) = e^{2I}$, so that e^{pmi} has variance $e^{2I}-1$ under Q. *Hint:* the integrand is proportional to a Gaussian density with precision matrix $2\Sigma^{-1}-\mathbf{I}$, where Σ is the covariance of the pair, and $\det(2\Sigma^{-1}-\mathbf{I}) = 1$.
   2. A random permutation of a batch leaves on average one y paired with its own x. Show that such a fixed point adds e^{pmi}/N to the batch mean of e^{pmi} over the permuted pairs, with expectation e^{2I}/N: 0.43, 23 and 1270 at $I = 2, 4, 6$ with $N = 128$. Why does `np.roll` avoid this?
   3. For the NWJ critic $T^* = 1 + \text{pmi}$, show that the batch estimate has mean I when the mismatched pairs are independent draws, and that its mismatched-pair term then has variance $(e^{2I}-1)/N$: standard deviations of 0.65, 4.8 and 36 nats at $I = 2, 4, 6$ with $N = 128$. Show that the rolled pairs (x_j, y_{j+1}) of the cell give the same variance for $N \geq 3$, although neighboring terms share a variable. *Hint:* condition on the shared pair.

2. **Barber–Agakov with a Gaussian decoder.** Take the standard bivariate Gaussian pair with correlation ρ, whose information is $-\frac{1}{2}\log(1-\rho^2)$, and the decoders $q(x \mid y) = \mathcal{N}(x; ay, s^2)$ with parameters $a \in \mathbb{R}$ and $s^2 > 0$. In the bound Equation 37.75, use the differential entropy $h(X) = \frac{1}{2}\log(2\pi e)$.
   1. Evaluate the right-hand side of Equation 37.75 as a function of a and s^2. Maximize it over both parameters and show that the maximum is the true information.
   2. The text fixes $s^2 = 1$, which is what training with the squared error alone amounts to, and finds the best bound $\rho^2/2$. Show that its gap to the true information is $D_{\text{KL}}(\mathcal{N}(0, 1-\rho^2) \,\|\, \mathcal{N}(0,1))$, as the proof of the proposition says it must be. How does the gap behave as $\rho \to 1$?
   3. Let $\hat{x}(y)$ be any predictor of X with mean squared error D. Use the decoder $\mathcal{N}(x; \hat{x}(y), D)$ to show that $I(X;Y) \geq -\frac{1}{2}\log D$. What is the corresponding bound when $X \sim \mathcal{N}(0, v)$? Compare it with the Gaussian rate–distortion function $R(D) = \frac{1}{2}\log(v/D)$ for $D < v$ of Section 37.6.3.2, the least information any description of X needs to reach squared error D.

3. **Temperature of a cosine critic.** SimCLR-style systems score candidates with $T(x,y) = s(x,y)/\tau$, where $s(x,y) = \mathrm{sim}(z_x, z_y) \in [-1,1]$ is a cosine similarity and $\tau > 0$ is the temperature. For one anchor x, write $s_j = s(x,y_j)$, with $y_1 = y^+$ the positive, and let ℓ be the loss inside the expectation Equation 37.78.

   1. Show that $\mathcal{L}_{\mathrm{NCE}} \geq \log\big(1 + (N-1)e^{-2/\tau}\big)$, and hence $\hat{I}_{\mathrm{NCE}} < 2/\tau$ for every N. Evaluate the ceiling $\log N - \log\big(1 + (N-1)e^{-2/\tau}\big)$ for $N = 4096$ and $\tau \in \{1, 0.5, 0.1\}$ (you should find about 2.0, 4.0 and 8.3 nats). When does the temperature rather than the batch size limit the bound?
   2. Show that $\partial\ell/\partial s_j = p_j/\tau$ for every negative $j \geq 2$, where p_j is the softmax probability of candidate j. Express $\partial\ell/\partial s_1$ through the p_j.
   3. Show that $\tau\ell \to \max\{0, \max_{j\geq 2} s_j - s_1\}$ as $\tau \to 0$, and that

$$\ell = \log N - \frac{1}{\tau}\Big(s_1 - \frac{1}{N}\sum_{j=1}^{N} s_j\Big) + O(\tau^{-2}) \quad \text{as } \tau \to \infty. \tag{37.83}$$

   Which negatives drive the gradient in each limit (Wang and Liu 2021)?

4. **Entropy bounds from samples.** ● Section 37.7.2.4 states the theorem of McAllester and Stratos (2020) informally and proves a version of it for entropy; this problem sharpens and applies that proof. Draw a sample S of n i.i.d. values from a discrete distribution p. The *type* of S records, for each $i \geq 1$, how many distinct values occur exactly i times in S; it forgets which values these are. Let B be a function of the type that is a lower confidence bound for every discrete distribution: $P_{S\sim p^n}\big(B \leq H(p)\big) \geq 0.95$ for every p. Let $\mathcal{T}_0$ be the type of a sample of n distinct values.

   1. The text applies the guarantee to the uniform distribution on n^2 values. Show that any integer $M \geq n^2/1.9$ serves as well, so that $B(\mathcal{T}_0) \leq \log M$. By how much does this improve on $2\log n$?
   2. Let p be uniform on $M = 2^{256}$ values. How large must n be before $2\log n$ reaches $H(p)$? What does this say about measuring the entropy of a cryptographic key from samples?

5. **Optimal InfoNCE critic.** ● For one anchor x and one bag $v = (y_1, \dots, y_N)$ with $y_1 = y^+$, write $\ell(x,v) = -\log\big(e^{T(x,y_1)}/\sum_{j=1}^{N} e^{T(x,y_j)}\big)$, so that $\mathcal{L}_{\mathrm{NCE}}$ of Equation 37.78 is its expectation.

   1. Show that replacing $T(x,y)$ by $T(x,y) + c(x)$, for any function c, leaves $\mathcal{L}_{\mathrm{NCE}}$ unchanged.
   2. Shuffle the bag so that the positive sits at a uniformly random position J and the other $N-1$ positions hold the negatives. Show that

$$P(J = i \mid x, y_1, \dots, y_N) = \frac{e^{\mathrm{pmi}(x,y_i)}}{\sum_{j=1}^{N} e^{\mathrm{pmi}(x,y_j)}}. \tag{37.84}$$

   Use Gibbs' inequality to conclude that every critic $T(x,y) = \mathrm{pmi}(x,y) + c(x)$ minimizes $\mathcal{L}_{\mathrm{NCE}}$.

3. For this critic, show that

$$\hat{I}_{\text{NCE}} = I(X;Y) - \mathbb{E}\left[\log \frac{1}{N}\sum_{j=1}^{N} e^{\text{pmi}(x,y_j)}\right]. \tag{37.85}$$

Use the law of large numbers to argue that the correction tends to zero as $N \to \infty$ when $I(X;Y)$ is finite. Evaluate the correction by Monte Carlo at $\rho = 0.99$ and $N = 8$, and compare it with the gap between the exact-critic value (about 1.4) and the truth (1.96) in Section 37.7.4.5.

4. The approximation Equation 37.82 replaces the negatives' share of the average by 1 and $\text{pmi}(x, y^+)$ by I. Use Jensen's inequality to show that the first replacement lowers the estimate and the second raises it. Which of the two wins in the table of Section 37.7.4.3?

6. **In-batch negatives and the symmetric loss.** The cells of Section 37.7.4.5 score all $N \times N$ pairs of a batch of N independent pairs (x_i, y_i) and use row i's off-diagonal entries as the negatives of anchor x_i.

   1. Show that each row satisfies the hypotheses of the InfoNCE proposition: its candidates y_j, $j \neq i$, are draws from P_Y that are independent of (x_i, y_i). Conclude that the mean row loss $\mathcal{L}$ certifies $I(X;Y) \geq \log N - \mathcal{L}$ in expectation.
   2. The rows share candidates, so their losses are dependent. Does this affect the validity of the bound, the expectation of the mean row loss, or its variance?
   3. CLIP uses the symmetric loss, the mean of the row-wise and the column-wise cross-entropies. Show that it bounds $I(X;Y)$ in the same way, with the same ceiling $\log N$.
   4. A joint critic $T(x, y)$, a network applied to the concatenated pair, needs N^2 network evaluations for the scores of a batch, where the separable critic $g(x)^\top h(y)$ needs $2N$. Evaluate both counts for CLIP's batches of $N = 32{,}768$. A separable critic cannot represent every function of the pair. Show that the pmi of the Gaussian pair is separable with embeddings of three dimensions, and that two suffice for InfoNCE.

7. **Negative sampling as shifted PMI.** ● Skip-gram with negative sampling, the training method of word2vec, scores a center word w and a context word o by the inner product $s(w, o) = \mathbf{u}_o^\top \mathbf{v}_w$ of their vectors. It learns to tell the pairs that occur in a corpus from pairs whose context word is one of K noise words drawn from a distribution P. Let $p(w, o)$ be the frequency of the pair among all (center, context) pairs of the corpus, with marginals $p(w)$ and $p(o)$. Averaged over the corpus, the loss is

$$\ell = -\sum_{w,o} p(w,o)\log\sigma(s(w,o)) - K\sum_{w} p(w)\sum_{o} P(o)\log\sigma(-s(w,o)), \tag{37.86}$$

where $\sigma(s) = 1/(1+e^{-s})$ is the sigmoid. Assume that the embeddings have enough dimensions for every score $s(w, o)$ to be chosen independently.

   1. Show that ℓ is minimized at $s^*(w,o) = \log \frac{p(w,o)}{p(w)\,P(o)} - \log K$, which equals $\text{pmi}(w,o) - \log K$ when P is the unigram distribution $p(o)$ (Levy and Goldberg 2014).

2. word2vec draws noise words with probability proportional to their count raised to the power 0.75. Show that s^* then differs from $\mathrm{pmi}(w, o)$—$\log K$ by a term that depends on o alone. For which context words is this term negative? How does it act on pairs of rare words, such as the "sri lanka" of Exercise 1 of Section 37.6?
3. Compare s^* with the optimal InfoNCE critic of Exercise 5b. Why does the binary classification of negative sampling fix the additive term that InfoNCE leaves free?

8. [extended] **Compression phase at small scale.** ●● Shwartz-Ziv and Tishby (2017) studied fully connected tanh networks of widths 12-10-7-5-4-3-2 on the 4096 binary input patterns of 12 bits. For each hidden layer Z they binned every unit's activation into 30 equal intervals of $[-1, 1]$ and computed $I(X; Z)$ and $I(Y; Z)$ from the binned values. Section 37.7.2.5 describes the debate about the compression phase they reported.

   1. Let X be uniform on the 4096 patterns and let Z_{bin} be the binned layer of a deterministic network. Show that the binned estimate of $I(X; Z)$ equals $H(Z_{\text{bin}})$ and never exceeds $\ln 4096 \approx 8.32$ nats. When does it decrease during training?
   2. Fix a random vector $\mathbf{w} \in \mathbb{R}^{12}$ and let $Y = 1$ when $\mathbf{w}^\top \mathbf{x}$ exceeds its median over the 4096 patterns. Train the tanh network, with a softmax over its two output units, by SGD with the cross-entropy loss on a random 85% of the patterns. Plot the trajectory of every hidden layer in the information plane $(I(X; Z), I(Y; Z))$ over training.
   3. Repeat (b) with ReLU hidden units, binning each unit into 30 equal intervals between 0 and its largest activation over the run.
   4. Record the fraction of tanh units with $|z| > 0.99$ during training. Does the decrease of the binned $I(X; Z)$ in (b) track this fraction? Which conclusions about compression survive the change of activation in (c) (Saxe et al. 2018)?

37.8 Further Reading

The main sections of this chapter stick to results that we can prove cleanly and check on small examples. Each short note below picks up where a section left off, covering neighboring ideas that did not make the core path. If you are short on time, start with the three topics that show up most directly in everyday practice: model calibration, the irreducible-loss constant in scaling laws, and the bias–variance trade-off of mutual-information estimators. Bits-back coding is also well worth reading simply because the trick is so clever. Channel capacity and information geometry can easily wait, and nothing later in the book depends on this page.

37.8.1 Entropy and Losses

37.8.1.1 Calibration after Training

In principle, minimizing the population log-loss recovers the true conditional class probabilities (Section 37.1.3.1), which are automatically calibrated. Trained neural networks, however, rarely sit at that optimum. As C. Guo et al. (2017) famously observed, modern networks tend to err in one particular direction: they are overconfident. One-hot training labels are a major culprit. Because they reward pushing the winning logit infinitely far ahead of the rest (Section 37.1), once a network classifies the training set perfectly, the only way it can keep lowering the loss is to push its logits further and further apart. Late in training, the same authors see held-out log-loss begin to climb even while held-out classification

error stays flat: the network is overfitting its probability estimates rather than its decisions. Temperature scaling (Section 3.4) fixes much of this with a single scalar fitted on a held-out validation set. It is the same temperature parameter used in distillation, repurposed here to soften overconfident predictions.

How far can a post-hoc recalibration like this go? Let $\bar{\mathbf{p}}(\mathbf{q})$ denote the actual empirical class frequencies among all inputs for which the model outputs the probability vector $\mathbf{q}$. Since any cross-entropy decomposes into an entropy plus a KL divergence, applying that identity conditionally at each predicted vector $\mathbf{q}$ splits the held-out log-loss into two pieces:

$$\mathbb{E}\big[-\log q_y\big] = \mathbb{E}_{\mathbf{q}}\big[D_{\mathrm{KL}}(\bar{\mathbf{p}}(\mathbf{q}) \,\|\, \mathbf{q})\big] + \mathbb{E}_{\mathbf{q}}\big[H(\bar{\mathbf{p}}(\mathbf{q}))\big].$$

The first term measures miscalibration in nats: it vanishes if and only if events actually happen as often as the model's predicted probabilities claim. The second term is the uncertainty that still remains about the true label even after we know the model's prediction. Any post-hoc remapping of the predicted probabilities can drive the first term toward zero, but it can never reduce the second. Temperature scaling is simply a one-parameter family of such remappings, which makes it wonderfully simple, though a single scalar cannot fix a model that is overconfident on some classes and underconfident on others. More generally, every proper scoring rule admits an analogous decomposition into a divergence term and an entropy term (Gneiting and Raftery 2007).

37.8.1.2 Entropy Relative to a Reference

In Section 37.1.4.1, we saw an awkward quirk of differential entropy: switching units from meters to centimeters shifts it up by $\ln 100$. A KL divergence, by contrast, does not budge, because both densities inside the ratio rescale by the same factor. We can give entropy the same coordinate invariance by measuring it against a reference density m, defining $-\int p \log(p/m)\, dx = -D_{\mathrm{KL}}(p\|m)$. Ordinary differential entropy h is just the special case $m = 1$ in whatever units we happened to pick. Because a uniform density of one per meter is not the same measure as one per centimeter, that $\ln 100$ shift was really a hidden change of reference. Jaynes (1968) derives this relative form as the limit of discrete entropies on increasingly fine grids; with it, the maximum-entropy principle takes the natural form "stay as close to the reference measure as the constraints allow." Up to a sign, the KL penalty in a variational autoencoder (Section 37.6.4.3) is an entropy of this kind with the prior playing the role of m, which is why it remains invariant under smooth reparameterizations of the latent space.

37.8.2 Codes and Compression

37.8.2.1 Compression without a Model

Every compressor in Section 37.2 relies on an explicit probabilistic model to predict the next symbol. The Lempel–Ziv family does away with that requirement altogether. LZ77 (Ziv and Lempel 1977) replaces any repeated substring with a short pointer back to an earlier occurrence, while LZ78 (Ziv and Lempel 1978) builds up a dictionary of phrases on the fly as it scans the file. Remarkably, both algorithms are *universal*: for any stationary ergodic source (roughly, one whose long-run statistics are revealed by a single long sample), their average code length approaches the true entropy rate as the file grows, even though nobody tells them anything about the source in advance. In practice, however, convergence to that limit is slow. On *The Time Machine*, `zlib` (which pairs LZ77 with Huffman coding) uses 2.84

bits per character, whereas the simple counting model of Section 37.2.2.2, which predicts each character from the three characters before it, already gets down to 2.55.

37.8.2.2 Bits-Back Coding

A latent-variable model such as a variational autoencoder cannot directly drive the arithmetic coder of Section 37.2.2.1. The coder needs the exact marginal probability $p(x)$, whereas the model only gives us the ELBO (Section 36.7), a lower bound on $\log p(x)$ built from an approximate posterior $q(z \mid x)$. Surprisingly, that lower bound still corresponds to an achievable file length, thanks to a clever duality: decoding a compressed stream is mathematically the same as sampling. If you feed random bits into a decoder designed for $q(z \mid x)$, it consumes roughly $\log_2 1/q(z \mid x)$ bits and outputs a latent sample z drawn from q.

Bits-back coding turns this observation into a practical scheme. Suppose the sender already has a compressed file holding earlier data points; because good compression removes redundancy, the bits at the end of that file look indistinguishable from random coin flips. To encode a new data point x, the sender borrows $\log_2 1/q(z \mid x)$ bits from the end of the file and "decodes" them with $q(z \mid x)$ to draw a latent z. The sender then encodes x using $p(x \mid z)$ and z using the prior $p(z)$, appending both to the file. To unpack x, the receiver reverses the process: it decodes z from the prior, decodes x from $p(x \mid z)$, and then *re-encodes* z using $q(z \mid x)$, which puts the borrowed bits right back where they came from. Overall, the file grew by $\log_2 1/p(x \mid z) + \log_2 1/p(z)$ bits and shrank by $\log_2 1/q(z \mid x)$ bits, so its expected net growth under q is the negative ELBO. Because the receiver must undo the sender's operations in last-in, first-out order, the underlying coder has to behave like a stack. The bits-back argument itself goes back to Wallace (1990) and Hinton and Camp (1993), and Frey and Hinton (1997) built the first working coder from it on top of traditional arithmetic coding, which behaves like a queue instead. Asymmetric numeral systems (Duda 2013) are naturally stack-based, and using them Townsend et al. (2019) demonstrated compressing MNIST with a small variational autoencoder at rates right near its negative ELBO.

37.8.2.3 Description Length beyond Two-Part Codes

The two-part code of Section 37.2.2.3 encodes the model parameters explicitly at some chosen precision, which makes the total description length sensitive to how finely we discretize the parameter space. A *prequential* code avoids sending any parameters at all. Instead, it encodes each data point using the model fitted to the points seen so far, just like the adaptive counting model in Section 37.2.2.2, so the cost of learning the model is absorbed directly into the data code length. Blier and Ollivier (2018) used prequential coding to compress the training labels of deep neural networks and obtained far shorter codes than either two-part codes or variational codes (which apply bits-back coding to the network weights). Even a heavily overparameterized network compresses its labels when measured this way. Grünwald (2007) develops both prequential codes and normalized maximum likelihood, the code that minimizes worst-case regret relative to the best model in the class chosen in hindsight. Pushing this compression viewpoint to its theoretical extreme leads to Kolmogorov complexity, the length of the shortest computer program that outputs the data, which unifies all of these ideas, though it is famously uncomputable (Li and Vitányi 2019).

37.8.2.4 The Constant of a Scaling Law

In Section 11.6, the held-out loss of language models across many scales is modeled as a constant E plus two power-law terms, one decaying with the number of parameters and the other with the number of training tokens (Hoffmann et al. 2022). It is tempting to read E as the true entropy of natural text, and in the best case E does correspond to the entropy rate per token (Section 37.2.3.1), the floor below which no expected loss can fall. In practice, however, E is only a fitted regression constant extrapolated from losses that all sit well above it, not a direct measurement of the entropy rate. Earlier work by Kaplan et al. (2020) fit similar scaling curves with no additive constant at all, an approximation that must eventually break down at large enough scale because the loss cannot fall below the entropy floor. Keep in mind, too, that E is measured per token: constants fitted under different tokenizers cannot be compared directly until both are converted to bits per byte (Section 8.3.2.4).

37.8.3 Comparing Distributions

37.8.3.1 Rényi Divergences and Privacy

How much does a randomized algorithm leak about a single training record when we run it repeatedly? Differential privacy formalizes this leak as a divergence between the algorithm's output distributions on two datasets that differ in just that one record. Because differentially private SGD injects noise at every gradient step and runs for thousands of iterations, we need a divergence that composes cleanly across independent steps. The Rényi divergence of order α (Rényi 1961),

$$D_\alpha(P\|Q) = \frac{1}{\alpha - 1} \log \sum_x p(x)^\alpha q(x)^{1-\alpha},$$

has this property. Notice that the logarithm sits *outside* the sum, whereas an f-divergence applies its convex function inside. Because the sum factorizes over independent coordinates, the logarithm turns that product into a sum, so D_α adds cleanly over independent steps, just like KL divergence does and unlike χ^2 or total variation distance. Rényi differential privacy (Mironov 2017) bounds D_α for a single noisy step and then sums that bound over the entire training run. While KL divergence also adds across steps, it only controls the log-density ratio $\log(p(x)/q(x))$ on average. Because D_α increases monotonically with α, interpolating from KL divergence as $\alpha \to 1$ to the worst-case log-ratio as $\alpha \to \infty$, choosing a larger order α protects against rare, high-leakage outputs as well. Erven and Harremoës (2014) provide a thorough catalog of these limits and inequalities, and Exercise 1d of Section 37.3 connects D_α to the α-divergences.

37.8.3.2 Information Geometry

In Section 37.3, a quick Taylor expansion showed that every smooth f-divergence between $q_{\theta+\mathbf{h}}$ and q_θ reduces, to second order, to the quadratic form $\frac{1}{2} f''(1)\, \mathbf{h}^\top \mathbf{I}(\theta)\, \mathbf{h}$ in the Fisher information matrix. That might look like an algebraic coincidence, yet Chentsov's theorem shows that it could not have turned out any other way. A positive-definite quadratic form assigns a length to every infinitesimal step $\mathbf{h}$, turning the space of distributions into a Riemannian manifold (measuring local steps, whereas the metrics of Section 37.3.1.1 measure global distances between two distributions). For distributions on finite sets, Chentsov proved that the Fisher information is, up to a positive constant, the *only* Riemannian metric that remains invariant when the raw data are replaced by a sufficient statistic (Čencov

1982). Every f-divergence shares this invariance: since the data can be resampled from a sufficient statistic without knowing θ, the data-processing inequality of Section 37.3.3.3 applies in both directions, forcing equality. Consequently, the local quadratic approximation of any smooth f-divergence must be a multiple of the Fisher information. Taking gradient steps whose length is measured in this geometry yields the natural gradient, which K-FAC approximates with one Kronecker product per layer (Section 35.5.5), and the KL trust region of Section 15.2 is, to second order, a ball in this very metric. Amari (2016) develops this geometric perspective in full, showing how maximum likelihood and both steps of the EM algorithm (Section 36.7) can be understood as orthogonal projections.

37.8.3.3 Sinkhorn Divergences and Sliced Wasserstein

Turning an optimal transport distance into a practical loss on high-dimensional mini-batches runs into two obstacles: computational cost and statistical bias. Entropic regularization (Section 37.4.2.4) solves the computational problem, yet it introduces a subtle bias: the regularized transport cost from a distribution to itself is strictly nonzero, so a model trained to minimize it tends to shrink toward a narrower distribution than the data. The Sinkhorn divergence of Exercise 5 in Section 37.4 fixes this by subtracting the average of the two self-transport costs. Beyond that exercise lies the theorem that makes it a trustworthy training objective: on a bounded domain, using the Euclidean distance or its square as the ground cost, the Sinkhorn divergence is nonnegative, equals zero if and only if the two distributions agree, and converges to zero if and only if the model converges in distribution to the data (Feydy et al. 2019).

The *sliced* Wasserstein distance takes a completely different route. If we project two high-dimensional samples onto a single one-dimensional line, optimal transport reduces to simply sorting the projected values (Section 37.4.2). Averaging this one-dimensional cost over many random directions produces the sliced Wasserstein distance (Bonneel et al. 2015). The catch in high dimensions is that a random direction rarely aligns with the subspace where two distributions actually differ. To get around this, Deshpande et al. (2018) train a discriminator to extract informative features and compute the sliced Wasserstein distance in that learned feature space instead of raw pixels.

37.8.3.4 Lipschitz Critics in Wasserstein GANs

The Kantorovich–Rubinstein dual in Section 37.4.2 optimizes over 1-Lipschitz functions, a constraint that an ordinary neural network does not satisfy out of the box. Section 16.2 compares the three main ways to enforce it (weight clipping, spectral normalization, and gradient penalties), while Section 16.4 examines gradient penalties in detail. In particular, the penalty of Gulrajani et al. (2017) nudges the critic's gradient norm toward one at random points interpolated between real and generated samples, reflecting the geometric fact that an optimal critic has unit slope along the transport rays connecting matched points (Exercise 3d of Section 37.4).

37.8.3.5 Stein Variational Gradient Descent

Drawing samples from a distribution Q known only through its score s_Q ordinarily requires running a long stochastic random walk (Section 37.5). In Section 37.5.3, we saw that the kernel Stein discrepancy measures how far a sample distribution P sits from such a model Q. Stein variational gradient descent (Liu and Wang 2016) turns that discrepancy into a deterministic algorithm that actively transports a set of particles toward Q. Starting from

any initial cloud of points with distribution P, each particle y moves in small steps along the velocity field

$$v(y) = \mathbb{E}_{x\sim P}\big[k(x,y)\, s_Q(x) + \nabla_x k(x,y)\big].$$

The two terms inside the expectation play complementary roles. The first is a kernel-weighted average of the scores near y, which steers particles uphill toward regions of higher probability density. The second is a repulsive force that pushes y away from its neighbors, taking over the role of random noise in Langevin dynamics; without it, every particle would collapse onto a single mode. When normalized to the unit ball of the reproducing kernel Hilbert space from Section 37.5.3, this velocity field is the steepest descent direction for $D_{\mathrm{KL}}(P\|Q)$, reducing the KL divergence at rate $\|v\| = \mathrm{KSD}(P,Q)$. Remarkably, the same Stein operator serves both as a goodness-of-fit test and as a particle transport rule. Although the particles still require many small steps, the updates are deterministic and the particles spread out cooperatively, which makes the method well suited to approximating a posterior with only a few dozen particles, such as the weights of a small neural network in the original paper.

37.8.4 Mutual Information

37.8.4.1 Codes That Approach Capacity

Shannon's channel coding theorem (Section 37.6.3.1) guarantees that reliable communication is possible at every rate below channel capacity, but his proof relies on random codebooks that take exponential time to decode. Bridging the gap between that existence theorem and practical hardware took decades. Low-density parity-check (LDPC) codes, decoded efficiently by passing messages along a sparse factor graph, come remarkably close to capacity in practice (Richardson and Urbanke 2008). Polar codes (Arıkan 2009) go a step further: they are the first explicit code family provably shown to achieve capacity on symmetric binary-input channels, with both encoding and decoding running in $O(n \log n)$ operations for block length n. Modern 5G wireless standards put both families to work.

37.8.4.2 Learned Lossy Compression

In Section 37.6.3.2, we derived the exact rate–distortion curve for a Gaussian source. For natural photographs, no closed-form rate–distortion function is known, so any learned image codec represents a practical operating point somewhere above that unknown frontier. Ballé et al. (2017) train three components jointly: an encoder, a decoder, and an *entropy model* that defines a learned probability distribution over the quantized latent representation. Because rounding has zero gradient almost everywhere, they replace quantization with additive uniform noise during training. Their objective minimizes $R + \lambda D$, where the rate R is the arithmetic-coding length of the latent under the entropy model (Section 37.2) and D is the reconstruction distortion of the decoded image; varying λ traces out the codec's rate–distortion curve by selecting the point with slope $-\lambda$. In follow-up work, Ballé et al. (2018) condition the entropy model on a second, smaller latent called a *hyperprior*, which captures spatial variations in image complexity and substantially lowers the bit rate at the same visual quality.

37.8.5 Estimating Mutual Information

37.8.5.1 Upper Bounds

Every variational estimator developed in Section 37.7 is a *lower* bound on mutual information. That is useful when you want to maximize shared information, but it cannot certify that a learned representation keeps *little* information about a sensitive or protected attribute it is supposed to discard. Obtaining a genuine upper bound requires access to the conditional density $p(z \mid x)$, which a stochastic encoder provides. The variational information bottleneck (Section 37.6.4.3) is one such upper bound, using a fixed marginal approximation $r(z)$. The contrastive log-ratio upper bound, or CLUB (Cheng et al. 2020), removes the need for $r(z)$ altogether: it averages $\log p(z \mid x)$ over matched pairs in a minibatch and subtracts the average over mismatched pairs. When $p(z \mid x)$ is the true encoder density, CLUB is an upper bound in expectation. Between a representation and a protected attribute, however, no analytic conditional density is available, so one must fit a neural network in its place. The upper bound then holds only if the fitted conditional is closer in KL divergence to the true joint distribution than to the product of the marginals, a condition that cannot be verified without the very density we were missing in the first place.

37.8.5.2 Bias and Variance of the Bounds

Even when given the exact optimal critic, variational bounds face a sharp trade-off: NWJ is unbiased on average but suffers from high variance, whereas InfoNCE is low-variance but capped at $\log N$ (Section 37.7.4.4). Poole et al. (2019) introduce a single mixing parameter that smoothly interpolates between the two bounds, letting you dial along this trade-off. Worse still, J. Song and Ermon (2020) prove that the variance explosion is fundamental rather than an artifact of any one objective: for both NWJ and Donsker–Varadhan bounds, the estimator variance grows exponentially with the true mutual information, even with an exact critic. Their proposed fix clips the critic's density ratio, which tames the variance at the cost of introducing bias. In short, every available remedy shifts error between variance and bias rather than eliminating it.

Because ground-truth mutual information is rarely known on real data, the same authors suggest self-consistency sanity checks that require no known answer: a reliable estimator should remain unchanged when a deterministic function of one variable is appended to that variable, and it should double when a single pair (X, Y) is replaced by two independent copies of itself. On image datasets, the standard estimators they evaluate fail these basic tests. As a practical rule of thumb, always compare a reported mutual information estimate of several nats against the logarithm of the batch size, and trust an estimator on your own data only after verifying that it passes these consistency checks there.

37.8.5.3 Noisy Networks and the Bottleneck View

The compression debate of Section 37.7.2.5 began when Tishby and Zaslavsky (2015) proposed plotting each layer of a neural network against the information bottleneck frontier (Section 37.6.4), which tracks the maximum $I(Y; Z)$ achievable for a given $I(X; Z)$. As we saw in that subsection, however, the idea runs into a mathematical snag: in a deterministic network fed continuous inputs, $I(X; Z)$ is formally infinite, leaving no finite compression curve to track. Goldfeld et al. (2019) make the question well posed by injecting small Gaussian noise into the hidden activations, which renders $I(X; Z)$ finite. Under this noisy model, $I(X; Z)$ can indeed decrease during training, and the authors trace that drop to geometric clustering as hidden representations from the same class pull closer together. This

clustering, they argue, is what earlier binned mutual-information estimates on deterministic networks were really picking up all along.

37.8.6 General Information Theory Resources

Books

- Elements of Information Theory — Cover & Thomas: the classic graduate reference that develops entropy, KL divergence (here called relative entropy), mutual information, and their chain rules from first principles, alongside source coding, channel capacity, and rate–distortion theory; accompanies Section 37.1, Section 37.2, and Section 37.6.
- Information Theory, Inference, and Learning Algorithms — David MacKay: freely available online; weaves information theory and machine learning together throughout, with chapters on symbol codes and stream codes (including arithmetic coding) that accompany Section 37.2.
- Information Theory: From Coding to Learning — Polyanskiy & Wu: a modern graduate text geared toward statistics and machine learning, whose treatment of f-divergences, variational representations, and data processing accompanies Section 37.3 and Section 37.6; the MIT 6.441 lecture notes from which the book grew are freely available.
- Computational Optimal Transport — Peyré & Cuturi: covers Wasserstein distances, entropic regularization, and Sinkhorn iterations for data science, supporting the transport half of Section 37.4; the full text is also on arXiv.
- The Minimum Description Length Principle — Peter Grünwald: develops statistical learning as data compression through two-part and universal codes, expanding on Section 37.2.2.3; the core ideas are also presented in a freely available tutorial.

Courses, tutorials, and software

- Stanford EE376A Information Theory — Tsachy Weissman: an accessible course with public, student-scribed lecture notes covering entropy, prefix codes, and typical sets, accompanying Section 37.1 and Section 37.2.
- Visual Information Theory — Christopher Olah: an intuitive visual introduction that builds from codeword lengths to entropy, cross-entropy, and mutual information; accompanies Section 37.1, Section 37.2, and Section 37.6.
- From GAN to WGAN — Lilian Weng: a clear walkthrough comparing the KL, Jensen–Shannon, and Wasserstein losses for generative models, paralleling Section 37.3 and Section 37.4.
- POT: Python Optimal Transport — Flamary et al.: an open-source Python library providing solvers for both exact optimal transport and entropically regularized Sinkhorn iterations, useful for computing the distances of Section 37.4.2 on your own datasets.
- Generative Modeling by Estimating Gradients of the Data Distribution — Yang Song: an accessible tutorial explaining why score functions avoid normalizing constants and how score matching fits them to data (covering Section 37.5) before moving on to sampling and diffusion models.

Foundational papers

- Language Modeling Is Compression — Delétang et al.: pairs an arithmetic coder with a language model's next-token probabilities, the exact idea demonstrated in Section 37.2.2.2.

- f-GAN: Training Generative Neural Samplers using Variational Divergence Minimization — Nowozin et al.: shows that any f-divergence yields an adversarial training objective via its Fenchel conjugate lower bound, as derived in Section 37.3.4.
- A Kernel Two-Sample Test — Gretton et al.: introduces the maximum mean discrepancy, its unbiased two-sample estimator, and the resulting hypothesis test from Section 37.4.1.
- Wasserstein GAN — Arjovsky et al.: replaces the Jensen–Shannon divergence with the Wasserstein-1 distance so that gradients remain informative even when supports do not overlap; accompanies Section 37.4.2.
- Estimation of Non-Normalized Statistical Models by Score Matching — Aapo Hyvärinen: fits unnormalized models by matching the gradient of the log-density to that of the data and eliminating the unknown data score via integration by parts, the two key steps of Section 37.5.
- A Kernelized Stein Discrepancy for Goodness-of-fit Tests — Liu, Lee & Jordan: tests whether a sample comes from a model known only through its score using pairwise kernel evaluations, as presented in Section 37.5.3.
- The Information Bottleneck Method — Tishby, Pereira & Bialek: formulates the problem of compressing X while preserving information about Y, the framework studied in Section 37.6.4.
- Estimating Mutual Information — Kraskov, Stögbauer & Grassberger: estimates the mutual information of continuous variables from nearest-neighbor distances, the estimator of Section 37.7.2.2 on which scikit-learn's `mutual_info_regression` is built.
- On Variational Bounds of Mutual Information — Poole et al.: unifies the Barber–Agakov, Donsker–Varadhan, NWJ, and InfoNCE bounds in a single framework and analyzes their bias–variance trade-offs, accompanying Section 37.7.3.
- Representation Learning with Contrastive Predictive Coding — van den Oord et al.: introduces the InfoNCE loss as a lower bound on mutual information for self-supervised representation learning, as developed in Section 37.7.4.

Chapter 38

Dynamics: Differential Equations and Generative Flows

A residual network adds a small update to its input at every layer, gradient descent nudges weights downhill step by step, and a diffusion model turns noise into an image along a learned vector field. All three discretize differential equations—rules specifying how fast a state changes. Two practical questions follow immediately: how few steps can a sampler take, and how can we learn the velocity or score direction from example images alone?

Adding noise gradually to training images answers the second question from the reverse end. Although each noisy particle wanders unpredictably, the density of the whole cloud evolves deterministically, and reversing that evolution turns Gaussian noise back into data. Earlier chapters stop short of this machinery: backpropagation differentiates a fixed computation graph rather than an ODE solution, static probability does not track moving densities, and Brownian increments of order $\sqrt{\Delta t}$ break the ordinary chain rule.

The chapter unfolds in three stages. First, deterministic ODEs move single states exactly and numerically: the fastest decaying mode caps the step size of gradient descent and every explicit Runge–Kutta solver (Section 38.2), while straighter transport paths enable few-step generation (Section 38.7.2). Along the way, driven linear ODEs yield the recurrence of state space models (Section 38.1.2.3, Section 12.4), and deep residual networks (Section 7.4) pass to neural ODEs and continuous normalizing flows (Section 38.3). Second, stochastic calculus repairs the chain rule (Section 38.4) and tracks cloud densities through the Fokker–Planck equation, showing that time reversal requires only the *score* $\nabla \log p_t$ (Section 38.5). Third, least-squares regression on noisy or interpolated pairs learns the score or marginal velocity directly without ground-truth field labels (Section 38.6, Section 38.7).

Readers headed straight for Chapter 17 may skip the continuous adjoint method (Section 38.3.2) and optimal transport (Section 38.7.3) on a first pass. Most experiments here test samplers on closed-form vector fields so discretization errors can be isolated cleanly. We assume eigendecompositions (Section 33.3), Jacobians (Section 34.4.1), and Gaussian conditioning, without measure theory.

Notation

Table 38.1 lists the symbols that recur from one section to the next. Each section restates the ones it needs and defines the letters it uses only once. Where a letter in the table carries two meanings, its row says where each one holds.

Table 38.1: Notation for the dynamics chapter.

Symbol	Meaning	First used in
$\mathbf{x}(t) \in \mathbb{R}^d$, $\dot{\mathbf{x}}$	state, its time derivative	Section 38.1
$\mathbf{x}_0$	state at $t = 0$: the start of a trajectory; for a noising process, a data point	Section 38.1
$\mathbf{f}(\mathbf{x}, t)$	vector field of an ODE; drift of an SDE	Section 38.1; Section 38.4
Φ_t; $\Phi_{s\to t}$	flow map, $\Phi_t(\mathbf{x}_0) = \mathbf{x}(t)$; the map from the state at time s to the state at time t	Section 38.1
L	Lipschitz constant of $\mathbf{f}$	Section 38.1
T	final time (horizon)	Section 38.1
$\mathbf{A}$; $\lambda_j = a_j + \mathrm{i}\, b_j$; $\mathbf{V}$, Λ	matrix of a linear field $\dot{\mathbf{x}} = \mathbf{A}\mathbf{x}$; its eigenvalues; matrix of eigenvectors, diagonal matrix of eigenvalues	Section 38.1
λ	an eigenvalue of $\mathbf{A}$ or of a Jacobian; for the reverse-time samplers, the multiple λg of the noise amplitude that they inject	Section 38.1; Section 38.5
$\mathbf{K}$	matrix $\left(\begin{smallmatrix} 0 & -1 \\ 1 & 0 \end{smallmatrix}\right)$ of the rotation $\dot{\mathbf{x}} = \mathbf{K}\mathbf{x}$	Section 38.1
$\mathbf{J}$	Jacobian of $\mathbf{f}$	Section 38.1
$\mathbf{x}^\star$	fixed point of a field, $\mathbf{f}(\mathbf{x}^\star) = \mathbf{0}$; for a gradient flow, a critical point of the loss	Section 38.1
$\mathbf{u}(t)$, $\mathbf{B}$	an input and its matrix, in $\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u}(t)$	Section 38.1
h, $t_n = nh$, $\mathbf{x}_n$	step size (written Δt in Section 38.4 and Section 38.5), time grid, state at t_n (exact, or a solver's iterate)	Section 38.1; Section 38.2
$\bar{\mathbf{A}}$, $\bar{\mathbf{B}}$	exact step with the input held over h, $\mathbf{x}_{n+1} = \bar{\mathbf{A}}\mathbf{x}_n + \bar{\mathbf{B}}\mathbf{u}_n$	Section 38.1
$R(z)$, $z = h\lambda$	stability function of a solver	Section 38.2
ε_n	global error $\lVert\mathbf{x}_n - \mathbf{x}(t_n)\rVert$	Section 38.2
$\mathcal{L}$, η, $\mathbf{H}$	loss, learning rate, Hessian	Section 38.1; Section 38.2
N	number of steps to the final time: solver steps $n = 1, \dots, N$ with $Nh = T$ (the first two sections write n for this number as well); in Section 38.6 also the noise levels $t = 1, \dots, N$ of DDPM	Section 38.3
θ	network parameters	Section 38.3
$\mathbf{a}(t)$	adjoint state	Section 38.3
p_t	density of the state at time t	Section 38.3
ϵ	in Section 38.3, the random probe of the trace estimator; from Section 38.4 on, standard Gaussian noise, as in $\mathbf{x}_t = \alpha_t \mathbf{x}_0 + \sigma_t \epsilon$, which Section 38.7 also writes $\mathbf{x}_1$, the noise endpoint	Section 38.3; Section 38.4
$\mathbf{X}_t$ (X_t in one dimension)	state of a stochastic process at time t, a random variable; $\mathbf{x}$ is a value it takes	Section 38.4
$\mathbf{W}_t$	Brownian motion	Section 38.4

Symbol	Meaning	First used in
g, $g(t)$; $\mathbf{G}$	noise amplitude of an SDE (its diffusion coefficient), constant or varying in time; noise matrix	Section 38.4
θ	rate of the OU process $dX = -\theta X\,dt + g\,dW$; in the scalar example of Section 38.3, the parameter of $\dot{x} = \theta x$	Section 38.3; Section 38.4
α_t, σ_t	mean factor and standard deviation of $\mathbf{X}_t$ given $\mathbf{X}_0$; σ_t is the noise level	Section 38.4
ϕ	a smooth function of the state: the function in Itô's lemma and the test function behind the Fokker–Planck equation (elsewhere an angle)	Section 38.4
ψ	a smooth scalar field, in the vector-calculus preliminaries	Section 38.5
$\mathbf{s}_t = \nabla \log p_t$; $\mathbf{s}_\theta$	score; learned score	Section 38.5; Section 38.6
$\mathbf{u}_t$; $\mathbf{u}_\theta$	velocity that transports p_t (the probability-flow velocity), unrelated to the input $\mathbf{u}(t)$; learned velocity	Section 38.5; Section 38.7
$\mathbf{j}$	probability current	Section 38.5
$\bar{t} = T - t$	reversed clock	Section 38.5
w_k, $r_k(\mathbf{x})$, v_k	mixture weights, responsibilities, component variances	Section 38.5
$\beta(t)$; β_t, $\bar{\alpha}_t$	rate of the variance-preserving SDE; noise schedule of DDPM, about $\beta(t)$ times the step (written β_k in Section 38.4), and its running product $\bar{\alpha}_t = \prod_{s \le t}(1 - \beta_s)$, which is our α_t^2	Section 38.4; Section 38.6
$\bar{\mathbf{x}}_t$, $\bar{\sigma}_t$	DDIM coordinates: the rescaled point $\mathbf{x}_t/\sqrt{\bar{\alpha}_t}$ and its ratio of noise to signal $\sqrt{(1-\bar{\alpha}_t)/\bar{\alpha}_t}$	Section 38.6
$w(t)$; γ	weight of noise level t in the loss; guidance scale	Section 38.6
π	coupling: a joint law of pairs of data and noise, also called a pairing	Section 38.7
i, $\mathbf{I}$	imaginary unit (upright), identity matrix	throughout

38.1 Ordinary Differential Equations

A gradient step, a residual block and one step of a diffusion sampler have something in common, as the chapter introduction claimed: each is a short move along an arrow that says how fast a state changes. The next section asks how good such steps are. This one asks what they are steps *of*. Suppose a velocity is prescribed at every point, as the wind is on a weather map, and a balloon is released into it. Is its path determined? Could two balloons released at the same spot part company? Can we tell where the balloon was an hour ago? And where does it end up?

Newton's laws of 1687 are rules of this kind: they say how the state of a body changes, not where it will be. That such a rule has exactly one solution is not automatic. The theorem that secures it, proved by Picard and Lindelöf in the 1890s, asks that the velocity not change too abruptly from one point to the next, and we shall meet a harmless-looking rule that fails the test.

Models need this theorem. When a network supplies the velocity and the model's output is the place to which the motion carries its input, uniqueness makes the output a function of

the input, and running the motion backward inverts that function, which a generative flow needs. For a linear rule we can do more and write the solution down. The eigenvalues of its matrix say whether the motion decays, grows or turns; the memory of a state space model (Section 12.4) and the convergence of a training run are read from them.

We begin with two motions that can be solved by hand, prove the theorem, and then solve linear equations completely. Everything later rests on two results: a velocity rule that changes gently has one trajectory through each point, and the real parts of the eigenvalues of a linear rule decide whether its motion decays. The last two subsections, on a decaying system that grows before it decays and on the pendulum, can be skipped on a first reading.

38.1.1 Vector Fields and Flows

38.1.1.1 Decay, Rotation, and the Spiral

The simplest motion is *decay*. The rule "move toward zero at a speed proportional to the distance" is the equation $\dot{x} = -x$, where the dot is the derivative in time. Its solution from $x(0) = x_0$ is

$$x(t) = e^{-t} x_0,$$

as differentiating shows: $\dot{x} = -e^{-t} x_0 = -x$. Each unit of time multiplies the distance to the origin by $e^{-1} \approx 0.37$.

Next comes *rotation*. In the plane, let the velocity be the position turned by a right angle,

$$\dot{\mathbf{x}} = \mathbf{K}\mathbf{x}, \qquad \mathbf{K} = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix},$$

that is $\dot{x}_1 = -x_2$ and $\dot{x}_2 = x_1$. The velocity is perpendicular to the position, because $\mathbf{x}^\top \mathbf{K}\mathbf{x} = -x_1 x_2 + x_2 x_1 = 0$, so the length of $\mathbf{x}$ never changes:

$$\frac{d}{dt}\|\mathbf{x}\|^2 = 2\,\mathbf{x}^\top \dot{\mathbf{x}} = 2\,\mathbf{x}^\top \mathbf{K}\mathbf{x} = 0.$$

A point moves on a circle, and since its speed $\|\mathbf{K}\mathbf{x}\|$ equals its radius $\|\mathbf{x}\|$ it turns at one radian per unit of time: one revolution takes 2π.

Adding half of the decay to the rotation gives the equation that serves as the running example of this section and the next two,

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x}, \qquad \mathbf{A} = -\tfrac{1}{2}\mathbf{I} + \mathbf{K} = \begin{pmatrix} -0.5 & -1 \\ 1 & -0.5 \end{pmatrix}.$$

The rotation part does not change the length and the decay part shrinks it: $\frac{d}{dt}\|\mathbf{x}\|^2 = 2\mathbf{x}^\top \mathbf{A}\mathbf{x} = -\|\mathbf{x}\|^2$. This is the decay equation again, now for the squared length, so $\|\mathbf{x}(t)\|^2 = e^{-t}\|\mathbf{x}_0\|^2$ and $\|\mathbf{x}(t)\| = e^{-t/2}\|\mathbf{x}_0\|$. A point loses length at rate $\frac{1}{2}$ while it turns at unit speed: it spirals into the origin (Figure 38.1). The eigenvalues of $\mathbf{A}$ are $-0.5 \pm \mathrm{i}$. Their real part, $-\frac{1}{2}$, gives the decay and their imaginary part, ± 1, the angular speed. This is no accident: Section 38.1.2 shows that every linear equation of this kind is a sum of decays, growths and rotations, one for each eigenvalue.

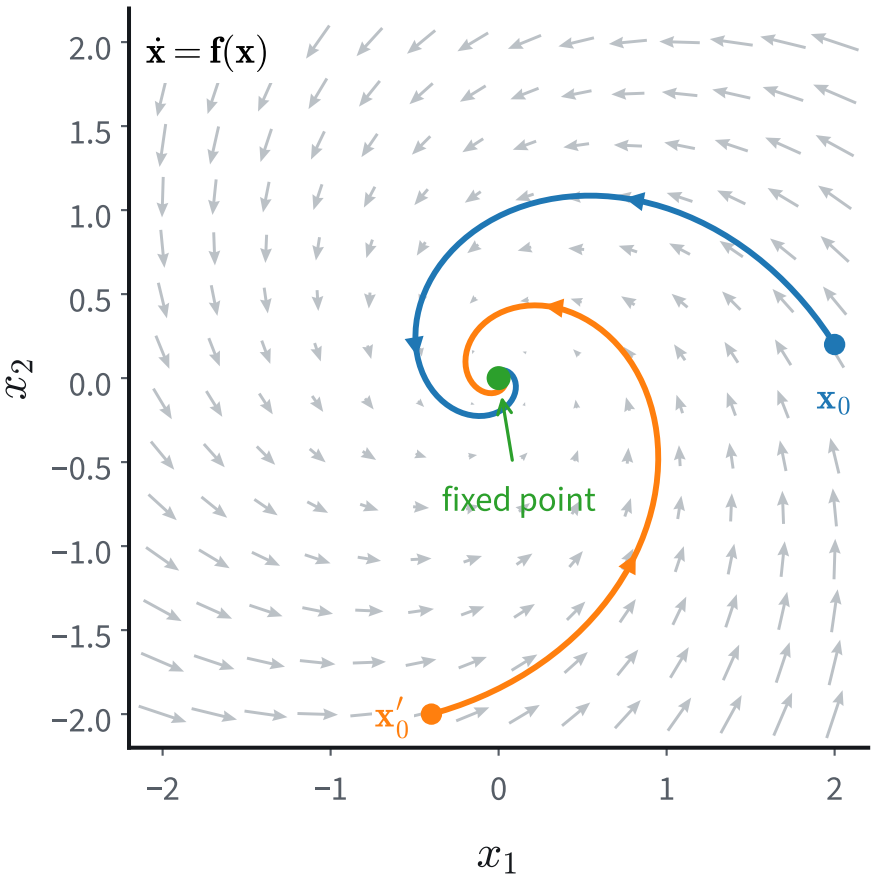


Figure 38.1: An ODE as a velocity field. The faint grid arrows are the field $\mathbf{f}(\mathbf{x}) = \mathbf{A}\mathbf{x}$ with $\mathbf{A} = \left(\begin{smallmatrix} -0.5 & -1 \\ 1 & -0.5 \end{smallmatrix}\right)$ and eigenvalues $-0.5 \pm \mathrm{i}$. The two coloured curves are trajectories: each follows the local arrow at every instant and spirals into the origin, the one point where the field vanishes (a fixed point).

38.1.1.2 The Initial-Value Problem

Most rules of motion are not linear, so we need a form that covers them all. The general problem gives a velocity rule and a starting point:

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), t), \qquad \mathbf{x}(0) = \mathbf{x}_0. \tag{38.1}$$

Here $\mathbf{x}(t) \in \mathbb{R}^d$ is the state. The function $\mathbf{f} : \mathbb{R}^d \times \mathbb{R} \to \mathbb{R}^d$ is a **vector field**: it attaches an arrow to every point at every time. A **trajectory** is a differentiable path whose tangent at each instant is the arrow at its current position.

The vector $\mathbf{x}_0$ is the **initial value**, and Equation 38.1 is called an **initial-value problem**. When $\mathbf{f}$ does not depend on t, the system is **autonomous**: the arrows stay fixed and only the state moves. Decay, rotation and the spiral are the autonomous fields $\mathbf{f}(\mathbf{x}) = -\mathbf{x}$, $\mathbf{K}\mathbf{x}$ and $\mathbf{A}\mathbf{x}$. A weather map is not autonomous, since the wind changes while the balloon flies.

Integrating both sides from 0 to t (Section 34.6) gives the equivalent **integral form**

$$\mathbf{x}(t) = \mathbf{x}_0 + \int_0^t \mathbf{f}(\mathbf{x}(s), s)\, ds. \tag{38.2}$$

The solution is the start plus the accumulated velocity. The unknown path appears on both sides, so this is still an equation and not a formula, but the existence proof below works with it.

38.1.1.3 Existence and Uniqueness

Does an initial-value problem have a solution, and only one? Not always. Consider $\dot{x} = \sqrt{|x|}$ from $x_0 = 0$. The constant path $x \equiv 0$ solves it. So does, for every waiting time $c \geq 0$,

$$x_c(t) = \begin{cases} 0, & t \leq c, \\ (t-c)^2/4, & t > c, \end{cases}$$

since $\dot{x}_c = (t-c)/2 = \sqrt{x_c}$ for $t > c$. A particle at rest may leave at any time. The solutions fan out from the axis, one for each departure time (panel (a) of Figure 38.2), and we shall call this picture the *fan*. A model defined by this field would not assign one output to its input. The cause is the slope of $\sqrt{|x|}$, which is unbounded at 0: a small displacement x produces the much larger velocity $\sqrt{|x|}$.

For a well-behaved field a solution can be built by iterating the integral form Equation 38.2: guess a path, read off the velocities that the field prescribes along it, add them up from the start to obtain a new path, and repeat. This is **Picard iteration**. For $\dot{x} = x$ with $x_0 = 1$, start from the constant path $\varphi_0 \equiv 1$ and set $\varphi_{k+1}(t) = 1 + \int_0^t \varphi_k(s)\, ds$:

$$\varphi_1(t) = 1 + t, \qquad \varphi_2(t) = 1 + t + \tfrac{t^2}{2}, \qquad \varphi_3(t) = 1 + t + \tfrac{t^2}{2} + \tfrac{t^3}{6}.$$

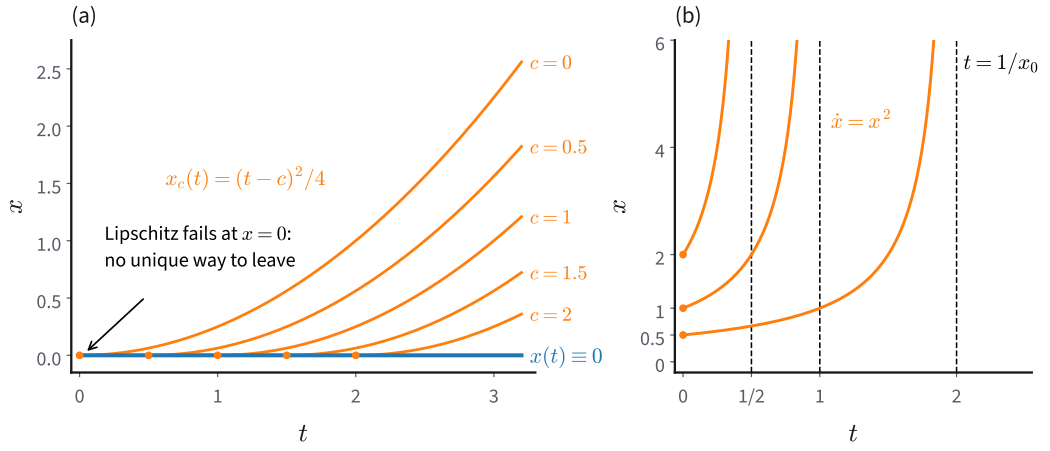


Figure 38.2: Two fields that the theorem below does not cover. (a) Solutions of $\dot{x} = \sqrt{|x|}$ through $x(0) = 0$: the constant solution $x \equiv 0$ and a parabolic solution $x_c(t) = (t-c)^2/4$ for every departure time c. The slope of $\sqrt{|x|}$ is unbounded at the origin, so no constant bounds how fast the velocity changes there (the Lipschitz condition of the theorem fails), and the solution is not unique. (b) Solutions of $\dot{x} = x^2$ from $x_0 = 2$, 1 and $1/2$ (dots). Here the slope of the field is bounded near every point but not by one constant for all x, and each solution, $x_0/(1 - x_0 t)$, exists only up to the time $t = 1/x_0$ (dashed), where it reaches infinity.

The iterates converge to the solution e^t. Here each iteration adds one Taylor term of e^t; for other fields the iterates look different, but they still converge, provided the field does not change too abruptly from point to point. The theorem makes "too abruptly" precise.

Theorem (Picard–Lindelöf). *Let* $\mathbf{f}(\mathbf{x}, t)$ *be continuous in* t *and globally Lipschitz in* $\mathbf{x}$*: there is an* L *with*

$$\|\mathbf{f}(\mathbf{x}, t) - \mathbf{f}(\mathbf{y}, t)\| \le L \, \|\mathbf{x} - \mathbf{y}\| \qquad \text{for all } \mathbf{x}, \mathbf{y}, t.$$

Then the initial-value problem Equation 38.1 has exactly one solution on $[0, T]$*, for every* $\mathbf{x}_0$ *and every* $T > 0$ (Coddington and Levinson 1955).

This is the theorem that allows a model to be defined by a field. The constant L bounds how fast the arrows may change from point to point. Decay has $L = 1$, and a linear field $\mathbf{Ax}$ has $L = \|\mathbf{A}\|$, where the norm of a matrix is, here and in the rest of the section, its spectral norm: the largest factor by which the matrix stretches a vector (Section 33.4.2.2). The fan is what happens when no such L exists near 0. The proof shows that the Picard iterates converge, and that they converge to the only solution.

Proof. A continuous path φ on $[0, \delta]$ solves the problem if and only if it is a fixed point of the **Picard operator**

$$(P\varphi)(t) = \mathbf{x}_0 + \int_0^t \mathbf{f}(\varphi(s), s) \, ds.$$

In words, P is one round of Picard iteration: it reads the velocities along a guessed path and adds them up from the start. (A solution satisfies the integral form; conversely, a continuous fixed point is differentiable by the fundamental theorem of calculus, with the right derivative and the right start.)

The step that is not routine pulls the Lipschitz bound through the integral. For two paths on $[0,\delta]$, write $d(\varphi,\psi)=\max_{t\le\delta}\|\varphi(t)-\psi(t)\|$. Then for $t\le\delta$,

$$\|(P\varphi)(t)-(P\psi)(t)\| \le \int_0^t L\,\|\varphi(s)-\psi(s)\|\,ds \le L\,\delta\;d(\varphi,\psi).$$

With $\delta=1/(2L)$, every application of P halves the distance between two candidate paths.

Start from the constant path $\varphi_0\equiv\mathbf{x}_0$ and set $\varphi_{k+1}=P\varphi_k$. Consecutive iterates then differ by at most $2^{-k}d(\varphi_1,\varphi_0)$, and these differences add up like a geometric series. So the iterates converge in the distance d, that is, uniformly on $[0,\delta]$, to a path φ. We take one fact from analysis on trust here: a uniform limit of continuous paths is continuous. Since $d(P\varphi_k,P\varphi)\le\frac{1}{2}d(\varphi_k,\varphi)\to 0$, the paths $P\varphi_k=\varphi_{k+1}$ converge to $P\varphi$ as well as to φ. Hence $\varphi=P\varphi$: the limit is a solution. Two solutions φ,ψ would satisfy $d(\varphi,\psi)=d(P\varphi,P\psi)\le\frac{1}{2}d(\varphi,\psi)$, so they coincide.

Finally restart at $t=\delta$ from $\varphi(\delta)$. The step length $1/(2L)$ does not depend on where the solution is, because L is the same everywhere, so $\lceil 2LT\rceil$ pieces of this length cover $[0,T]$, with one solution on each. ■

The theorem asks for a Lipschitz bound and for that bound to hold globally, and each requirement rules out a different failure. The fan shows what happens without a Lipschitz bound. Without the global bound, the solution can blow up. The field $\dot{x}=x^2$ is smooth, and we can solve it. Since $\frac{d}{dt}(-1/x)=\dot{x}/x^2=1$, the quantity $-1/x$ grows at unit rate: $-1/x(t)=t-1/x_0$, and so

$$x(t)=\frac{x_0}{1-x_0t},$$

which from a start $x_0>0$ reaches infinity at the finite time $t=1/x_0$ (panel (b) of Figure 38.2). The last step of the proof shows why the theorem does not cover it. On the interval $|x|\le\rho$ the field is Lipschitz with $L=2\rho$, since $|x^2-y^2|=|x+y|\,|x-y|$. The restart length $1/(2L)=1/(4\rho)$ therefore shrinks as the solution grows, and the restarts can add up to a finite time. A field that is Lipschitz only on bounded sets guarantees a solution only for a while.

38.1.1.4 The Flow Map

A generative model moves a whole cloud of starting points at once, so we now fix the field and vary the start. Let us take the simple case first, a field that does not change in time; the wind returns at the end of the subsection. For an autonomous, globally Lipschitz field, write

$$\Phi_t(\mathbf{x}_0)=\mathbf{x}(t)$$

for the state at time t of the one trajectory started at $\mathbf{x}_0$. This **flow map** $\Phi_t : \mathbb{R}^d \to \mathbb{R}^d$ moves all of space forward by t along the arrows.

For decay the flow map is $\Phi_t(x) = e^{-t}x$. Flowing for s and then for t gives $\Phi_t(\Phi_s(x)) = e^{-t}e^{-s}x = e^{-(t+s)}x = \Phi_{t+s}(x)$, and $\Phi_{-t}(x) = e^{t}x$ undoes Φ_t. The same law holds for every autonomous field:

$$\Phi_0 = \mathrm{id}, \qquad \Phi_{t+s} = \Phi_t \circ \Phi_s. \tag{38.3}$$

The reason is that the arrows do not change with time: the path $\tau \mapsto \mathbf{x}(s+\tau)$ starts at $\Phi_s(\mathbf{x}_0)$ and follows the same arrows, so by uniqueness it is the trajectory from that point. The decay example suggests that the law also holds for negative times. It does, because running a field backward is the same theorem applied to a second field.

Corollary (the flow map is a bijection). *Let $\mathbf{f}(\mathbf{x})$ be autonomous and globally Lipschitz. Then Φ_t exists for every $t \in \mathbb{R}$, and each Φ_t is a bijection of $\mathbb{R}^d$ with inverse Φ_{-t}. If instead $\mathbf{f}(\mathbf{x}, t)$ depends on time and satisfies the hypotheses of the theorem, then for every $T > 0$ the map that carries the state at time 0 to the state at time T is a bijection of $\mathbb{R}^d$.*

Proof. The field $\mathbf{g}(\mathbf{y}) = -\mathbf{f}(\mathbf{y})$ is Lipschitz with the same L. A path $\mathbf{x}(t)$ solves $\dot{\mathbf{x}} = \mathbf{f}$ on $[-T, 0]$ if and only if $\mathbf{y}(s) = \mathbf{x}(-s)$ solves $\dot{\mathbf{y}} = \mathbf{g}$ on $[0, T]$, since $\dot{\mathbf{y}}(s) = -\dot{\mathbf{x}}(-s) = -\mathbf{f}(\mathbf{y}(s))$. The theorem applied to $\mathbf{g}$ gives a unique solution backward in time for every start, so Φ_t is defined for all real t. The composition law Equation 38.3, whose argument did not use the sign of s or t, then gives $\Phi_{-t} \circ \Phi_t = \Phi_t \circ \Phi_{-t} = \Phi_0 = \mathrm{id}$.

The reversal did not need the arrows to stand still. For a field that depends on time, play the film backward from time T: a path $\mathbf{x}(t)$ solves $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, t)$ on $[0, T]$ if and only if $\mathbf{y}(s) = \mathbf{x}(T - s)$ solves $\dot{\mathbf{y}} = \mathbf{g}(\mathbf{y}, s)$ with $\mathbf{g}(\mathbf{y}, s) = -\mathbf{f}(\mathbf{y}, T - s)$. This $\mathbf{g}$ is again continuous in time and Lipschitz with the same L, so by the theorem exactly one reversed path leaves each point $\mathbf{z}$. Read forward, it is the one trajectory that ends at $\mathbf{z}$ at time T: every end state comes from exactly one start. ■

A field that changes in time loses the law Equation 38.3 in this form only. An hour of wind from noon and an hour from midnight are different maps, so the map depends on the starting time as well as on the duration. Write $\Phi_{s\to t}(\mathbf{x})$ for the state at time t of the trajectory that is at $\mathbf{x}$ at time s. Travelling from s to r and on from r to t is travelling from s to t, by the same uniqueness argument as before:

$$\Phi_{s\to s} = \mathrm{id}, \qquad \Phi_{r\to t} \circ \Phi_{s\to r} = \Phi_{s\to t}. \tag{38.4}$$

The three times may come in any order, so $\Phi_{t\to s}$ inverts $\Phi_{s\to t}$. For an autonomous field $\Phi_{s\to t} = \Phi_{t-s}$, and Equation 38.4 is the old law. We keep the symbol Φ_t for a field that depends on time, as short for $\Phi_{0\to t}$. The two-time law is not a formality: the few-step samplers of Section 17.8 train a network to imitate $\Phi_{s\to t}$ itself, and some of them take the law as their training signal: two short jumps of the network must make its long one.

One consequence is that two different trajectories are never at the same point at the same time: run backward from such a meeting, the theorem allows only one path. For a Lipschitz wind this answers the first three questions of the opening. The balloon's path is determined, two balloons released together stay together, and playing the weather map backward shows where they were an hour ago.

An invertible map from noise to data whose inverse can be computed is what a generative flow needs, and the fields of many neural networks qualify. A field built from linear maps and Lipschitz activations, such as $\mathbf{f}(\mathbf{x}) = \mathbf{W}_2 \tanh(\mathbf{W}_1\mathbf{x}+\mathbf{b}_1)+\mathbf{b}_2$, is globally Lipschitz, with L at most the product of the weight norms and the activations' constants ($\|\mathbf{W}_2\|\,\|\mathbf{W}_1\|$ here). The model of Section 38.3, whose field may also depend on time, is therefore well-posed, and by the corollary its map from input to output is a bijection.

38.1.2 Linear ODEs and Stability

The last question of the opening, where the motion ends up, has a complete answer for one family. Linear fields can be solved in closed form, and they describe most other fields near a point where the field vanishes.

38.1.2.1 The Matrix Exponential

The scalar equation $\dot{x} = ax$ is solved by $x(t) = e^{at}x_0$. For a linear system

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x}, \qquad \mathbf{x}(0) = \mathbf{x}_0, \qquad \mathbf{A} \in \mathbb{R}^{d\times d},$$

we look for the same form with a matrix in the exponent. For any square matrix $\mathbf{C}$, define the **matrix exponential** by the power series that defines e^z:

$$e^{\mathbf{C}} = \sum_{k=0}^{\infty} \frac{\mathbf{C}^k}{k!} = \mathbf{I} + \mathbf{C} + \frac{\mathbf{C}^2}{2} + \frac{\mathbf{C}^3}{6} + \cdots. \tag{38.5}$$

The series converges for every matrix. Stretching by one matrix and then by another stretches by at most the product of the two factors, $\|\mathbf{CD}\| \le \|\mathbf{C}\|\,\|\mathbf{D}\|$, so $\|\mathbf{C}^k\| \le \|\mathbf{C}\|^k$, and term by term the series is dominated by the scalar series for $e^{\|\mathbf{C}\|}$, which converges.

Proposition (the matrix exponential solves linear ODEs). *$\mathbf{x}(t) = e^{\mathbf{A}t}\mathbf{x}_0$ is the unique solution of $\dot{\mathbf{x}} = \mathbf{A}\mathbf{x}$, $\mathbf{x}(0) = \mathbf{x}_0$.*

Proof. Differentiate the series Equation 38.5 with $\mathbf{C} = \mathbf{A}t$ term by term (legitimate: on any bounded t-interval the differentiated series still converges uniformly, by the same domination):

$$\frac{d}{dt}e^{\mathbf{A}t} = \sum_{k=1}^{\infty} \frac{\mathbf{A}^k\, t^{k-1}}{(k-1)!} = \mathbf{A}\sum_{j=0}^{\infty} \frac{(\mathbf{A}t)^j}{j!} = \mathbf{A}\, e^{\mathbf{A}t}.$$

So $\mathbf{x}(t) = e^{\mathbf{A}t}\mathbf{x}_0$ satisfies the ODE, and at $t = 0$ the series collapses to $\mathbf{I}$, so $\mathbf{x}(0) = \mathbf{x}_0$. Uniqueness is Picard–Lindelöf: the field $\mathbf{x} \mapsto \mathbf{A}\mathbf{x}$ is Lipschitz with $L = \|\mathbf{A}\|$. $\blacksquare$

An infinite series of matrix powers is not yet a formula we can read. The eigendecomposition turns it into d scalar exponentials. Recall from Section 33.3 that if $\mathbf{A}$ has d independent eigenvectors $\mathbf{v}_j$, $\mathbf{A}\mathbf{v}_j = \lambda_j\mathbf{v}_j$, then with $\mathbf{V} = (\mathbf{v}_1, \dots, \mathbf{v}_d)$ and $\Lambda = \operatorname{diag}(\lambda_1, \dots, \lambda_d)$ we have $\mathbf{A} = \mathbf{V}\Lambda\mathbf{V}^{-1}$.

Powers then telescope, $(\mathbf{A}t)^k = \mathbf{V}(\Lambda t)^k\mathbf{V}^{-1}$, because each adjacent $\mathbf{V}^{-1}\mathbf{V}$ cancels, and summing the series Equation 38.5 gives

$$e^{\mathbf{A}t} = \mathbf{V} e^{\Lambda t} \mathbf{V}^{-1}, \qquad e^{\Lambda t} = \operatorname{diag}\left(e^{\lambda_1 t}, \dots, e^{\lambda_d t}\right). \tag{38.6}$$

The eigenvalue section quoted this formula and left its development to us. It says that the system decouples. Expand the start in the eigenbasis, $\mathbf{x}_0 = \sum_j c_j \mathbf{v}_j$; then

$$\mathbf{x}(t) = \sum_{j=1}^{d} c_j \, e^{\lambda_j t} \, \mathbf{v}_j .$$

Along each eigendirection the system is the scalar equation $\dot{x} = \lambda_j x$, evolving independently of the others. Each term $c_j e^{\lambda_j t}\mathbf{v}_j$ is called a **mode** of the system.

For the spiral we can do better than the general formula. Its matrix is a sum, $\mathbf{A} = -\frac{1}{2}\mathbf{I} + \mathbf{K}$, and we would like to split the exponential as $e^{p+q} = e^p e^q$ splits for numbers. For matrices the rule $e^{\mathbf{P}+\mathbf{Q}} = e^{\mathbf{P}} e^{\mathbf{Q}}$ holds when $\mathbf{PQ} = \mathbf{QP}$, since then $(\mathbf{P}+\mathbf{Q})^k$ expands by the binomial theorem as it does for numbers. The identity commutes with every matrix, so $e^{\mathbf{A}t} = e^{-t/2} e^{\mathbf{K}t}$. Because $\mathbf{K}^2 = -\mathbf{I}$, the powers of $\mathbf{K}$ cycle through $\mathbf{I}, \mathbf{K}, -\mathbf{I}, -\mathbf{K}$, and the series for $e^{\mathbf{K}t}$ splits into even and odd terms, the series of $\cos t$ and $\sin t$:

$$e^{\mathbf{A}t} = e^{-t/2}\left(\cos t\, \mathbf{I} + \sin t\, \mathbf{K}\right) = e^{-t/2} \begin{pmatrix} \cos t & -\sin t \\ \sin t & \cos t \end{pmatrix}.$$

The flow shrinks by $e^{-t/2}$ and turns by t radians, as we found from the field itself. From $\mathbf{x}_0 = (2, 0)$ at $t = 2$ the state is $e^{-1}(2\cos 2,\ 2\sin 2) = (-0.306,\ 0.669)$, with length $2e^{-1} = 0.736$. For this $\mathbf{A}$ the eigenvectors and eigenvalues $-0.5 \pm \mathrm{i}$ are complex, but the product $\mathbf{V} e^{\Lambda t} \mathbf{V}^{-1}$ is real.

A word of caution: when the two matrices do not commute, the split fails. For $\mathbf{P} = \left(\begin{smallmatrix} 0 & 1 \\ 0 & 0 \end{smallmatrix}\right)$ and $\mathbf{Q} = \mathbf{P}^{\top}$, which both square to zero, $e^{\mathbf{P}} e^{\mathbf{Q}} = (\mathbf{I}+\mathbf{P})(\mathbf{I}+\mathbf{Q}) = \left(\begin{smallmatrix} 2 & 1 \\ 1 & 1 \end{smallmatrix}\right)$, whereas $e^{\mathbf{P}+\mathbf{Q}}$ has $\cosh 1 \approx 1.54$ in both diagonal entries.

There is a third route to $e^{\mathbf{A}t}$, which needs neither a series nor eigenvectors: the compound-interest limit $(\mathbf{I} + t\mathbf{A}/n)^n$ as $n \to \infty$. Each factor moves the state along the field for the short time t/n. How fast does the limit arrive? In the eigenbasis the product is $\mathbf{V}\operatorname{diag}\big((1+\lambda_j t/n)^n\big)\mathbf{V}^{-1}$, so it is enough to ask one eigenvalue. Since $\log(1+z) = z - z^2/2 + \cdots$,

$$\left(1 + \frac{\lambda t}{n}\right)^n = \exp\Big(n \log\big(1 + \tfrac{\lambda t}{n}\big)\Big) \approx e^{\lambda t}\left(1 - \frac{(\lambda t)^2}{2n}\right),$$

so each $(1+\lambda_j t/n)^n$ tends to $e^{\lambda_j t}$ and misses it by the fraction $|\lambda_j t|^2/(2n)$ of its size. For the spiral at $t = 2$ this fraction is $2.5/n$ for both eigenvalues, and $|e^{\lambda t}| = e^{-1}$, which predicts an error of about $0.9/n$. The cell computes $e^{\mathbf{A}t}$ from the eigendecomposition and compares the compounding with it at three values of n.

```
A = np.array([[-0.5, -1.0], [1.0, -0.5]])      # spiral: eigenvalues -0.5 +- i
t, x0 = 2.0, np.array([2.0, 0.0])
lam, V = np.linalg.eig(A)                      # A = V diag(lam) V^{-1}
expAt = (V @ np.diag(np.exp(lam * t)) @ np.linalg.inv(V)).real
```

```
x = expAt @ x0
print('eigenvalues of A:', lam.round(4))
print('x(2) =', x.round(4), f' length {np.linalg.norm(x):.6f}')
for n in [2 ** 10, 2 ** 16, 2 ** 24]:          # compounding (I + tA/n)^n
    err = np.abs(np.linalg.matrix_power(np.eye(2) + (t / n) * A, n) - expAt).max()
    print(f'n = 2^{int(np.log2(n))}: error {err:.1e}, n * error = {n * err:.2f}')
```

```
eigenvalues of A: [-0.5+1.j -0.5-1.j]
x(2) = [-0.3062  0.669 ]  length 0.735759
n = 2^10: error 8.8e-04, n * error = 0.90
n = 2^16: error 1.4e-05, n * error = 0.90
n = 2^24: error 5.4e-08, n * error = 0.90
```

The state $(-0.3062,\ 0.6690)$ and its length 0.735759 agree with the values computed by hand. The compounding converges at the predicted rate and no faster: the largest entry of its error, multiplied by n, stays at 0.90 from $n = 2^{10}$ to $n = 2^{24}$. This is compound interest with a matrix interest rate. It defines the exponential well and computes it badly. After nearly seventeen million compounding periods the error is still 5×10^{-8}.

38.1.2.2 Stability from Eigenvalues

The decoupled form tells us at a glance whether a linear system settles down. Write each eigenvalue as $\lambda_j = a_j + \mathrm{i}\, b_j$. Its mode has magnitude $|e^{\lambda_j t}| = e^{a_j t}$, since $e^{\mathrm{i}b_j t} = \cos b_j t + \mathrm{i} \sin b_j t$ has modulus one. The real part sets growth or decay; the imaginary part sets rotation. For a diagonalizable $\mathbf{A}$ this gives the following classification.

- If $\operatorname{Re} \lambda_j < 0$ for all j, every mode decays, and $\mathbf{x}(t) \to \mathbf{0}$ from every start: the origin is **asymptotically stable**. The slowest mode sets the decay rate $-\max_j \operatorname{Re} \lambda_j$.
- If $\operatorname{Re} \lambda_j > 0$ for some j, that mode grows exponentially, and every start with a component along $\mathbf{v}_j$ escapes: the origin is **unstable**.
- In between, a mode with $\operatorname{Re} \lambda_j = 0$ neither grows nor decays. It stands still or circles forever, as the rotation did.

A complex pair $a \pm \mathrm{i}b$ is the spiral with general rates: a rotation at angular speed b that decays or grows with the sign of a. To see this, write the eigenvector of $\lambda = a + \mathrm{i}b$ as $\mathbf{v} = \mathbf{p} + \mathrm{i}\mathbf{q}$ with real $\mathbf{p}, \mathbf{q}$ (independent, since $b \neq 0$). The real and imaginary parts of $\mathbf{A}\mathbf{v} = (a + \mathrm{i}b)\mathbf{v}$ are

$$\mathbf{A}\mathbf{p} = a\mathbf{p} - b\mathbf{q}, \qquad \mathbf{A}\mathbf{q} = b\mathbf{p} + a\mathbf{q}.$$

So $\mathbf{A}$ maps the plane of $\mathbf{p}$ and $\mathbf{q}$ to itself, and in the basis $(\mathbf{q}, \mathbf{p})$ it is the matrix $a\mathbf{I} + b\mathbf{K}$. The computation for the spiral then gives

$$e^{(a\mathbf{I} + b\mathbf{K})t} = e^{at}\left(\cos bt\, \mathbf{I} + \sin bt\, \mathbf{K}\right) :$$

scaling by e^{at} composed with rotation by the angle bt. The same block, iterated in discrete time instead of flowed, appears in Section 33.3.1.6.

In two dimensions this leaves six pictures, and Figure 38.3 draws one field of each kind. With real eigenvalues the origin is a **stable node** (both negative), an **unstable node** (both positive) or a **saddle** (one of each sign). With a complex pair it is a **stable spiral**, an **unstable**

spiral or, when the real part is zero, a **center**. (A zero eigenvalue is left out: it gives a whole line of points at rest.) The stable spiral is our running example.

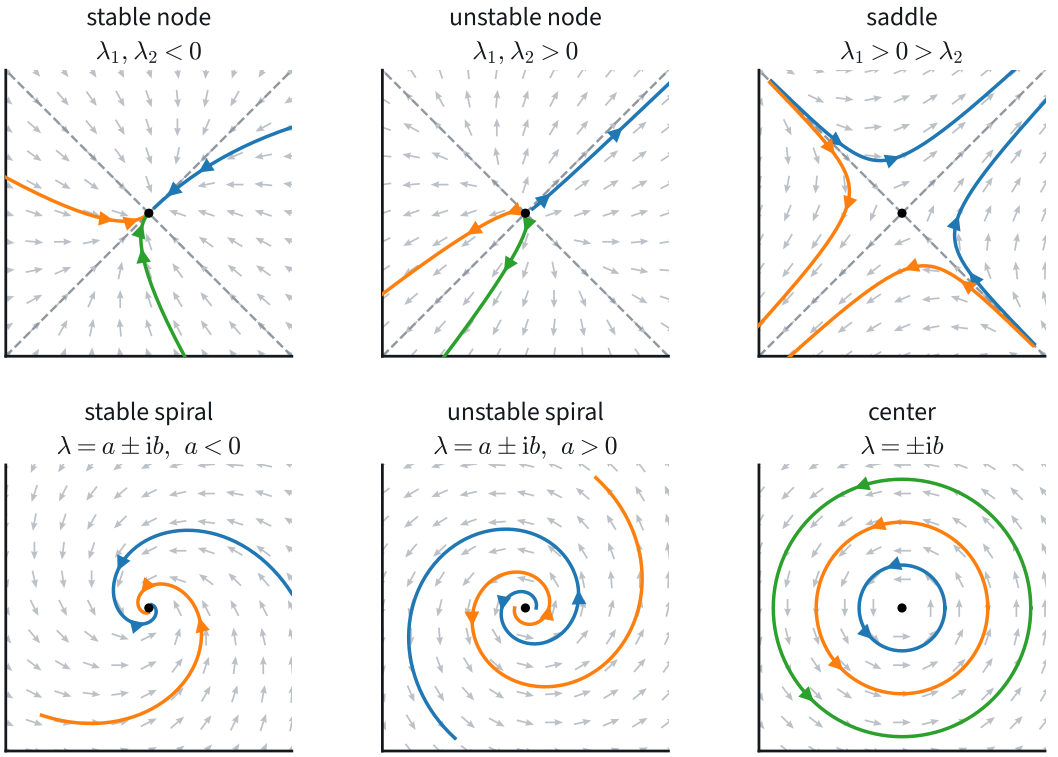


Figure 38.3: Six planar linear systems. Top row, real eigenvalues: a stable node (both negative), an unstable node (both positive) and a saddle (opposite signs), with the eigendirections dashed; trajectories enter or leave along them. Bottom row, a complex pair $a \pm ib$: a stable spiral ($a < 0$), an unstable spiral ($a > 0$) and a center ($a = 0$, closed orbits).

In training we meet the top row most often. Gradient descent in continuous time on a quadratic loss $\frac{1}{2}\mathbf{x}^\top \mathbf{H}\mathbf{x}$ is the linear field $\dot{\mathbf{x}} = -\mathbf{H}\mathbf{x}$, and a symmetric $\mathbf{H}$ has real eigenvalues, so such a loss produces only nodes and saddles. Rotation needs a field that is not a gradient, such as the two players of the smallest GAN, who circle their equilibrium in a center (Section 16.4).

When $\mathbf{A}$ has too few eigenvectors to be diagonalized, the modes acquire polynomial factors $t^k e^{\lambda t}$. They change the verdict only for eigenvalues on the imaginary axis, and Exercise 3 works out the smallest case.

38.1.2.3 Linear Systems with an Input

So far nothing pushed the system from outside. The state of a state space model that reads a sequence (Section 12.4) obeys a linear equation with an input $\mathbf{u}(t)$, which enters the state through a matrix $\mathbf{B}$:

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u}(t), \qquad \mathbf{x}(0) = \mathbf{x}_0.$$

The matrix exponential solves this equation as well, and one observation finds the solution. Without the input, the quantity $e^{-\mathbf{A}t}\mathbf{x}(t)$ stands still at $\mathbf{x}_0$: it is the state carried back to time 0 along the free motion. With the input it can move only as fast as the input pushes it. Indeed, since $\frac{d}{dt}e^{-\mathbf{A}t} = -e^{-\mathbf{A}t}\mathbf{A}$, the product rule gives $\frac{d}{dt}(e^{-\mathbf{A}t}\mathbf{x}) = e^{-\mathbf{A}t}(\dot{\mathbf{x}} - \mathbf{A}\mathbf{x}) = e^{-\mathbf{A}t}\mathbf{B}\mathbf{u}$, and the right-hand side no longer contains $\mathbf{x}$. Integrating from 0 to t and multiplying by $e^{\mathbf{A}t}$ yields

$$\mathbf{x}(t) = e^{\mathbf{A}t}\mathbf{x}_0 + \int_0^t e^{\mathbf{A}(t-s)}\,\mathbf{B}\mathbf{u}(s)\,ds. \tag{38.7}$$

This is the **variation of constants** formula, so called because the constant $\mathbf{x}_0$ of the free solution $e^{\mathbf{A}t}\mathbf{x}_0$ has been allowed to vary in time. The state is the decayed start plus every past input, each decayed for the time since it arrived.

A sequence model receives its input one sample per step, not at every instant. So cut time into steps of length h, with grid times $t_n = nh$ and grid states $\mathbf{x}_n = \mathbf{x}(t_n)$, and suppose that the input is constant during each step, $\mathbf{u}(s) = \mathbf{u}_n$ for $s \in [t_n, t_n + h)$. This is called a *zero-order hold*. Apply Equation 38.7 to one step, which starts from $\mathbf{x}_n$ at time t_n. The constant $\mathbf{u}_n$ comes out of the integral, and with $\tau = t_{n+1} - s$ what remains is $\int_0^h e^{\mathbf{A}\tau}d\tau$, the same matrix for every step. The grid states therefore obey a linear recurrence, without any approximation:

$$\mathbf{x}_{n+1} = \bar{\mathbf{A}}\,\mathbf{x}_n + \bar{\mathbf{B}}\,\mathbf{u}_n, \qquad \bar{\mathbf{A}} = e^{\mathbf{A}h}, \qquad \bar{\mathbf{B}} = \int_0^h e^{\mathbf{A}\tau}\,d\tau\;\mathbf{B} = \mathbf{A}^{-1}\big(e^{\mathbf{A}h} - \mathbf{I}\big)\mathbf{B}, \tag{38.8}$$

where the last form needs $\mathbf{A}$ to be invertible. In one dimension these are $\bar{a} = e^{h\lambda}$ and $\bar{b} = (e^{h\lambda} - 1)\,b/\lambda$, the coefficients on which the state space models of Section 12.4.1.2 are built (there the step is Δ and the state is h).

The point to remember is that $|e^{h\lambda}| = e^{h\,\mathrm{Re}\,\lambda} < 1$ for every step $h > 0$ whenever $\mathrm{Re}\,\lambda < 0$: the exact recurrence is stable whenever the equation is. Compare the factor $1 + h\lambda$ of one compounding period, which follows the field for a time h without looking again. On $\dot{x} = -50x$ with $h = 0.05$ the exact factor is $e^{-2.5} = 0.08$, while $1 + h\lambda = -1.5$ flips the sign of the state and enlarges it. Stepping along the field in this way is Euler's method, and Section 38.2 shows what the factor -1.5 does to a training run.

The recurrence is exact only while the input is constant over each step, and a sequence model reads samples of a signal that moves between them. What does the hold cost then? The cell runs the scalar recurrence on $\dot{x} = \lambda x + \sin t$ from $x(0) = 1$ to $T = 2$, with the input held at its value at the start of each step, and compares the result with the closed-form solution. It does so for a slow rate $\lambda = -1$ and a fast one $\lambda = -50$, and prints $1 + h\lambda$ beside the exact factor $e^{h\lambda}$.

```
def zoh(lam, h, u, x0=1.0, T=2.0):
    """Exact steps of dx/dt = lam x + u(t), u held at u(t_n) over each step."""
    a, b = np.exp(h * lam), np.expm1(h * lam) / lam
    x = x0
    for n in range(round(T / h)):
```

```
            x = a * x + b * u(n * h)
        return x

    def exact_sin(lam, T=2.0):
        """Solution of dx/dt = lam x + sin t with x(0) = 1, at time T."""
        c = 1 + lam ** 2
        return np.exp(lam * T) * (1 + 1 / c) - (lam * np.sin(T) + np.cos(T)) / c

    print(f'{"lam":>5} {"h":>5} {"e^(h lam)":>10} {"1 + h lam":>10}'
          f' {"error at T":>11}')
    for lam in [-1.0, -50.0]:
        for h in [0.5, 0.1, 0.02]:
            err = abs(zoh(lam, h, np.sin) - exact_sin(lam))
            print(f'{lam:5.0f} {h:5.2f} {np.exp(h * lam):10.3g}'
                  f' {1 + h * lam:10.2f} {err:11.1e}')
```

```
  lam     h   e^(h lam)   1 + h lam   error at T
   -1  0.50       0.607        0.50      6.8e-02
   -1  0.10       0.905        0.90      9.7e-03
   -1  0.02        0.98        0.98      1.8e-03
  -50  0.50    1.39e-11      -24.00      1.6e-03
  -50  0.10     0.00674       -4.00      5.8e-04
  -50  0.02       0.368        0.00      9.2e-05
```

The recurrence itself approximates nothing, so the only error is that of the hold, which pretends that the sine stands still during each step. It shrinks about in proportion to the step: for $\lambda = -1$ it is 0.068, 0.0097 and 0.0018 at $h = 0.5$, 0.1 and 0.02. The row to study is $\lambda = -50$ with $h = 0.5$, a step twenty-five times longer than the time $1/|\lambda|$ in which the state forgets. There the exact factor is 1.4×10^{-11} and the error is 0.0016, where the factor $1 + h\lambda = -24$ would have blown the state up. The models of Section 12.4.1.2 rest on this fact. As long as the rates stay negative, no step size that training might choose makes the recurrence grow.

38.1.2.4 Transient Growth

One caveat to the verdict by eigenvalues is still owed. When every mode decays, the state goes to zero, but its length $\|\mathbf{x}(t)\|$ need not decrease on the way. A triangular matrix shows it:

$$\dot{\mathbf{x}} = \begin{pmatrix} -1 & 10 \\ 0 & -2 \end{pmatrix} \mathbf{x}, \qquad \mathbf{x}_0 = (0, 1),$$

with eigenvalues -1 and -2, both negative. The second row involves only x_2, so $x_2(t) = e^{-2t}$. The first row is then $\dot{x}_1 = -x_1 + 10e^{-2t}$, a decay equation with an input, and Equation 38.7 gives

$$x_1(t) = 10 \int_0^t e^{-(t-s)} e^{-2s}\, ds = 10\left(e^{-t} - e^{-2t}\right).$$

The first coordinate starts at 0 and rises until its derivative $10(2e^{-2t} - e^{-t})$ vanishes, at $e^{-t} = \tfrac{1}{2}$. There, at $t = \log 2 \approx 0.69$, it equals $10(\tfrac{1}{2} - \tfrac{1}{4}) = 2.5$, and $x_2 = \tfrac{1}{4}$, so a state that

started with length 1 now has length $\sqrt{2.5^2 + 0.25^2} \approx 2.51$. From then on x_1 decays, in the long run like $10e^{-t}$, the rate of the slower mode (Figure 38.4).

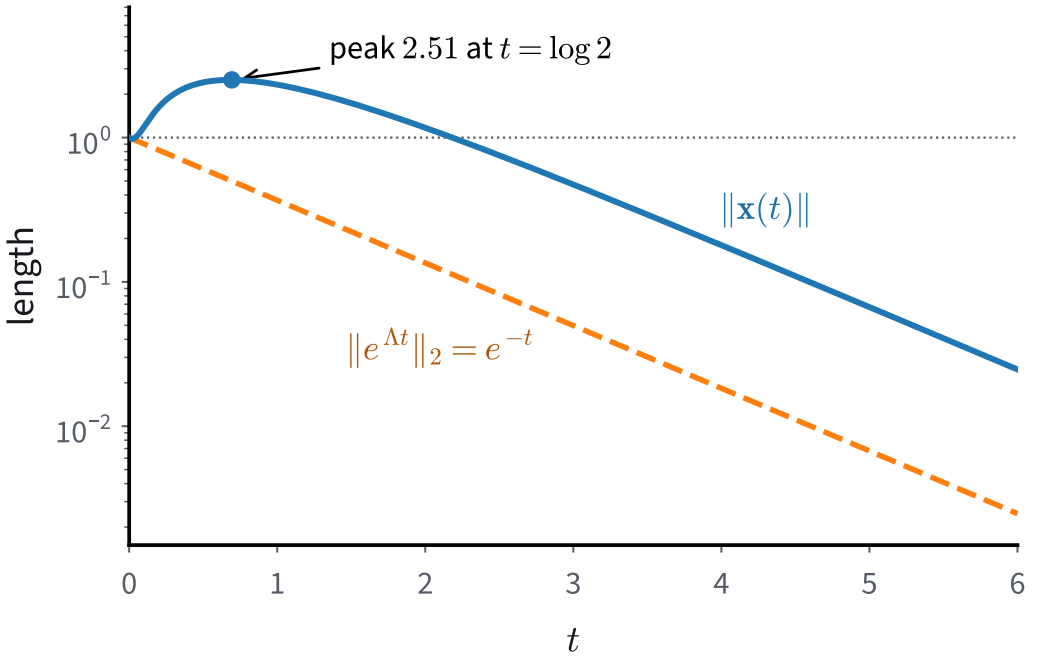


Figure 38.4: The length $\|\mathbf{x}(t)\|$ of the state of the triangular system, started at $\mathbf{x}_0 = (0, 1)$, on a logarithmic axis. It rises to 2.51 at $t = \log 2$ before it decays like $10e^{-t}$. Dashed: $\|e^{\Lambda t}\|_2 = e^{-t}$ for $\Lambda = \mathrm{diag}(-1, -2)$, the greatest length that a unit start can have at time t under the diagonal matrix with the same eigenvalues. The two curves end with the same slope, which is all that the eigenvalues determine.

The reason is the one of Section 33.4.2.3, which treats the discrete-time case in full: the eigenvectors $\mathbf{v}_1 = (1, 0)$ and $\mathbf{v}_2 = (0.995, -0.0995)$ span lines only 5.7° apart, so the short vector $\mathbf{x}_0 = 10.0\,\mathbf{v}_1 - 10.05\,\mathbf{v}_2$ is the difference of two long ones. They decay at different rates, and the cancellation stops before both have died away. Eigenvalues give the asymptotic rate of decay, not the size of the transient. This is worth remembering in front of a training curve: a momentum run that converges can grow for its first few steps (Exercise 5).

38.1.2.5 Linearization at Fixed Points

Few nonlinear fields have a closed-form solution, but near a point where the field vanishes they behave like a linear one. A **fixed point** of an autonomous field is a state $\mathbf{x}^\star$ with $\mathbf{f}(\mathbf{x}^\star) = \mathbf{0}$: a particle placed there never moves. For a continuously differentiable $\mathbf{f}$ and a small displacement $\delta(t) = \mathbf{x}(t) - \mathbf{x}^\star$, Taylor's theorem (Section 34.2) gives

$$\dot{\delta} = \mathbf{f}(\mathbf{x}^\star + \delta) = \mathbf{J}\,\delta + o(\|\delta\|), \qquad \mathbf{J} = \frac{\partial \mathbf{f}}{\partial \mathbf{x}}(\mathbf{x}^\star).$$

Near the fixed point the field is the linear field of its Jacobian $\mathbf{J}$, up to a remainder that van-

ishes faster than the displacement. May we ignore the remainder? In most cases, yes. A fixed point is **hyperbolic** if no eigenvalue of $\mathbf{J}$ has zero real part. For a hyperbolic fixed point, the Hartman–Grobman theorem (Hartman 1960; Grobman 1959), which we cite without proof, says that near $\mathbf{x}^\star$ the trajectories of the nonlinear field are those of $\dot{\delta} = \mathbf{J}\delta$, up to a continuous change of coordinates. The eigenvalues of the Jacobian then decide local stability by the classification above.

For gradient descent in continuous time, $\dot{\mathbf{x}} = -\nabla \mathcal{L}(\mathbf{x})$, this is a familiar statement in new words. The fixed points are the critical points of the loss, and the Jacobian is $-\mathbf{H}$, minus the Hessian. A minimum with positive curvature in every direction is therefore a stable node, and a saddle point of the loss is a saddle of the flow.

The damped pendulum is the one nonlinear field we work out. In units of time that make the ratio g/ℓ of gravity to length equal to one, its angle ϕ from the bottom obeys $\ddot{\phi} = -\sin\phi - c\dot{\phi}$, with damping coefficient $c \geq 0$. With the angular velocity $\omega = \dot{\phi}$ it becomes a first-order system, written here with its Jacobian:

$$\dot{\phi} = \omega, \qquad \dot{\omega} = -\sin\phi - c\,\omega, \qquad \mathbf{J}(\phi, \omega) = \begin{pmatrix} 0 & 1 \\ -\cos\phi & -c \end{pmatrix}.$$

The fixed points have $\omega = 0$ and $\sin\phi = 0$: the pendulum hanging at $\phi = 0$ and balanced upright at $\phi = \pi$.

- At the bottom, $\mathbf{J}$ has characteristic polynomial $\lambda^2 + c\lambda + 1$, with roots $\lambda = -c/2 \pm \mathrm{i}\sqrt{1 - c^2/4}$ for $c < 2$. At $c = 0.5$ they are $-0.25 \pm 0.97\,\mathrm{i}$, a stable spiral. This is the ring-down of a released pendulum: it swings with period about $2\pi/0.97$ while its amplitude decays like $e^{-t/4}$.
- At the top, the polynomial is $\lambda^2 + c\lambda - 1$, whose roots $(-c \pm \sqrt{c^2+4})/2$ have product -1, so one is positive and one negative for every c. At $c = 0.5$ they are 0.78 and -1.28: a saddle. Only the starts on one curve approach the top. They are the pendulum swung up with just the right speed, which comes to rest balanced like a pencil on its tip.

The same computation, a Jacobian and its eigenvalues, classifies the fixed points of any smooth field, with one exception. When no eigenvalue has positive real part but one lies on the imaginary axis, the linearization does not decide stability. The fields $\dot{x} = -x^3$ and $\dot{x} = x^3$ both have Jacobian 0 at the origin. The first is stable, with $x(t) = x_0/\sqrt{1 + 2x_0^2 t} \to 0$; the second escapes to infinity at $t = 1/(2x_0^2)$.

We could tell the two apart only because we could solve them. Lyapunov's idea, from 1892, decides such cases without a solution: find a quantity that the motion can only decrease. A **Lyapunov function** for a fixed point $\mathbf{x}^\star$ is a continuously differentiable function V of the state with $V(\mathbf{x}^\star) = 0$ and $V(\mathbf{x}) > 0$ at every other point nearby, whose value does not rise along any trajectory:

$$\frac{d}{dt} V(\mathbf{x}(t)) = \nabla V(\mathbf{x}(t))^\top \mathbf{f}(\mathbf{x}(t)) \leq 0.$$

The middle expression is the chain rule, and it needs the field but no trajectory. A state that starts where V is small can never reach a place where V is larger, so it stays near $\mathbf{x}^\star$: the fixed point is stable. If the inequality is strict away from $\mathbf{x}^\star$, the nearby states converge to it. For the two cubic fields take $V(x) = x^2$, with derivative $2x\dot{x}$. Under $\dot{x} = -x^3$ this is $-2x^4$,

and the origin is stable. Under $\dot{x} = x^3$ it is $+2x^4$: now x^2 grows along every trajectory, and the origin is unstable.

We have in fact used the idea already. For the spiral the squared length $V = \|\mathbf{x}\|^2$ obeyed $\dot{V} = -V$. For gradient descent in continuous time the loss, measured from its value at an isolated minimum, is a Lyapunov function near that minimum, since $\frac{d}{dt}\mathcal{L} = \nabla\mathcal{L}^\top \dot{\mathbf{x}} = -\|\nabla\mathcal{L}\|^2$ (Exercise 6). And the pendulum has its energy $E = \frac{1}{2}\omega^2 + 1 - \cos\phi$, kinetic plus potential, which is zero at the bottom and changes at the rate

$$\dot{E} = \omega\,\dot{\omega} + \sin\phi\,\dot{\phi} = \omega(-\sin\phi - c\,\omega) + \omega\sin\phi = -c\,\omega^2 \le 0.$$

Damping can only remove energy. This holds for swings of any size, far from the bottom, where the Jacobian has nothing to say. The inequality is not strict, since the energy stands still for an instant at each turning point, where $\omega = 0$. That the damped pendulum nevertheless comes to rest takes one more argument, which we skip. The Jacobian comes with a recipe and a Lyapunov function does not. It has to be guessed, and an energy or a loss is the usual first guess.

So far a formula has done all the work. For a field given by a network there is no formula at all, and a computer can only take steps: read the arrow, move a little, read it again. How far do such steps stray from the trajectory they imitate, and can they fail where the equation does not?

38.1.3 Summary

An ordinary differential equation prescribes a velocity at every point, and the first question is whether that determines a motion. When the field is globally Lipschitz it does: one trajectory passes through each point, forward and backward, so the flow map is a bijection and a model can be defined by a field. Linear systems are solved by $e^{\mathbf{A}t} = \mathbf{V}e^{\Lambda t}\mathbf{V}^{-1}$: the real parts of the eigenvalues decide decay, the imaginary parts rotation. An input held constant over each step turns the equation into a recurrence that is exact, and stable at every step size when the equation is stable. Eigenvalues give the rate of decay in the long run, not the size of the transient, and at a hyperbolic fixed point they classify a nonlinear field through its Jacobian. Where the Jacobian does not decide, or the displacement is large, a Lyapunov function does: a quantity such as an energy or a loss that no trajectory can increase. None of this computes a trajectory when no formula exists.

38.1.4 Exercises

1. **One exponent, two failures.** Consider $\dot{x} = |x|^\alpha$ for $\alpha > 0$.
   1. For $0 < \alpha < 1$, find every solution with $x(0) = 0$, and check that $\alpha = \frac{1}{2}$ gives the fan of Figure 38.2. Which hypothesis of the Picard–Lindelöf theorem fails, and at which point?
   2. For $\alpha = 1$ the theorem applies. What is L, and what is the solution from $x(0) = 0$?
   3. For $\alpha > 1$ and $x_0 > 0$, solve the equation as the text solved $\dot{x} = x^2$ (differentiate $x^{1-\alpha}$) and find the time at which the solution reaches infinity. Which hypothesis fails now? Show that the solution from $x(0) = 0$ is unique all the same. (Two solutions both stay in $|x| \le 1$ for a short time, where the field is Lipschitz

with constant α; apply the uniqueness step of the proof on a short interval, then restart.)

4. Reverse time in the case $\alpha = \frac{1}{2}$. The equation $\dot{x} = -\sqrt{x}$ for $x \geq 0$ describes, in suitable units, the level of water in a bucket that drains through a hole in its bottom (Torricelli's law). Show that every solution reaches 0 in finite time and stays there. An empty bucket does not say when it ran dry: which statement of the corollary on the flow map fails?

2. **Separation of trajectories.** Consider a field of the form $\mathbf{f}(\mathbf{x}) = \mathbf{W}_2 \tanh(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2$, the kind of field that a Neural ODE integrates (Section 38.3).
   1. Show that $\mathbf{f}$ is Lipschitz with constant $L = \|\mathbf{W}_2\|\,\|\mathbf{W}_1\|$ in the spectral norm. Which property of tanh does the proof use? Does the bound change when tanh is replaced by ReLU?
   2. Let $\mathbf{x}(t)$ and $\mathbf{y}(t)$ be two trajectories of an autonomous field with Lipschitz constant L. Differentiate $\|\mathbf{x}(t) - \mathbf{y}(t)\|^2$ and show that
   $$e^{-Lt}\|\mathbf{x}(0) - \mathbf{y}(0)\| \leq \|\mathbf{x}(t) - \mathbf{y}(t)\| \leq e^{Lt}\|\mathbf{x}(0) - \mathbf{y}(0)\|.$$
   What does the lower bound imply for two trajectories that start at distinct points? What do the two bounds imply for the Lipschitz constants of Φ_T and of its inverse?

3. **A repeated eigenvalue.** The matrix $\mathbf{A} = \left(\begin{smallmatrix} \lambda & 1 \\ 0 & \lambda \end{smallmatrix}\right)$ with real λ has a single eigenvalue and only one eigendirection, so Equation 38.6 does not apply to it.
   1. Write $\mathbf{A} = \lambda\mathbf{I} + \mathbf{N}$ and use $\mathbf{N}^2 = \mathbf{0}$ to show that $e^{\mathbf{A}t} = e^{\lambda t}\left(\begin{smallmatrix} 1 & t \\ 0 & 1 \end{smallmatrix}\right)$. Why may the exponential be split here?
   2. From $\mathbf{x}_0 = (0, 1)$ the first coordinate is $te^{\lambda t}$. For $\lambda < 0$, find its peak and the time at which it occurs. For which λ does the factor t change the verdict that the sign of λ alone would give?
   3. Let the two eigenvalues of the matrix of Section 38.1.2.4 approach each other. For $\left(\begin{smallmatrix} -1 & 10 \\ 0 & -1-\varepsilon \end{smallmatrix}\right)$ and $\mathbf{x}_0 = (0, 1)$, compute $x_1(t)$ and its limit as $\varepsilon \to 0$. What happens to the angle between the two eigendirections in this limit, and what does that say about a matrix that cannot be diagonalized?

4. code **The zero-order hold.** Consider the scalar equation $\dot{x} = \lambda x + b\,u(t)$ with $\lambda < 0$. (The exercise on discretization rules in Section 12.4 compares the hold with three other ways to turn this equation into a recurrence.)
   1. Specialize Equation 38.8 to one dimension to obtain $\bar{a} = e^{h\lambda}$ and $\bar{b} = (e^{h\lambda} - 1)\,b/\lambda$, and find the limit of $\bar{b}$ as $\lambda \to 0$. Show that for a constant input the recurrence and the equation settle at the same value.
   2. On the samples $u(t_n)$ of a moving input, the recurrence solves the equation driven by a staircase. Argue that the staircase is, on average, the input delayed by $h/2$, and conclude that to first order in h the error at time T has size $\frac{h}{2}\left|\int_0^T e^{\lambda(T-s)}\,b\,u'(s)\,ds\right|$. Evaluate this for $\lambda = -1$, $b = 1$, $u = \sin$ and $T = 2$. How close is it to the three errors that the cell of Section 38.1.2.3 prints for $\lambda = -1$?
   3. Change the cell of that subsection so that it samples the input at the middle of each step, $u(t_n + h/2)$. How does the error for $\lambda = -1$ fall with h now, and why?

5. code **Transient growth and nonnormality.** ● The text shows one matrix with transient growth. This exercise finds how the growth depends on the angle between the eigenvectors, and meets it again in momentum.
   1. For $\mathbf{A}_\nu = \left(\begin{smallmatrix} -1 & \nu \\ 0 & -2 \end{smallmatrix}\right)$ and $\mathbf{x}_0 = (0, 1)$, show that $x_1(t) = \nu(e^{-t} - e^{-2t})$ and find

the peak of x_1 and the time at which it occurs. Find numerically the smallest ν for which $\|\mathbf{x}(t)\|$ ever exceeds $\|\mathbf{x}_0\|$.

2. Compute the angle between the two eigenvectors of $\mathbf{A}_\nu$ as a function of ν, express $\mathbf{x}_0$ in the eigenbasis, and explain the growth as the cancellation of two large components.
3. Compute $\max_t \|e^{\mathbf{A}_\nu t}\|_2$ for $\nu = 1, 5, 10, 20$ numerically and compare it with the peak of the trajectory from $(0, 1)$. Why can the first never be smaller than the second?
4. Heavy-ball momentum on $\mathcal{L}(x) = \frac{\lambda}{2}x^2$ is the recurrence $x_{k+1} = (1 + \beta - \eta\lambda)x_k - \beta x_{k-1}$, which Exercise 6 of Section 38.2 derives from a damped oscillator. It is the linear map $(x_{k+1}, x_k) = \mathbf{M}(x_k, x_{k-1})$ with $\mathbf{M} = \left(\begin{smallmatrix} 1+\beta-\eta\lambda & -\beta \\ 1 & 0 \end{smallmatrix}\right)$. Show that $\mathbf{M}$ is not normal, that is, $\mathbf{M}\mathbf{M}^\top \neq \mathbf{M}^\top\mathbf{M}$ (normal matrices have orthogonal eigenvectors and no transient growth, Section 33.4.2.3). For $\beta = 0.9$, compute its eigenvalues at $\eta\lambda = 0.1$ and at $\eta\lambda = 3$, and check that their modulus is $\sqrt{\beta} \approx 0.95$ in both cases. Run both from rest, $x_0 = x_{-1} = 1$. In which run does $\|(x_{k+1}, x_k)\|$ grow before it decays, for how many steps and to what peak, and what happens to the loss in its first step? What does this say about reading the first iterations of a momentum run as divergence?

6. **A gradient flow cannot circle.** Let $\mathcal{L}$ be twice continuously differentiable and bounded below, and consider the gradient flow $\dot{\mathbf{x}} = -\nabla\mathcal{L}(\mathbf{x})$.
   1. Show that $\frac{d}{dt}\mathcal{L}(\mathbf{x}(t)) = -\|\nabla\mathcal{L}(\mathbf{x}(t))\|^2$. Conclude that no trajectory returns to a point it has left, so that a gradient flow has no closed orbit and no center.
   2. Integrate the identity from 0 to T and show that $\min_{t\le T}\|\nabla\mathcal{L}(\mathbf{x}(t))\|^2 \le (\mathcal{L}(\mathbf{x}_0) - \inf\mathcal{L})/T$. Which assumptions on the loss did this bound not need?
   3. The game $\min_{x_1}\max_{x_2} x_1x_2$, in which x_1 descends and x_2 ascends the value x_1x_2, has the field $\dot{x}_1 = -x_2$, $\dot{x}_2 = x_1$: the rotation $\mathbf{K}$. Show that this field is not the negative gradient of any function. (Compare $\partial f_1/\partial x_2$ with $\partial f_2/\partial x_1$.) What does part (a) then say about the difference between training one loss and training two players against each other?
7. **Limits of linearization.**
   1. The field $\dot{x} = x^2$ has Jacobian 0 at the origin, like the two cubic fields of the text. Its solution in Section 38.1.1.3 holds for every start x_0. Find the starts from which the state approaches the origin. Is the origin stable, unstable, or stable from one side only? What does the Hartman–Grobman theorem say about the three fields?
   2. For the damped pendulum, find the damping c at which the bottom rest point changes from a spiral to a node. Sketch the portrait near the bottom for $c = 0.5$, $c = 2$ and $c = 4$, and describe how a released pendulum moves in each case.
   3. The loss $\mathcal{L}(w_1, w_2) = \frac{1}{2}(w_1w_2 - 1)^2$ is that of the smallest deep linear network, a product of two weights fitted to the target 1. Write down its gradient flow $\dot{\mathbf{w}} = -\nabla\mathcal{L}(\mathbf{w})$, find all fixed points, and classify the origin. Show that at every point of the curve $w_1w_2 = 1$ the Jacobian has the eigenvalues 0 and $-(w_1^2 + w_2^2)$. Why can no point of a curve of fixed points be hyperbolic, and what does the negative eigenvalue still tell you?

38.2 Numerical Solvers

Anyone who has trained a network has seen a run blow up because the learning rate was too large. It is tempting to blame the loss surface, with its sharp valleys and its plateaus. Yet the same thing happens on the simplest loss there is. Gradient descent on the bowl

$\mathcal{L}(x) = 25x^2$ takes the step $x \leftarrow x - \eta \cdot 50x$, and for every learning rate above $\eta = 0.04$ the iterates grow without bound, although the bowl has a single minimum and nothing about it is difficult. If the loss is not the problem, what is?

The answer comes from an older subject. Gradient descent is a step-by-step solution of the differential equation $\dot{x} = -50x$, which says: move downhill at a speed proportional to the slope. This is the decay equation solved in Section 38.1, fifty times faster, and its exact solution $x(t) = e^{-50t}x_0$ decays to zero from every start. Euler proposed in 1768 to solve such an equation by following the arrow for a short time h and then reading it again, and the learning rate plays the role of h. One step multiplies x by $1 - 50h$. With $h = 0.01$ the factor is 0.5, and the computed solution decays like the true one. With $h = 0.05$ it is -1.5, where the exact factor is $e^{-2.5} = 0.08$, and the computed solution flips sign and grows. The equation is stable; the steps are not. In the previous section the eigenvalues of an equation gave its exact motion. Here the same eigenvalues decide how large a step can follow that motion.

Deep learning meets this gap between an equation and its steps wherever it takes steps: in gradient descent (Section 9.1), in the layers of a residual network (Section 7.4), and in the samplers of diffusion and flow-matching models (Section 17.5). A sampler evaluates a network once or twice in every step, so the number of steps sets the cost of an image.

Three questions decide what a step-by-step solution is worth. How far does it stray from the true one? How can it stray less for the same work? And how long may a step be before the errors grow instead of dying out? It turns out that one object answers all three: the factor by which a step multiplies the solution, which for Euler on the bowl was $1 - 50h$. At the end we return to gradient descent and ask what a better solver would do for training.

38.2.1 Forward Euler

A trajectory is its start plus the velocity accumulated on the way. Over a short step, Euler's proposal is the crudest one possible: pretend that the velocity keeps the value it has at the start. Let us first see what this does on an equation whose answer we know.

Take $\dot{x} = x$ from $x_0 = 1$ to $T = 1$, where the answer is $e = 2.718$. Divide the interval into n steps of length $h = 1/n$, and in each step move with the velocity at its start: $x \leftarrow x + hx = (1 + 1/n)x$. After n steps, $x_n = (1 + 1/n)^n$. This is a dollar at 100% annual interest compounded n times a year:

n	1	2	4	8	16	32	365
$(1 + 1/n)^n$	2.000	2.250	2.441	2.566	2.638	2.677	2.7146
error	0.718	0.468	0.277	0.153	0.080	0.041	0.0037

The ratios of consecutive errors are $1.53, 1.69, 1.82, 1.90, 1.95$: doubling the number of steps comes closer and closer to halving the error. Even daily compounding falls more than a third of a cent short of e dollars.

We have met this slow limit before. The compounding $(\mathbf{I} + t\mathbf{A}/n)^n$ of Section 38.1 was Euler's method before it had a name: n steps of length t/n along a linear field. The expansion found there, $(1 + \lambda t/n)^n \approx e^{\lambda t}(1 - \frac{(\lambda t)^2}{2n})$, gives the dollar's error with $\lambda t = 1$. It is about $e/(2n) = 1.36\,h$. Error proportional to h is what we will call order one.

For a general field $\mathbf{f}(\mathbf{x}, t)$, with step size h and grid $t_n = nh$, where n from here on counts the steps taken so far, **forward Euler** is

$$\mathbf{x}_{n+1} = \mathbf{x}_n + h\,\mathbf{f}(\mathbf{x}_n, t_n). \tag{38.9}$$

The method follows the arrow at the current point for time h, then reads the field again. Equivalently, it is the Taylor expansion of the true solution cut after the linear term, and the error of one step is the first term dropped. If $\|\ddot{\mathbf{x}}(t)\| \le M$ along the solution, then

$$\mathbf{x}(t_{n+1}) = \mathbf{x}(t_n) + h\,\mathbf{f}(\mathbf{x}(t_n), t_n) + \mathbf{r}_n, \qquad \|\mathbf{r}_n\| \le \tfrac{Mh^2}{2}.$$

The remainder $\mathbf{r}_n$ is the **local error**: what one step loses when it starts on the true solution. What we want to know is the **global error**, the distance from the true solution after all the steps to a fixed horizon T. A local error of order h^2 does not give a global error of order h^2. Reaching T takes T/h steps, their local errors accumulate, and each step may amplify the error already made. One power of h is lost. The proof is short, and its structure matters more than its constant.

Proposition (Euler converges at order 1). *Let* $\mathbf{f}$ *be* L*-Lipschitz in* $\mathbf{x}$*, that is* $\|\mathbf{f}(\mathbf{x}, t) - \mathbf{f}(\mathbf{y}, t)\| \le L\|\mathbf{x} - \mathbf{y}\|$ *for all* $\mathbf{x}$*,* $\mathbf{y}$ *and* t*, and let the solution of* $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, t)$ *from* $\mathbf{x}(0) = \mathbf{x}_0$ *satisfy* $\|\ddot{\mathbf{x}}(t)\| \le M$ *on* $[0, T]$*. Then the Euler iterates Equation 38.9 from the same start obey*

$$\max_{n \le T/h} \|\mathbf{x}_n - \mathbf{x}(t_n)\| \;\le\; \frac{Mh}{2L}\,(e^{LT} - 1) \;=\; O(h).$$

Proof. Let $\varepsilon_n = \|\mathbf{x}_n - \mathbf{x}(t_n)\|$, and subtract the Taylor identity above from the update Equation 38.9:

$$\varepsilon_{n+1} \le \underbrace{\|\mathbf{x}_n - \mathbf{x}(t_n)\| + h\,\|\mathbf{f}(\mathbf{x}_n, t_n) - \mathbf{f}(\mathbf{x}(t_n), t_n)\|}_{\le (1 + hL)\,\varepsilon_n \text{ by Lipschitz}} + \underbrace{\|\mathbf{r}_n\|}_{\le Mh^2/2}.$$

Each step multiplies the existing error by at most $1 + hL$ and adds at most $Mh^2/2$. Unrolling the recursion from $\varepsilon_0 = 0$ sums a geometric series:

$$\varepsilon_n \le \frac{Mh^2}{2} \sum_{j=0}^{n-1} (1 + hL)^j = \frac{Mh^2}{2} \cdot \frac{(1 + hL)^n - 1}{hL} \le \frac{Mh}{2L}\,(e^{LT} - 1)\,,$$

using $(1 + hL)^n \le e^{nhL} \le e^{LT}$. ■

For $\dot{x} = x$ on $[0, 1]$ we have $M = \max |\ddot{x}| = e$ and $L = 1$, so the bound is $\frac{e}{2}(e - 1)h = 2.3\,h$, against the true error $1.36\,h$: the right order, with a pessimistic constant. The structure of the bound recurs for every solver. A local error of order h^{p+1} per step becomes a global error of order h^p after T/h steps, inflated by the factor e^{LT} that measures how strongly the dynamics amplify old mistakes. A method with global error of order h^p has **order** p; Euler has order 1.

The factor e^{LT} is the crude part. The constant L records how steep the field is and not which way it leans, so the bound cannot tell a field that drives trajectories apart from one that pulls them together. For the bowl's equation $\dot{x} = -50x$ on $[0, 1]$ the factor is $e^{50} \approx 5 \times 10^{21}$, useless at any step a computer could take, although small steps follow the decay well. A field that contracts needs an analysis of its own, and we come to it after raising the order.

38.2.2 Runge–Kutta Methods

Euler commits to the slope at the start of the step while the trajectory curves away from it. Runge–Kutta methods measure the slope at points inside the step and combine the measurements. Figure 38.5 shows one step of size $h = 1$ on $\dot{x} = x$ from $x = 1$.

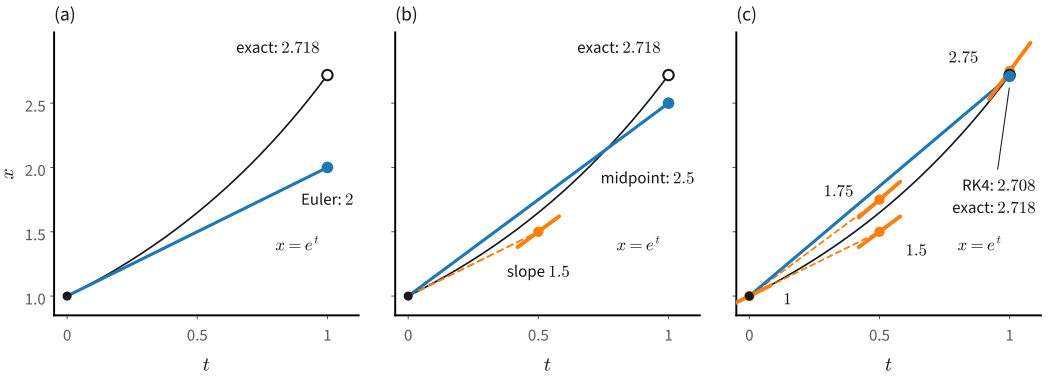


Figure 38.5: One step of size $h = 1$ on $\dot{x} = x$ from $x = 1$. (a) Euler follows the starting slope and lands at 2. (b) The midpoint method takes a half step only to measure the slope there, 1.5, and lands at 2.5. (c) RK4 measures four slopes, 1, 1.5, 1.75 and 2.75, averages them with weights $\frac{1}{6}(1, 2, 2, 1)$ and lands at 2.708; the exact value is $e = 2.718$.

38.2.2.1 Midpoint, Heun and RK4

The simplest improvement, the **midpoint method**, takes half an Euler step only to measure the slope there, and then takes the real step with that slope. Writing $\mathbf{f}(\mathbf{x})$ for a field that does not depend on time, it is $\mathbf{x}_{n+1} = \mathbf{x}_n + h\,\mathbf{f}\big(\mathbf{x}_n + \frac{h}{2}\mathbf{f}(\mathbf{x}_n)\big)$. One Taylor line in one dimension shows why it helps:

$$x + hf\big(x + \tfrac{h}{2}f(x)\big) = x + hf + \tfrac{h^2}{2}f'f + O(h^3).$$

The true expansion is $x(t + h) = x + h\dot{x} + \frac{h^2}{2}\ddot{x} + O(h^3)$ with $\ddot{x} = \frac{d}{dt}f(x(t)) = f'f$, so the h^2 term is now right instead of missing. The local error is $O(h^3)$ and the global order 2, for two field evaluations per step.

The midpoint method spends its first evaluation on a half step. **Heun's method** takes a full trial Euler step instead and averages the slopes at the two ends of the step:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \tfrac{h}{2}\Big[\mathbf{f}(\mathbf{x}_n) + \mathbf{f}\big(\mathbf{x}_n + h\,\mathbf{f}(\mathbf{x}_n)\big)\Big]. \tag{38.10}$$

If the midpoint method is the midpoint rule for an integral, this is the trapezoid rule. The same Taylor line gives it the same order 2, at the same two evaluations. Heun's method is the one to remember of the two, because many diffusion samplers use it (Section 17.5).

Four slope measurements give the standard general-purpose method of scientific computing, **classical Runge–Kutta (RK4)**, which goes back to Kutta in 1901:

$$\begin{aligned}
\mathbf{k}_1 &= \mathbf{f}(\mathbf{x}_n,\, t_n),\\
\mathbf{k}_2 &= \mathbf{f}(\mathbf{x}_n + \tfrac{h}{2}\mathbf{k}_1,\, t_n + \tfrac{h}{2}),\\
\mathbf{k}_3 &= \mathbf{f}(\mathbf{x}_n + \tfrac{h}{2}\mathbf{k}_2,\, t_n + \tfrac{h}{2}),\\
\mathbf{k}_4 &= \mathbf{f}(\mathbf{x}_n + h\,\mathbf{k}_3,\, t_n + h),\\
\mathbf{x}_{n+1} &= \mathbf{x}_n + \tfrac{h}{6}\left(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4\right).
\end{aligned} \tag{38.11}$$

The update averages four slope measurements, weighting the two at the midpoint twice. These are the weights of Simpson's rule, $\int_a^b g \approx \frac{b-a}{6}(g(a) + 4g(\frac{a+b}{2}) + g(b))$, which integrates cubics exactly, with the midpoint's share split between two measurements.

38.2.2.2 The Stability Function

Why order 4? On a linear field the answer takes one computation. Consider the scalar equation $\dot{y} = \lambda y$, with λ real or complex. It is known as the *test equation*, and it stands for every linear field at once: each mode of $\dot{\mathbf{x}} = \mathbf{A}\mathbf{x}$ obeys it, with λ an eigenvalue of $\mathbf{A}$ (Section 38.1). Its true solution gains the factor $e^{h\lambda}$ over a time h. Each method multiplies y by a fixed function of $z = h\lambda$ per step, and we can compare that function with e^z.

Proposition (stability functions). *On the test equation, with $z = h\lambda$, one step of each method multiplies y by its* **stability function** $R(z)$*:*

$$\begin{aligned}
R_{\text{Euler}}(z) &= 1 + z, \qquad R_{\text{mid}}(z) = R_{\text{Heun}}(z) = 1 + z + \frac{z^2}{2},\\
R_{\text{RK4}}(z) &= 1 + z + \frac{z^2}{2} + \frac{z^3}{6} + \frac{z^4}{24}.
\end{aligned} \tag{38.12}$$

Proof. For RK4 the four measurements are $k_1 = \lambda y$, $k_2 = \lambda y(1 + \frac{z}{2})$, $k_3 = \lambda y(1 + \frac{z}{2} + \frac{z^2}{4})$ and $k_4 = \lambda y(1 + z + \frac{z^2}{2} + \frac{z^3}{4})$, each obtained by substituting the previous one. The update is $y + \frac{z}{6}\left(6 + 3z + z^2 + \frac{z^3}{4}\right) y$, which is the stated polynomial times y. For Euler, $y + zy$. For the midpoint method, $y + h\lambda(y + \frac{z}{2}y)$, and for Heun, $y + \frac{z}{2}(y + (1 + z)y)$; both are $(1 + z + \frac{z^2}{2})y$. ■

Each R is the Taylor polynomial of the exact factor e^z, cut at the method's order. For RK4, $e^z - R_{\text{RK4}}(z) = z^5/120 + O(z^6)$, so on this equation one step errs by $O(h^5)$ and T/h steps by $O(h^4)$. For nonlinear fields that are four times continuously differentiable and Lipschitz in $\mathbf{x}$, a longer Taylor match gives the same orders: midpoint and Heun have order 2, RK4 order 4 (Hairer et al. 1993). We do not prove this; the expansion of four nested evaluations fills pages and teaches little. The stability function is the object to keep from this section. How closely it follows e^z near $z = 0$ is the accuracy of a method, and how large it becomes far from zero will decide which steps are stable.

38.2.2.3 Order at Equal Cost

It remains to measure the orders, and to compare the methods at the same number of field evaluations. In code the two ends of the family take a few lines each, for a field `f` that does not depend on time.

```
def euler(f, x0, T, n):
    """Forward Euler: n steps of size h = T/n."""
    x, h = np.asarray(x0, dtype=float), T / n
    for _ in range(n):
        x = x + h * f(x)
    return x

def rk4(f, x0, T, n):
    """Classical Runge-Kutta: four slope measurements per step."""
    x, h = np.asarray(x0, dtype=float), T / n
    for _ in range(n):
        k1 = f(x)
        k2 = f(x + 0.5 * h * k1)
        k3 = f(x + 0.5 * h * k2)
        k4 = f(x + h * k3)
        x = x + (h / 6) * (k1 + 2 * k2 + 2 * k3 + k4)
    return x
```

On log–log axes, error $\approx Ch^p$ is a line of slope p, so the measured slopes should be 1 and 4. We test this on the spiral $\dot{\mathbf{x}} = \mathbf{A}\mathbf{x}$ of Figure 38.1, whose eigenvalues are $-0.5 \pm \mathrm{i}$. A point on it loses length at rate $\frac{1}{2}$ while it turns at unit speed, so from $\mathbf{x}_0 = (2, 0)$ the exact state at $T = 2$ is $e^{-1}(2\cos 2,\ 2\sin 2) = (-0.306,\ 0.669)$, of length 0.736 (Section 38.1).

```
A = np.array([[-0.5, -1.0], [1.0, -0.5]])      # the spiral, eigenvalues -0.5 +- i
x0, T = np.array([2.0, 0.0]), 2.0
x_true = np.exp(-1) * np.array([2 * np.cos(2), 2 * np.sin(2)])  # e^{AT} x0
```

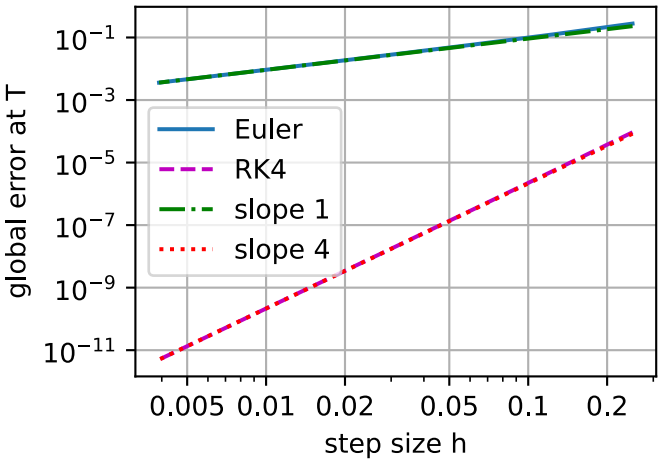
On a linear field we can predict the constants as well as the slopes. On the spiral a step of either solver is a rotation combined with a rescaling, as the exact motion is. A solver whose factor is off by some fraction in each mode is therefore off by the same fraction of the length 0.736 of the true endpoint. For Euler the fraction is the one found for the compounding, $|\lambda T|^2/(2n)$. Here $|\lambda T|^2 = 5$, so Euler should miss by $2.5/n$ of the length, that is by $1.84/n$.

For RK4 the fraction comes from the stability function. One step is off by the fraction $|z|^5/120$, with $|z| = |\lambda T|/n = \sqrt{5}/n$, and the errors of n steps add. The sum is $5^{5/2}/(120\,n^4)$ of the length, or $0.343/n^4$. The cell halves the step six times, from 8 steps to 512.

```
f = lambda x: A @ x                            # the spiral field
ns = 2 ** np.arange(3, 10)                     # 8, 16, ..., 512 steps
hs = T / ns
err_eu = np.array([np.linalg.norm(euler(f, x0, T, n) - x_true) for n in ns])
err_rk = np.array([np.linalg.norm(rk4(f, x0, T, n) - x_true) for n in ns])
print('measured slope, Euler:', f'{np.polyfit(np.log(hs), np.log(err_eu), 1)[0]:.3f}')
print('measured slope, RK4  :', f'{np.polyfit(np.log(hs), np.log(err_rk), 1)[0]:.3f}')
```

```
print(f'error at 512 steps: Euler {err_eu[-1]:.1e}, RK4 {err_rk[-1]:.1e}')
print('RK4 with 128 steps (the same 512 evaluations):',
      f'{np.linalg.norm(rk4(f, x0, T, 128) - x_true):.1e}')
d2l.plot(hs, [err_eu, err_rk, err_eu[-1] * (hs / hs[-1]),
              err_rk[-1] * (hs / hs[-1]) ** 4],
         'step size h', 'global error at T', xscale='log', yscale='log',
         legend=['Euler', 'RK4', 'slope 1', 'slope 4'])
```

```
measured slope, Euler: 1.040
measured slope, RK4  : 4.022
error at 512 steps: Euler 3.6e-03, RK4 5.0e-12
RK4 with 128 steps (the same 512 evaluations): 1.3e-09
```



The fitted slopes are 1.04 and 4.02, a little above 1 and 4 because the fit includes steps as coarse as $h = 0.25$, where the next terms of the expansions still count. At 512 steps the errors are 3.6×10^{-3} and 5.0×10^{-12}, which is what $1.84/n$ and $0.343/n^4$ predict. The slopes hide the cost, though. RK4 reads the field four times per step, so a fair comparison gives it a quarter of the steps. With the same 512 field evaluations, that is 128 steps, RK4 reaches 1.3×10^{-9}, six orders of magnitude below Euler. To match that, Euler's law $1.84/n$ asks for more than a billion steps. In general a method of order p needs a number of steps proportional to $\varepsilon^{-1/p}$ for an error ε.

Practical solvers also choose the step size as they go. An *embedded pair* such as Dormand–Prince computes solutions of orders 4 and 5 from the same evaluations and takes their difference as an estimate of the local error (Dormand and Prince 1980). The solver then shortens the step where the estimate is large and lengthens it where the estimate is small. Exercise 1 builds the simplest such controller.

All these methods start every step afresh and discard the slopes that the step before has measured. A *multistep* method keeps them. The simplest one, the two-step **Adams–Bashforth method**, reads the field once per step and extrapolates from the last two readings:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + h\left[\tfrac{3}{2}\,\mathbf{f}(\mathbf{x}_n) - \tfrac{1}{2}\,\mathbf{f}(\mathbf{x}_{n-1})\right]. \tag{38.13}$$

The bracket is $\mathbf{f}(\mathbf{x}_n) + \frac{1}{2}[\mathbf{f}(\mathbf{x}_n) - \mathbf{f}(\mathbf{x}_{n-1})]$, the line through the two readings followed for half a step more. It is an estimate of the slope at the middle of the coming step, so this is the midpoint method with the midpoint slope taken from memory and not from a second evaluation: order 2 at the cost of Euler. Two things are lost. The first step has no predecessor and must be taken by a one-step method, and the weights $\frac{3}{2}$ and $-\frac{1}{2}$ hold for equal steps only and must be recomputed when the step size changes. Where one evaluation is a pass through a large network, the saving outweighs both, and the multistep samplers of Section 17.5 are built on this step.

38.2.3 Stiffness and Implicit Methods

Order measures how accurate a step is when it is small. Stability asks a different question: whether the errors of many steps die out or compound. For Euler and RK4 the answer caps the step size, however little accuracy we need.

The smallest system that shows the problem has two decoupled modes, $\dot{\mathbf{x}} = \operatorname{diag}(-50, -1)\,\mathbf{x}$ from $(1, 1)$, followed to $T = 1$. By then the fast mode has decayed to $e^{-50} \approx 10^{-22}$ and the slow one stands at $e^{-1} = 0.368$. After $t \approx 0.1$ only the slow mode is left, and steps of $h = 0.1$ would follow it well. Yet the fast coordinate is the opening's equation $\dot{x} = -50x$, and Euler multiplies it by $1 - 50h$ at every step for as long as we integrate. A system is **stiff** when some Jacobian eigenvalues have large negative real parts compared with the rate at which the solution of interest varies: a mode that is long gone still sets the step.

Which steps are allowed? The stability function answers. On the test equation $\dot{y} = \lambda y$, whose solution decays if and only if $\operatorname{Re}\lambda < 0$, a method turns y_0 into $R(z)^n y_0$ after n steps. The computed mode therefore decays if and only if $|R(z)| < 1$. For forward Euler this reads $|1 + z| < 1$, and for a real $\lambda < 0$ it is $-2 < h\lambda < 0$, or

$$h < \frac{2}{|\lambda|}.$$

For the bowl, $2/|\lambda| = 2/50 = 0.04$ is the learning rate above which the iterates of the opening grew.

Below the limit we also get what the error bound of Euler's method lacked, an analysis for a field that contracts. A step now multiplies the error already made by $|1 + h\lambda| < 1$, where the general proof could only bound this factor by $1 + hL$. For $h \le 1/|\lambda|$ the geometric sum of that proof stays below $1/(h|\lambda|)$, and the bound becomes $Mh/(2|\lambda|)$: the old bound without its exponential. For the bowl's equation from $x_0 = 1$, where $M = 2500$, that is $25\,h$, within a factor of three of the true error.

38.2.3.1 Backward Euler

A shorter step is one way out. The other looks odd at first: take Euler's step with the slope at the *end* of the step,

$$\mathbf{x}_{n+1} = \mathbf{x}_n + h\,\mathbf{f}(\mathbf{x}_{n+1}, t_{n+1}).$$

This is **backward Euler**. The unknown $\mathbf{x}_{n+1}$ stands on both sides, a nuisance we postpone until we have seen what the method achieves.

Proposition (backward Euler is stable at every step). *On the test equation with* $\operatorname{Re}\lambda < 0$*, backward Euler has the stability function* $R(z) = 1/(1-z)$*, and* $|R(z)| < 1$ *for every step* $h > 0$*.*

Proof. One step is $y_{n+1} = y_n + h\lambda y_{n+1}$, so $y_{n+1} = y_n/(1-z)$, and $|1-z| \geq \operatorname{Re}(1-z) = 1 - \operatorname{Re} z > 1$ when $\operatorname{Re} z < 0$. ■

The reason can be seen without the algebra. Forward Euler extrapolates with the slope at the point where it stands, and on a steep mode that slope, held for a long step, carries it far past zero. Backward Euler asks for the point whose own slope leads back to where we are. A point that has almost decayed has almost no slope, so no step is long enough to overshoot.

Now for the nuisance: the field is evaluated at a point that is not yet known, so each step is an equation to solve, a linear system when the field is linear and a few Newton iterations otherwise (Exercise 3). Such methods are called *implicit*; Euler and RK4, which compute the new state by a formula, are *explicit*. An implicit step is worth its solve when stability, not accuracy, would set the step of an explicit method.

We now run both methods on the two-mode system at five step sizes, backward Euler with one linear solve per step. The reader can predict the first row: at $h = 0.1$ forward Euler multiplies the fast coordinate by $1 - 5 = -4$ ten times.

```
def backward_euler_linear(A, x0, T, n):
    """Backward Euler on dx/dt = A x: solve (I - h A) x_new = x at each step."""
    x, h = np.asarray(x0, dtype=float), T / n
    for _ in range(n):
        x = np.linalg.solve(np.eye(len(x)) - h * A, x)
    return x

A_stiff = np.diag([-50.0, -1.0])              # one fast mode, one slow mode
x0_stiff = np.array([1.0, 1.0])
T = 1.0
print(f'exact at T = 1: fast {np.exp(-50):.1e}, slow {np.exp(-1):.4f}')
print(f'{"h":>5} {"fwd fast":>10} {"fwd slow":>9} {"bwd fast":>10} {"bwd slow":>9}')
for n in [10, 20, 25, 50, 100]:
    fwd = euler(lambda x: A_stiff @ x, x0_stiff, T, n)
    bwd = backward_euler_linear(A_stiff, x0_stiff, T, n)
    print(f'{T / n:5.2f} {fwd[0]:10.1e} {fwd[1]:9.4f} {bwd[0]:10.1e} {bwd[1]:9.4f}')
```

```
exact at T = 1: fast 1.9e-22, slow 0.3679
    h   fwd fast  fwd slow   bwd fast  bwd slow
 0.10    1.0e+06    0.3487    1.7e-08    0.3855
 0.05    3.3e+03    0.3585    1.3e-11    0.3769
 0.04   -1.0e+00    0.3604    1.2e-12    0.3751
 0.02    0.0e+00    0.3642    8.9e-16    0.3715
 0.01    7.9e-31    0.3660    2.5e-18    0.3697
```

At $h = 0.1$ both methods have the slow mode within about 5% of the truth (0.349 and 0.386 against 0.368), accurate enough for many purposes. Forward Euler nonetheless returns $4^{10} \approx 10^6$ for the fast component, whose true value is 10^{-22}. At $h = 0.05$ its factor is the opening's -1.5, applied twenty times. At $h = 0.04$ the factor is -1, and the fast component flips sign at every step and never shrinks. (The zero in the row $h = 0.02$ is exact: there $1 - 50h = 0$, and one step removes the fast mode.) Forward Euler needs $h < 0.04$, a step

at which its slow mode is already within about 2%. We asked for 5%, and stability made us compute 2%.

Backward Euler is stable at every step size. Its slow-mode error is the $O(h)$ error of a first-order method, so stability does not bring accuracy.

38.2.3.2 Stability Regions

Forward Euler has a step-size limit and backward Euler has none. Where do other methods stand? A plot of the z plane shows all of them at once. The **stability region** of a method is the set $\{z : |R(z)| < 1\}$ where its update decays. A method is **A-stable** if its region contains the whole open left half-plane Re $z < 0$: it makes every decaying mode decay, whatever the step (Dahlquist 1963). Backward Euler is A-stable, since its region is everything outside the closed unit disc around $+1$.

Figure 38.6 marks the examples. The opening's step $h = 0.05$ on $\dot{x} = -50x$ is the point $z = -2.5$, outside Euler's disc, and $h = 0.01$ is $z = -0.5$, inside. The spiral, with $\lambda = -0.5 \pm \mathrm{i}$, gives $|1 + z|^2 = (1 - h/2)^2 + h^2$, which is below 1 if and only if $h < 0.8$; at $h = 0.8$ its two points $-0.4 \pm 0.8\,\mathrm{i}$ lie on the circle. A pure rotation puts z on the imaginary axis, which touches Euler's disc only at the origin, so forward Euler makes an undamped oscillation grow at every step size (Exercises 2 and 4). The exact step e^z of Section 38.1.2.3 has modulus below one on the whole left half-plane, which is the behaviour backward Euler imitates and forward Euler does not.

No explicit Runge–Kutta method is A-stable, however many stages it has. Its R is a polynomial (Exercise 2), and a polynomial other than a constant grows without bound as z runs out along the negative real axis. Higher order therefore does not cure stiffness. RK4's region ends on the real axis at $z = -2.79$, the negative root of $R_{\text{RK4}}(z) = 1$. On the bowl's equation it allows $h < 2.79/50 = 0.056$ where Euler allows 0.04, and it reads the field four times per step. This is why implicit methods are kept for stiff fields, while explicit ones are the default everywhere else. The stiff field that readers of this book meet most often is the gradient of a loss whose curvature is large in some directions and small in others.

38.2.4 Gradient Descent as a Solver

The opening treated one coordinate. With several, the same factor appears once per eigenvalue of the Hessian. Moving downhill continuously on a loss $\mathcal{L}$, always along the negative gradient, is the **gradient flow**

$$\dot{\mathbf{x}} = -\nabla \mathcal{L}(\mathbf{x}),$$

a field that vanishes at the critical points of $\mathcal{L}$ and along whose trajectories the loss only falls: $\frac{d}{dt}\mathcal{L}(\mathbf{x}(t)) = \nabla\mathcal{L}^{\top}\dot{\mathbf{x}} = -\|\nabla\mathcal{L}\|^2 \le 0$. Gradient descent with learning rate η, $\mathbf{x}_{k+1} = \mathbf{x}_k - \eta\,\nabla\mathcal{L}(\mathbf{x}_k)$, is forward Euler Equation 38.9 on this flow with $h = \eta$.

Section 35.1 derives the learning-rate ceiling $\eta < 2/\lambda_{\max}(\mathbf{H})$ on a quadratic with Hessian $\mathbf{H}$ and studies the speed below it. We do not repeat the study of the speed. The ceiling we derive once more, in three arrows, because the route is what the solver view adds. Write $\delta = \mathbf{x} - \mathbf{x}^{\star}$ for the displacement from a minimum $\mathbf{x}^{\star}$ with Hessian $\mathbf{H}$. Near the minimum the flow linearizes, and each eigenvalue λ_i of $\mathbf{H}$ becomes a decaying mode with rate $-\lambda_i$, a copy of the test equation:

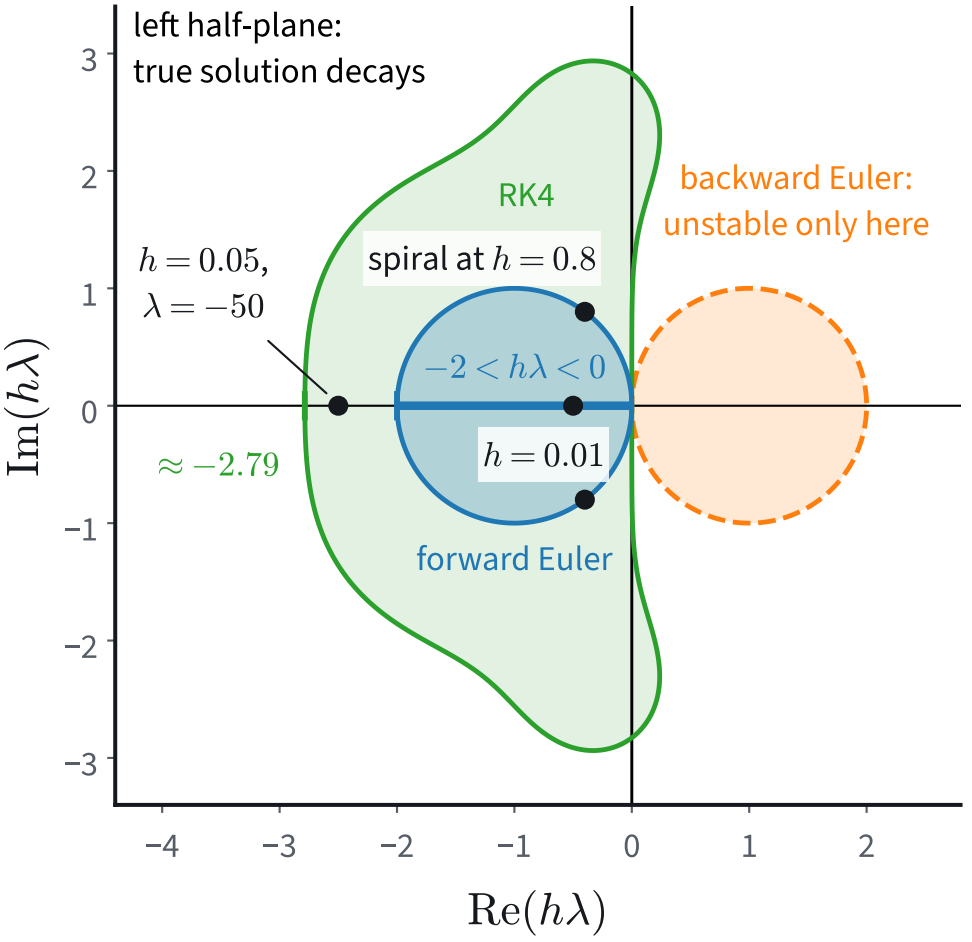


Figure 38.6: Stability regions in the complex plane of $z = h\lambda$ for the test equation $\dot{y} = \lambda y$; a decaying mode sits in the left half-plane. Forward Euler decays only inside the disc $|1 + z| < 1$, whose real slice is $-2 < h\lambda < 0$; RK4 enlarges it to a lobed region reaching ≈ -2.79 on the real axis; backward Euler is stable everywhere except the disc around $+1$, in particular on the entire left half-plane, which is A-stability. The marked points are the opening's $h = 0.05$ step, its $h = 0.01$ step, and the spiral at $h = 0.8$, on the edge of Euler's disc.

$$\dot{\mathbf{x}} = -\nabla\mathcal{L}(\mathbf{x}) \xrightarrow{\text{linearize}} \dot{\delta} = -\mathbf{H}\delta \xrightarrow{\text{Euler, } h=\eta} \delta_{k+1} = (\mathbf{I} - \eta\mathbf{H})\,\delta_k \xrightarrow{\text{stability}} \eta < \frac{2}{\lambda_{\max}(\mathbf{H})}.$$

The learning-rate ceiling is thus the boundary of Euler's stability disc, reached by the steepest mode at $z = -\eta\lambda_{\max} = -2$. The first arrow holds the Hessian fixed, and so the ceiling is a statement about a quadratic. On a network the Hessian moves during training, and a run above the ceiling often does not blow up: its largest eigenvalue comes back toward $2/\eta$, and the loss keeps falling over the long run, though not at every step (Section 35.1.2.4).

If this reading is right, the ceiling comes from Euler's method, and another solver must have another ceiling. Let us test this prediction on the valley on which Section 35.1 sweeps the learning rate, $\mathcal{L}(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top\mathbf{H}\mathbf{x}$ with $\mathbf{H} = \mathrm{diag}(10, 1)$ (that section lists the flat coordinate first). Its gradient flow $\dot{\mathbf{x}} = -\mathbf{H}\mathbf{x}$ has a steep mode with rate -10 and a flat one with rate -1. From $(1, 1)$ we take 100 steps of size η with three solvers. Forward Euler is gradient descent, with its ceiling at $2/10 = 0.2$. RK4 reads the gradient four times per step, and since its region reaches -2.79 on the real axis (Figure 38.6), its ceiling should be 0.279. Backward Euler, $\mathbf{x}_{k+1} = (\mathbf{I} + \eta\mathbf{H})^{-1}\mathbf{x}_k$, should have none. For each run the cell prints the factors by which one step multiplies the two modes, $R(-10\eta)$ and $R(-\eta)$, and the length of the iterate after the 100 steps. The factors are measured, by applying one step to the identity matrix.

```
H = np.diag([10.0, 1.0])                        # Hessian of L(x) = x^T H x / 2
flow = lambda x: -H @ x                         # the gradient flow
solvers = {                                     # n steps of size eta on the flow
    'forward Euler': lambda x0, eta, n: euler(flow, x0, n * eta, n),
    'RK4': lambda x0, eta, n: rk4(flow, x0, n * eta, n),
    'backward Euler': lambda x0, eta, n: backward_euler_linear(-H, x0, n * eta, n)}
print(f'{"solver":<15}{"eta":>6}{"steep factor":>14}{"flat factor":>13}{"||x_100||":>12}')
for name, eta in [('forward Euler', 0.19), ('forward Euler', 0.21), ('RK4', 0.27),
                  ('RK4', 0.29), ('backward Euler', 0.25), ('backward Euler', 25)]:
    steep, flat = np.diag(solvers[name](np.eye(2), eta, 1))  # one step's matrix
    x_100 = solvers[name](np.array([1.0, 1.0]), eta, 100)
    print(f'{name:<15}{eta:6.2f}{steep:14.3f}{flat:13.3f}{np.linalg.norm(x_100):12.1e}')
```

solver	eta	steep factor	flat factor	||x_100||
forward Euler	0.19	-0.900	0.810	2.7e-05
forward Euler	0.21	-1.100	0.790	1.4e+04
RK4	0.27	0.879	0.763	2.5e-06
RK4	0.29	1.187	0.748	2.8e+07
backward Euler	0.25	0.286	0.800	2.0e-10
backward Euler	25.00	0.004	0.038	3.2e-142

Each solver has its own ceiling, where its stability region ends. Forward Euler converges at $\eta = 0.19$, with a steep factor of -0.9, and diverges at $\eta = 0.21$, with -1.1. RK4 still converges at $\eta = 0.27$ and diverges at 0.29. Backward Euler converges at $\eta = 0.25$, above the ceiling of gradient descent, and at $\eta = 25$, far above both, where its factors are 0.004 and 0.038 and a hundred steps leave nothing measurable of the iterate. The opening asked what the problem is, if it is not the loss. The answer is that the ceiling is a quotient. The loss supplies its denominator, $\lambda_{\max}$, and the solver supplies its numerator, which is 2 for

forward Euler and 2.79 for RK4. Backward Euler has no ceiling at all.

Should we then train with RK4? No. Its ceiling is 1.4 times higher and its step takes four gradients, so each gradient advances the flow by at most 0.07, against 0.2 for gradient descent. The extra gradients serve to follow the path of the gradient flow more closely, and an optimizer has no use for the path. It wants the place where the path ends. Nor is leaving the path only an error. To second order in η the iterates of gradient descent follow the flow of a modified loss that penalizes large gradients, so the error that Euler makes as a solver acts as a regularizer (Exercise 5).

The two methods also fail differently. Euler's steep factor leaves the stable range through -1, and the iterate jumps across the valley floor to land higher each time. RK4's factor is positive on the whole real axis and leaves through $+1$: at $\eta = 0.29$ it is 1.187, and the iterate walks away from the minimum without once changing sign. The zigzag, too, belongs to Euler's method. Momentum, the usual remedy for it, solves a different equation: Polyak's heavy ball discretizes $m\ddot{\mathbf{x}} + c\dot{\mathbf{x}} = -\nabla\mathcal{L}(\mathbf{x})$, the motion of a ball with mass and friction (Polyak 1964) (Exercise 6).

Backward Euler only improves as the step grows, since its factors $1/(1+\eta\lambda_i)$ shrink. The catch is the solve. Its step $\mathbf{x}_{k+1} = \mathbf{x}_k - \eta\nabla\mathcal{L}(\mathbf{x}_{k+1})$ says that the gradient of $\mathcal{L}(\mathbf{x}) + \frac{1}{2\eta}\|\mathbf{x} - \mathbf{x}_k\|^2$ vanishes at $\mathbf{x}_{k+1}$, which for a convex loss makes it the proximal step $\text{prox}_{\eta\mathcal{L}}(\mathbf{x}_k)$ of Section 35.6.4.1, applied to the loss itself. On our quadratic that is one linear system. On a network it is an optimization problem of its own at every step, and for large η it is the problem we set out to solve.

The valley is the stiff system of Section 38.2.3 in a milder form, with rates 10 and 1 in place of 50 and 1, and η in place of h. At any $\eta < 0.2$ the flat factor $1-\eta$ of gradient descent stays above 0.8 (the table prints 0.81 at $\eta = 0.19$), so the flat direction loses less than a fifth of its length per step, however well the steep one does. In general it loses less than the fraction $2\lambda_{\text{min}}/\lambda_{\text{max}}$, which is a five-hundredth for a condition number of 1000. Stability fixes η through λ_{max}, and λ_{min} then sets the speed, so the condition number of the Hessian is the stiffness of the gradient flow. On a quadratic, a diverging learning rate is a solver instability.

We knew the eigenvalues of every field in this section before we took the first step, because we had written the field down ourselves. A field that is a network comes with no such list. Section 38.3 takes a learned field as the model itself, with layers for Euler steps, and asks how such a model is trained and what it does to a probability density.

38.2.5 Summary

A numerical solver replaces a trajectory by steps, and two properties judge it. Its order p says how fast the error falls with the step: a local error $O(h^{p+1})$ accumulates to a global error $O(h^p)$, with $p = 1$ for Euler and $p = 4$ for RK4. Its stability region says which steps do not blow up: forward Euler needs $|1 + h\lambda| < 1$, backward Euler is stable on the whole left half-plane and solves an equation at every step, and a system is stiff when a fast mode that has long decayed still sets the step. On a linear field both properties are read from the stability function $R(h\lambda)$, the factor by which one step multiplies a mode. Gradient descent is forward Euler on $\dot{\mathbf{x}} = -\nabla\mathcal{L}$, so its ceiling $\eta < 2/\lambda_{\text{max}}(\mathbf{H})$ is the edge of Euler's region, and ill-conditioning is stiffness.

38.2.6 Exercises

1. code **Adaptive step size.** ● An embedded pair estimates the local error from two methods of different order. The simplest pair uses the stages $\mathbf{k}_1 = \mathbf{f}(\mathbf{x})$ and $\mathbf{k}_2 = \mathbf{f}(\mathbf{x}+h\mathbf{k}_1)$ of Heun's method: the Euler value $\mathbf{x}+h\mathbf{k}_1$ and the Heun value $\mathbf{x}+\frac{h}{2}(\mathbf{k}_1+\mathbf{k}_2)$ differ by $\Delta = \frac{h}{2}\|\mathbf{k}_2-\mathbf{k}_1\|$. A controller accepts the step, keeping the Heun value, when $\Delta \le$ `tol`, and in either case continues with the step size

$$h \leftarrow h\,\min\!\Big(2,\ \max\!\big(0.2,\ 0.9\sqrt{\texttt{tol}/\Delta}\,\big)\Big).$$

   1. Which order of the local error does the exponent $1/2$ assume?
   2. Implement the controller, starting from $h = 0.1$ and shortening the last step so that it ends at T. Integrate the spiral field `f` from `x0` to $T = 2$ for `tol` $= 10^{-2}, 10^{-3}, \dots, 10^{-6}$. Count the field evaluations, including those of rejected steps, and compare the error with the exact endpoint `x_true`. How does the number of evaluations scale with `tol`?
   3. Integrate the field of `A_stiff` from $(1,1)$ to $T = 10$ with `tol` $= 10^{-3}$ and plot the accepted step sizes against time. At which value do they settle after the fast mode has decayed? Explain this value with Heun's stability function $1 + z + z^2/2$ from Equation 38.12.

2. **Rotating modes.** ● The stiff example has real eigenvalues. A complex eigenvalue λ enters through $z = h\lambda$, the coordinate of Figure 38.6, and its imaginary part can decide whether a step is stable.

   1. For the spiral field of Figure 38.1, with eigenvalues $-0.5 \pm \mathrm{i}$, the text found that forward Euler decays for $h < 0.8$. Find the largest such step for Heun's method from its stability function in Equation 38.12 (the condition is a cubic in h; solve it numerically), and for RK4. Heun's method reads the field twice per step and RK4 four times. Which of the three methods covers the most time per field evaluation without growing?
   2. For the rotational field $\dot{\mathbf{x}} = \left(\begin{smallmatrix} 0 & -b \\ b & 0 \end{smallmatrix}\right)\mathbf{x}$, show that forward Euler multiplies $\|\mathbf{x}\|$ by $\sqrt{1+h^2b^2}$ per step and that backward Euler divides it by the same factor. By which factor does each method change $\|\mathbf{x}\|$ over one revolution, that is $2\pi/(hb)$ steps, when $hb = 0.1$?
   3. The stability function of RK4 is the last formula in Equation 38.12, and its square modulus on the imaginary axis decides rotation. Show that $|R(\mathrm{i}y)|^2 = 1 - y^6/72 + y^8/576$ for real y, and find the range of hb for which RK4 does not amplify the rotation.
   4. An explicit Runge–Kutta method with s stages computes $\mathbf{k}_i = \mathbf{f}(\mathbf{x}_n + h\sum_{j<i} a_{ij}\mathbf{k}_j)$ for $i = 1, \dots, s$ and $\mathbf{x}_{n+1} = \mathbf{x}_n + h\sum_i b_i\mathbf{k}_i$, as RK4 does in Equation 38.11. Show that its stability function is a polynomial in z of degree at most s. Why does this rule out A-stability for explicit methods?

3. code **Implicit steps on a stiff nonlinear field.** ● The text's stiff system is linear and diagonal, so each backward-Euler step was one linear solve. Take instead the damped pendulum $\dot\phi = \omega$, $\dot\omega = -\sin\phi - c\,\omega$ of Section 38.1.2.5 with the heavy damping $c = 50$, from $(\phi,\omega) = (1,0)$ to $T = 2$.

   1. Compute the eigenvalues of its Jacobian $\mathbf{J}(\phi,\omega) = \left(\begin{smallmatrix} 0 & 1 \\ -\cos\phi & -c \end{smallmatrix}\right)$ at the bottom, $(0,0)$, and the step-size limits of forward Euler and RK4 that they imply.

2. Show that after the fast transient the motion nearly satisfies $\dot{\phi} = -\sin\phi/c$, and compute $\phi(2)$ from this one-dimensional equation.
3. One backward-Euler step solves $\mathbf{g}(\mathbf{y}) = \mathbf{y} - \mathbf{x}_n - h\mathbf{f}(\mathbf{y}) = \mathbf{0}$ for $\mathbf{y} = \mathbf{x}_{n+1}$. Write Newton's iteration $\mathbf{y} \leftarrow \mathbf{y} - (\mathbf{I} - h\mathbf{J}(\mathbf{y}))^{-1}\mathbf{g}(\mathbf{y})$ for it, and implement backward Euler with three Newton iterations per step, started from $\mathbf{x}_n$.
4. Run forward Euler, RK4 and backward Euler with $n = 20$, 33, 36 and 100 equal steps ($h = 0.1$, 0.061, 0.056 and 0.02), against a reference computed with `rk4` and 2^{16} steps. Which runs blow up, and do the limits of part a predict them? Then find for each method the smallest n with an endpoint error below 10^{-4}, and count the field and Jacobian evaluations it takes. For which method does accuracy set n, and for which does stability?

4. code **Symplectic Euler and splitting.** ● The harmonic oscillator $\dot{q} = p$, $\dot{p} = -q$ conserves $q^2 + p^2$. Forward Euler spirals out of its circular orbits and backward Euler spirals in. The *symplectic Euler* update advances q with the old p and then p with the new q: $q_{n+1} = q_n + hp_n$, $p_{n+1} = p_n - hq_{n+1}$.
   1. Write the update as a matrix $\mathbf{S}_h$ acting on (q, p) and show that $\det \mathbf{S}_h = 1$ for every h, so the update preserves area as the exact flow does, while forward Euler's matrix has determinant $1 + h^2$.
   2. Show that the eigenvalues of $\mathbf{S}_h$ are distinct and lie on the unit circle if and only if $0 < h < 2$. What happens at $h = 2$ and for $h > 2$?
   3. Show that $q^2 + p^2 + hqp$ is conserved exactly by the update. What does its level set through $(1, 0)$ look like, and how far from the origin can the computed orbit get at $h = 0.5$ and at $h = 1$? (Hint: the axes of the quadratic form have lengths in the ratio $\sqrt{(2+h)/(2-h)}$.)
   4. Integrate from $(1, 0)$ to $T = 200$ with forward Euler, backward Euler and symplectic Euler at $h = 0.1$ and $h = 1$, and report the largest and smallest $\|(q, p)\|$ along each run.
   5. The update is the exact flow of $\dot{q} = p$ with p frozen, for time h, followed by the exact flow of $\dot{p} = -q$ with q frozen. Write the two generators $\mathbf{P}$ and $\mathbf{Q}$, whose sum is the oscillator's matrix, compute $e^{h\mathbf{P}}$, $e^{h\mathbf{Q}}$ and $e^{h(\mathbf{P}+\mathbf{Q})}$, and show that $e^{h\mathbf{Q}}e^{h\mathbf{P}} = \mathbf{S}_h$ differs from $e^{h(\mathbf{P}+\mathbf{Q})}$ by terms of order h^2. The spiral's closed form in Section 38.1 split an exponential without any error. Which property that it used fails here? The symmetric splitting $e^{h\mathbf{Q}/2}e^{h\mathbf{P}}e^{h\mathbf{Q}/2}$ is the *leapfrog* integrator of Hamiltonian Monte Carlo (Neal 2011); what is its order?

5. code **Implicit gradient regularization.** ●● Gradient descent is forward Euler on the gradient flow (Section 38.2.4), so each step leaves the flow of $\mathcal{L}$ by $O(\eta^2)$. Backward error analysis looks for a modified field $\tilde{\mathbf{f}} = \mathbf{f} + \eta\mathbf{f}_1$ whose exact time-η flow agrees with one Euler step up to $O(\eta^3)$. Barrett and Dherin (2021) find that for gradient descent this field is again a gradient field, of the modified loss

$$\tilde{\mathcal{L}}(\mathbf{x}) = \mathcal{L}(\mathbf{x}) + \frac{\eta}{4}\|\nabla\mathcal{L}(\mathbf{x})\|^2.$$

   1. Expand the exact flow of $\tilde{\mathbf{f}}$ over time η to second order, and show that $\mathbf{f}_1 = -\frac{1}{2}(\partial\mathbf{f}/\partial\mathbf{x})\,\mathbf{f}$. Derive $\tilde{\mathcal{L}}$ for $\mathbf{f} = -\nabla\mathcal{L}$. Which regions of parameter space does the added term penalize?
   2. For $\mathcal{L}(x) = \frac{\lambda}{2}x^2$ with $0 < \eta\lambda < 1$, show that the iterates of gradient descent lie

exactly on the gradient flow of $\frac{\tilde{\lambda}}{2}x^2$ with $\tilde{\lambda} = -\log(1-\eta\lambda)/\eta$. Expand $\tilde{\lambda}$ in η and compare it with the curvature of $\tilde{\mathcal{L}}$.

3. Gradient descent on this quadratic still converges for $1 < \eta\lambda < 2$, with iterates that alternate in sign. Trajectories of a scalar autonomous field with a Lipschitz constant cannot cross, so the time-η map of such a field preserves order. Conclude that no such field takes the values of the iterates at the times $k\eta$. For a quadratic with Hessian $\mathbf{H}$ in d dimensions, show that for every starting point with a component along the top eigenvector, the iterates lie on the flow of no gradient field $-\nabla(\frac{1}{2}\mathbf{x}^\top\tilde{\mathbf{H}}\mathbf{x})$ once $\eta \geq 1/\lambda_{\max}(\mathbf{H})$.
4. For $\mathcal{L}(x) = x^4/4 + x^2/2$ and $x_0 = 1.5$, run gradient descent to $T = 1$ with $\eta = 0.1, 0.05, 0.025, 0.0125$. Compute the gradient flows of $\mathcal{L}$ and of $\tilde{\mathcal{L}}$ to $T = 1$ with `rk4` and many steps, the latter separately for each η. Fit the slopes of the two endpoint errors of gradient descent against η.

6. **Heavy ball as a discretized ODE.** ● Heavy-ball momentum $\mathbf{x}_{k+1} = \mathbf{x}_k - \eta\nabla\mathcal{L}(\mathbf{x}_k) + \beta(\mathbf{x}_k - \mathbf{x}_{k-1})$ discretizes the equation $m\ddot{\mathbf{x}} + c\dot{\mathbf{x}} = -\nabla\mathcal{L}(\mathbf{x})$ of a ball with mass m and friction c.
   1. Replace $\ddot{\mathbf{x}}(t_k)$ by the central difference $(\mathbf{x}_{k+1} - 2\mathbf{x}_k + \mathbf{x}_{k-1})/h^2$ and $\dot{\mathbf{x}}(t_k)$ by the backward difference $(\mathbf{x}_k - \mathbf{x}_{k-1})/h$. Show that the result is the heavy-ball update, and express η and β through m, c, and h.
   2. For $\mathcal{L}(x) = \frac{\lambda}{2}x^2$ the ODE is a damped oscillator, critically damped at $c = 2\sqrt{m\lambda}$. Express this condition through η, β, and λ. The heavy-ball recurrence is critically damped when its characteristic polynomial $z^2 - (1+\beta-\eta\lambda)z + \beta$ has a double root. Find the corresponding β and compare the two conditions for small $\eta\lambda$.
   3. The heavy-ball recurrence on this quadratic converges if and only if $0 < \eta\lambda < 2(1+\beta)$ for $0 \leq \beta < 1$ (Exercise 8a of Section 35.1 asks for the proof). Translate the condition into one on h, m, c, and λ. The ODE decays for every $\lambda > 0$, but its discretization does not. What is the largest stable step h as $c \to 0$, in terms of the angular frequency $\sqrt{\lambda/m}$ of the oscillator? Which eigenvalue of the Hessian of an ill-conditioned quadratic sets the step limit?

38.3 Neural ODEs, Adjoints, and Continuous Flows

A residual block adds a correction to its input, and so does a step of Euler's method (Section 38.2). If a residual network (Section 7.4) is a numerical solution of a differential equation, what do we gain by taking the equation itself as the model, and what does it cost?

Generative modelling gives one answer. Maximum likelihood needs the density of what a model produces. A normalizing flow is a network that can be inverted, so that this density follows from the change-of-variables formula. The formula asks for the determinant of the network's Jacobian at every sample, a matrix with 3072 rows for a colour image of 32×32 pixels. In continuous time the determinant turns into a trace, and the trace can be estimated with one backward pass.

A *Neural ODE* (Chen et al. 2018) takes the equation itself as the model: a differential equation $\dot{\mathbf{x}} = \mathbf{f}_\theta(\mathbf{x}, t)$ whose field is a network and whose solution at a final time is the output. Training it and computing its density each come down to integrating one more equation beside the state. For training it is the *adjoint* equation, which optimal control developed in the 1950s to ask how the cost of a trajectory changes with its controls (Pontryagin et al. 1962).

For the density it is an integral of the Jacobian's trace.

Compared with a residual stack, the model gains an invertible map, a step count that is no longer part of the model, a gradient that needs no stored activations, and a computable density. Each gain has a catch, and this section measures all four: a flow on the line cannot swap two points, a trained field is tied to its solver, the gradient computed without stored states drifts, and the cheap trace is noisy.

We train such a model through a solver, derive the adjoint, and follow a density along the flow. The results to carry away are the adjoint equations and the trace rule Equation 38.21, on which the likelihoods of diffusion models rest; a reader who wants only the second can begin at Section 38.3.3. Nothing later builds on the subsection on memory and drift.

The cells below train through the Euler and RK4 steps of Section 38.2: the loop `solve` takes N of them, of size T/N, and a negative T runs the field backward. The check of the adjoint, further down, uses the spiral $\mathbf{A}$, its start $\mathbf{x}_0$ and the matrix exponential $e^{\mathbf{C}s} = \mathbf{V}e^{\Lambda s}\mathbf{V}^{-1}$ of Section 38.1.

```
rng = np.random.default_rng()
A = np.array([[-0.5, -1.0], [1.0, -0.5]])      # spiral sink: eigenvalues -0.5 +- i
x0 = np.array([2.0, 0.0])

def expm_eig(C, s):
    """e^{Cs} via the eigendecomposition C = V diag(mu) V^{-1}."""
    mu, V = np.linalg.eig(C)
    return (V @ np.diag(np.exp(mu * s)) @ np.linalg.inv(V)).real

def euler_step(f, x, h):
    """One forward-Euler step of size h for dx/dt = f(x)."""
    return x + h * f(x)

def rk4_step(f, x, h):
    """One classical Runge-Kutta step of size h for dx/dt = f(x)."""
    k1 = f(x)
    k2 = f(x + h / 2 * k1)
    k3 = f(x + h / 2 * k2)
    k4 = f(x + h * k3)
    return x + h / 6 * (k1 + 2 * k2 + 2 * k3 + k4)

def solve(step, f, x, T, N):
    """N steps of size T/N from time 0 to time T; T < 0 runs the field backward."""
    for _ in range(N):
        x = step(f, x, T / N)
    return x
```

38.3.1 Neural ODEs

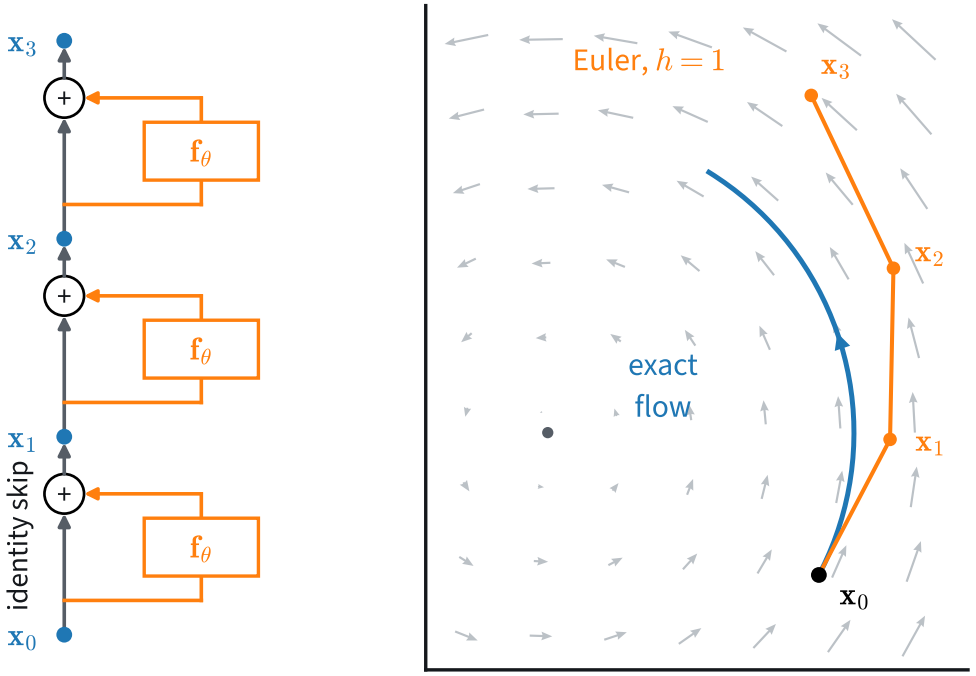
A Neural ODE is what a residual network becomes when its depth turns into time.

38.3.1.1 From Residual Blocks to a Continuous Model

Set a residual block (He et al. 2016a) beside a forward-Euler step of the ODE $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, t)$:

$$\mathbf{x}_{l+1} = \mathbf{x}_l + \mathbf{f}_{\theta_l}(\mathbf{x}_l) \qquad \text{and} \qquad \mathbf{x}_{n+1} = \mathbf{x}_n + h\,\mathbf{f}(\mathbf{x}_n, t_n). \tag{38.14}$$

With $h = 1$ and $\mathbf{f}(\cdot, t_l) = \mathbf{f}_{\theta_l}$ the two updates are the same formula. A ResNet whose blocks have different weights is therefore a time-dependent field sampled at the layer times and integrated by Euler's method with unit steps (Figure 38.7 draws the case in which all blocks share one field). Its horizon is its depth: ten blocks integrate the field to time 10.



$$\mathbf{x}_{l+1} = \mathbf{x}_l + \mathbf{f}_\theta(\mathbf{x}_l) \quad \Leftrightarrow \quad \mathbf{x}_{n+1} = \mathbf{x}_n + h\,\mathbf{f}(\mathbf{x}_n) \quad \text{with } h = 1$$

Figure 38.7: Left: a stack of residual blocks that share one field, each adding the learned branch to an identity skip. Right: the same updates read as forward-Euler steps of size $h = 1$, beside the exact flow of the same field, from which they drift.

To take the limit, fix the horizon T, keep one parameter vector θ for all times, and let the steps shrink. In the language of layers, the network grows deeper while each layer corrects less. (The field may still depend on t: the network takes t as an input, and that is how one parameter vector serves blocks that differ.) With N steps of size $h = T/N$ at the times $t_n = nh$,

$$\mathbf{x}_{n+1} = \mathbf{x}_n + h\,\mathbf{f}_\theta(\mathbf{x}_n, t_n), \qquad \mathbf{x}_N \xrightarrow{N\to\infty} \mathbf{x}(T) = \mathbf{x}_0 + \int_0^T \mathbf{f}_\theta(\mathbf{x}(t), t)\,dt. \tag{38.15}$$

The convergence is Euler's error bound from Section 38.2: for a smooth field that is Lipschitz in $\mathbf{x}$, the Euler iterate after N steps is within $O(h)$ of the exact solution. The **Neural ODE** is

the limit object: the model whose output is the solution $\mathbf{x}(T)$ of $\dot{\mathbf{x}} = \mathbf{f}_\theta(\mathbf{x}, t)$ with $\mathbf{x}(0) = \mathbf{x}_0$. The parameters define the field. The solver (Euler, RK4, an adaptive method) is a choice made afterwards, which changes the computed map, its cost and its error, but not the model.

The exact map and the computed map can differ in kind, not only in accuracy. The field $\dot{x} = -x$ has the exact flow $\Phi_t(x) = e^{-t}x$, a bijection of $\mathbb{R}$ for every t. One Euler step with $h = 1$ computes $x + 1 \cdot (-x) = 0$ and maps every input to zero. The exact map, by contrast, is a bijection whenever the field is continuous in t and globally Lipschitz in $\mathbf{x}$: running the field backward from $\mathbf{x}(T)$ is again a problem with exactly one solution, and that solution leads back to the start (Section 38.1.1.3).

38.3.1.2 Order Preservation on the Line

Invertibility is a gain, but it is also a constraint. Is a Neural ODE at least as expressive as the residual stack it came from? On the real line it is not, and a picture shows why. In the (t, x) plane, trajectories of a Lipschitz field never cross. Two trajectories that swapped -1 and 1 would have to meet on the way (panel (a) of Figure 38.8).

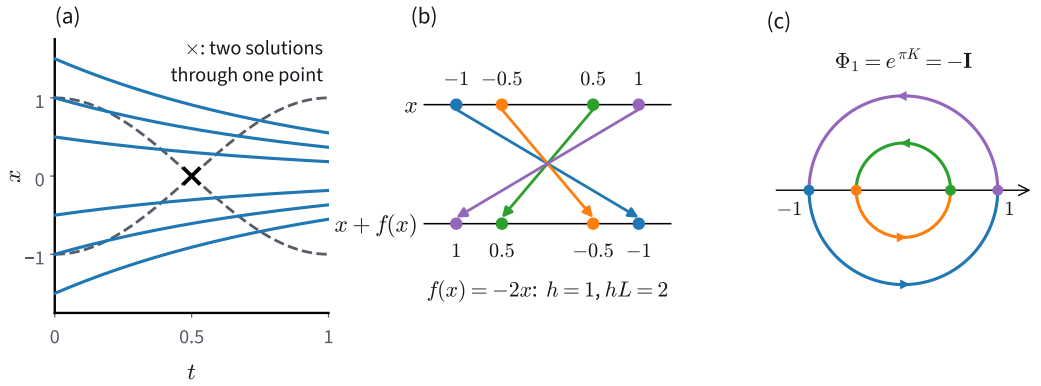


Figure 38.8: (a) On the line, trajectories of a Lipschitz field never cross (here $\dot{x} = -x$), so the time-T map preserves order; the dashed pair that swaps -1 and 1 would need two solutions through one point. (b) The residual step $x + f(x)$ with $f(x) = -2x$ swaps the points in one step: it is an Euler step with $hL = 2$, not a flow. (c) One extra dimension removes the obstruction: the rotation by π carries $(x, 0)$ to $(-x, 0)$ along half circles that pass without meeting (colours as in (b)).

Proposition (flows on $\mathbb{R}$ preserve order). *Let $f(x, t)$ be continuous in t and globally Lipschitz in x, and write $\Phi_t(x)$ for the solution at time t started from x at time 0. Then for every $T > 0$ the map Φ_T is strictly increasing.*

Proof. Let $x < y$ and suppose $\Phi_T(x) \geq \Phi_T(y)$. The function $g(t) = \Phi_t(y) - \Phi_t(x)$ is continuous with $g(0) > 0$ and $g(T) \leq 0$, so by the intermediate value theorem $g(s) = 0$ for some $s \in (0, T]$: the two trajectories meet at time s at a point z. Now run time backward from s. Both reversed trajectories $\tau \mapsto \Phi_{s-\tau}(x)$ and $\tau \mapsto \Phi_{s-\tau}(y)$ solve the same initial-value problem $\dot{w} = -f(w, s - \tau)$, $w(0) = z$, whose field is again Lipschitz. By Picard–Lindelöf that problem has only one solution, so the two reversed trajectories coincide, and at $\tau = s$ this says $x = y$, a contradiction. ■

The proof has one idea, to apply uniqueness backward from the meeting time. As a conse-

quence, no Neural ODE on $\mathbb{R}$ realizes $x \mapsto -x$: that map sends -1 to 1 and 1 to -1, which reverses their order.

The residual block $x + f(x)$ with $f(x) = -2x$ does realize it, in one step (panel (b) of Figure 38.8). That block is the Euler step with $h = 1$ of $\dot{x} = -2x$, whose exact flow $e^{-2t}x$ never reverses order. The two statements agree because one Euler step $x + hf(x)$ of a field with Lipschitz constant L is guaranteed to be increasing when $hL < 1$: for $x > y$,

$$(x + hf(x)) - (y + hf(y)) \;\geq\; (x - y) - hL\,(x - y) \;=\; (1 - hL)(x - y),$$

which is positive when $hL < 1$. Here $hL = 2$, so the step may reverse order. A finite step is not a flow, and the proposition does not constrain it.

One more dimension removes the obstruction. The rotation field $\dot{\mathbf{x}} = \mathbf{K}\mathbf{x}$ of Section 38.1, with $\mathbf{K} = \left(\begin{smallmatrix} 0 & -1 \\ 1 & 0 \end{smallmatrix}\right)$, has the flow $e^{s\mathbf{K}} = \cos s\,\mathbf{I} + \sin s\,\mathbf{K}$, which turns the plane by the angle s. Run π times as fast, as $\dot{\mathbf{x}} = \pi\mathbf{K}\mathbf{x}$, it has the time-one map

$$\Phi_1 = e^{\pi\mathbf{K}} = \cos\pi\,\mathbf{I} + \sin\pi\,\mathbf{K} = -\mathbf{I}.$$

It carries $(x, 0)$ to $(-x, 0)$ along half circles, and trajectories pass each other in the plane without meeting (panel (c) of Figure 38.8). Adding dimensions in this way is called *augmentation* (Dupont et al. 2019): the input x is embedded as $(x, 0)$, the flow runs in the plane, and the first coordinate is read out. A flow is a continuous rearrangement of space, and on a line there is no room for two points to pass.

38.3.1.3 Training Through the Solver

The forward pass of a Neural ODE is a loop of N solver steps. The direct way to train it is to *discretize, then differentiate*: write the solver into the computation graph, compute the loss on $\mathbf{x}_N$, and backpropagate through the unrolled steps. With Euler as the solver the program is a ResNet of N blocks that share one set of weights, so ordinary automatic differentiation trains it.

Let us try this on a task that is deliberately easy: carry 64 points of the unit circle to the matching points of the ellipse centred at $(1.5, 0)$ with semi-axes 1.2 and 0.4, a stretch and a shift. The exact flow of the affine field $\dot{\mathbf{x}} = \operatorname{diag}(\ln 1.2, \ln 0.4)\,\mathbf{x} + \mathbf{c}$ does this without error for a suitable $\mathbf{c}$, so whatever error we find will not be for want of a solution. Will the trained field still reach the targets when it is integrated accurately, or only with the ten steps it was trained with? In the first case training has found a flow, in the second only a ten-step program.

The field has one hidden layer of 32 tanh units, the loss is the mean squared error at the endpoints, and Adam (Section 35.5) runs full-batch iterations 0 to 400 at learning rate 0.05, lowered to 0.005 for the last hundred. We train twice from the same initial weights: once through $N = 10$ Euler steps of size $h = 0.1$, and once through ten steps of RK4, which evaluates the field four times per step.

```
angle = np.linspace(0, 2 * np.pi, 64, endpoint=False)
X0_ring = np.stack([np.cos(angle), np.sin(angle)], axis=1)           # circle
X1_ring = np.stack([1.5 + 1.2 * np.cos(angle), 0.4 * np.sin(angle)], axis=1)
solvers = {'Euler': euler_step, 'RK4': rk4_step}
```

```
seed = int(rng.integers(2**31))               # one initialization for both solvers
```

With the source circle `X0_ring`, target ellipse `X1_ring`, and shared initialization `seed` fixed, `train` fits the two-layer `tanh` vector field `net` by backpropagating through 10 steps of `solve(step, net, X, 1.0, 10)` for each solver in `solvers`:

```
X, Y = torch.tensor(X0_ring).float(), torch.tensor(X1_ring).float()

def train(name, step, seed):
    """Fit the field through 10 steps of one solver, from the given seed."""
    torch.manual_seed(seed)
    net = nn.Sequential(nn.Linear(2, 32), nn.Tanh(), nn.Linear(32, 2))
    for layer in net[::2]:                    # Glorot-uniform weights, zero biases
        nn.init.xavier_uniform_(layer.weight)
        nn.init.zeros_(layer.bias)
    opt = torch.optim.Adam(net.parameters(), lr=0.05)
    sched = torch.optim.lr_scheduler.MultiStepLR(opt, [300], 0.1)  # 0.005 from 300
    losses = []
    for it in range(401):
        loss = ((solve(step, net, X, 1.0, 10) - Y) ** 2).mean()
        opt.zero_grad()
        loss.backward()
        opt.step()
        sched.step()
        if it % 200 == 0:
            losses.append(loss.item())
    print(f'{name}-trained: loss', ', '.join(f'{l:.1e}' for l in losses),
          'at iterations 0, 200, 400')
    return net

trained = {name: train(name, step, seed) for name, step in solvers.items()}
```

```
Euler-trained: loss 1.5e+00, 3.1e-06, 5.2e-07 at iterations 0, 200, 400
RK4-trained: loss 1.5e+00, 4.3e-06, 5.8e-07 at iterations 0, 200, 400
```

Both losses fall from order one to below 10^{-5}. Judged by the loss they were trained on, the two fields are equally good.

Two tests follow. The first puts our question directly: integrate each trained field with its own solver at $N = 100$ and $N = 1000$ steps ($Nh = 1$ throughout). The larger N, the closer the computed map is to the exact time-one flow of the field. Before running the cell, guess what this does to the two endpoint errors. The second test is about invertibility, the first gain on our list. Take the Euler-trained field forward N steps and then backward N steps, and measure how far the round trip lands from where it started. An exact flow would return every point to its start.

```
dist = lambda P, Q: torch.linalg.vector_norm(P - Q, dim=1).mean().item()
print('   N   Euler-trained   RK4-trained   Euler round trip')
with torch.inference_mode():
    for N in (10, 100, 1000):
        err = [dist(solve(solvers[k], trained[k], X, 1.0, N), Y) for k in solvers]
```

```
        fwd = solve(euler_step, trained['Euler'], X, 1.0, N)
        back = solve(euler_step, trained['Euler'], fwd, -1.0, N)
        print(f'{N:4d}   {err[0]:13.1e}   {err[1]:11.1e}   {dist(back, X):16.1e}')
```

```
   N   Euler-trained   RK4-trained   Euler round trip
  10         9.0e-04       9.8e-04            5.4e-02
 100         1.7e-02       9.8e-04            5.5e-03
1000         1.9e-02       9.8e-04            5.5e-04
```

Take the round trip first, in the last column. It misses its start by about 0.055 at $N = 10$, 0.0055 at $N = 100$ and 0.00055 at $N = 1000$, tenfold less per tenfold more steps. Each pair of a forward and a backward Euler step leaves a residual of order h^2, and N of them add up to $O(h)$, Euler's order. Running the field backward undoes the exact flow. It undoes the Euler program only in the limit $h \to 0$. An Euler program with $hL < 1$ does have an inverse, since each of its steps is then a bijection (Exercise 4). But that inverse is made of implicit steps, not of Euler steps with $-h$.

Now the endpoint errors. At the training step count both fields land within a few thousandths of their targets. Surprisingly, finer steps make the Euler-trained field worse: its error *rises* to about 0.02 and then stops changing. Training fitted the ten-step program. The exact time-one flow of that field, which $N = 1000$ approximates, lands 0.02 from the targets, under 2% of the ellipse's long semi-axis: the field defines a flow close to its ten-step program, not equal to it.

The RK4-trained field prints the same error at ten, a hundred and a thousand steps. Ten RK4 steps already compute the flow of so smooth a field to the digits shown, so fitting the program fitted the flow, and the step count is free after training. A field trained through a crude solver is a program; a field trained through an accurate one is a flow.

The accuracy has to be supplied in every iteration, however. Training through RK4 evaluates the field forty times per iteration and backpropagates through all forty evaluations; training through Euler, ten times. A hundred Euler steps or a second-order solver would change the count and nothing else. The alternative is to take the solver out of training altogether: fit the field by regression onto targets that are known in closed form, at one field evaluation per example, so that what is fitted is a flow from the start. This is how Section 38.6 and Section 38.7 learn their fields.

38.3.2 The Adjoint Method

Training gave us the gradient of the ten-step program, and the program and the flow turned out to be different maps. Are their gradients different too? And can the gradient of the flow be computed at all, when no finite program computes the flow?

It can, by an idea that reverse-mode differentiation already contains. Reverse mode carries the derivative of the loss with respect to a layer's activations backward from layer to layer; Section 34.3.2 called it the adjoint of a node. The adjoint method carries the same quantity backward through time, with one node $\mathbf{x}(t)$ for every instant. Its central object is the *adjoint state* $\mathbf{a}(t) = \partial \mathcal{L} / \partial \mathbf{x}(t)$, the gradient of the loss with respect to the state at time t, and it turns out to obey a linear differential equation of its own.

38.3.2.1 The Scalar Case

Before the general statement we work one example to the end, an example in which every quantity has a closed form. Take $\dot{x} = \theta x$ with $x(0) = x_0$ and the loss $\mathcal{L} = \frac{1}{2}x(T)^2$ on the endpoint. Directly, $x(T) = e^{\theta T}x_0$, so $\partial x(T)/\partial\theta = T\,x(T)$ and

$$\frac{\partial \mathcal{L}}{\partial \theta} = x(T)\,\frac{\partial x(T)}{\partial \theta} = T\,x(T)^2.$$

With $\theta = T = x_0 = 1$ this is $e^2 = 7.389$.

The adjoint reaches the same number by another route. A nudge δ to the state at time t grows to $e^{\theta(T-t)}\delta$ by time T and changes the loss by $x(T)\,e^{\theta(T-t)}\delta$, so $a(t) = e^{\theta(T-t)}x(T)$. This adjoint solves the ODE $\dot{a} = -\theta a$ backward from the final value $a(T) = x(T)$, which is the derivative of the loss $\ell(x) = \frac{1}{2}x^2$ at $x(T)$. A change $\delta\theta$ of the parameter alters the velocity by $x(t)\,\delta\theta$ at every instant, since $\partial f/\partial\theta = x$, and the adjoint converts each of these nudges into a change of the loss. Summing over time,

$$\int_0^T a(t)\,\frac{\partial f}{\partial \theta}\,dt = \int_0^T e^{\theta(T-t)}x(T)\;e^{\theta t}x_0\,dt = \int_0^T x(T)^2\,dt = T\,x(T)^2. \tag{38.16}$$

The integrand is constant: x grows forward, a shrinks by the same factor when read forward, and their product stays at $x(T)^2$ (Figure 38.9). The gradient is the area under it.

38.3.2.2 The Adjoint Equations

The general case keeps the shape of the scalar one: an adjoint that runs backward from the gradient of the loss, and a parameter gradient that integrates the adjoint against $\partial\mathbf{f}/\partial\theta$.

Proposition (the adjoint equations). *Let* $\mathbf{f}$ *be continuously differentiable in* $\mathbf{x}$ *and* θ*, let* $\mathbf{x}(t)$ *solve* $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \theta, t)$ *(the field* $\mathbf{f}_\theta$*, with its parameters written as an argument) on* $[0, T]$ *from a fixed* $\mathbf{x}(0)$*, and let* $\mathcal{L} = \ell(\mathbf{x}(T))$ *for a differentiable* ℓ*. Then the adjoint state satisfies the linear ODE*

$$\dot{\mathbf{a}}(t) = -\left(\frac{\partial \mathbf{f}}{\partial \mathbf{x}}\right)^{\top}\mathbf{a}(t), \qquad \mathbf{a}(T) = \nabla\ell(\mathbf{x}(T)), \tag{38.17}$$

integrated backward from T*, and the parameter gradient is the integral*

$$\frac{\partial \mathcal{L}}{\partial \theta} = \int_0^T \left(\frac{\partial \mathbf{f}}{\partial \theta}\right)^{\top}\mathbf{a}(t)\,dt, \tag{38.18}$$

with both partial derivatives evaluated along the trajectory $(\mathbf{x}(t), \theta, t)$ (Chen et al. 2018).

Proof (variational). Perturb the parameters by an infinitesimal $\delta\theta$. Because $\mathbf{f}$ is C^1, the solution is differentiable in θ, and the first-order perturbation $\delta\mathbf{x}(t)$ of the trajectory obeys the *linearization* of the dynamics (differentiate the ODE in θ and use the chain rule):

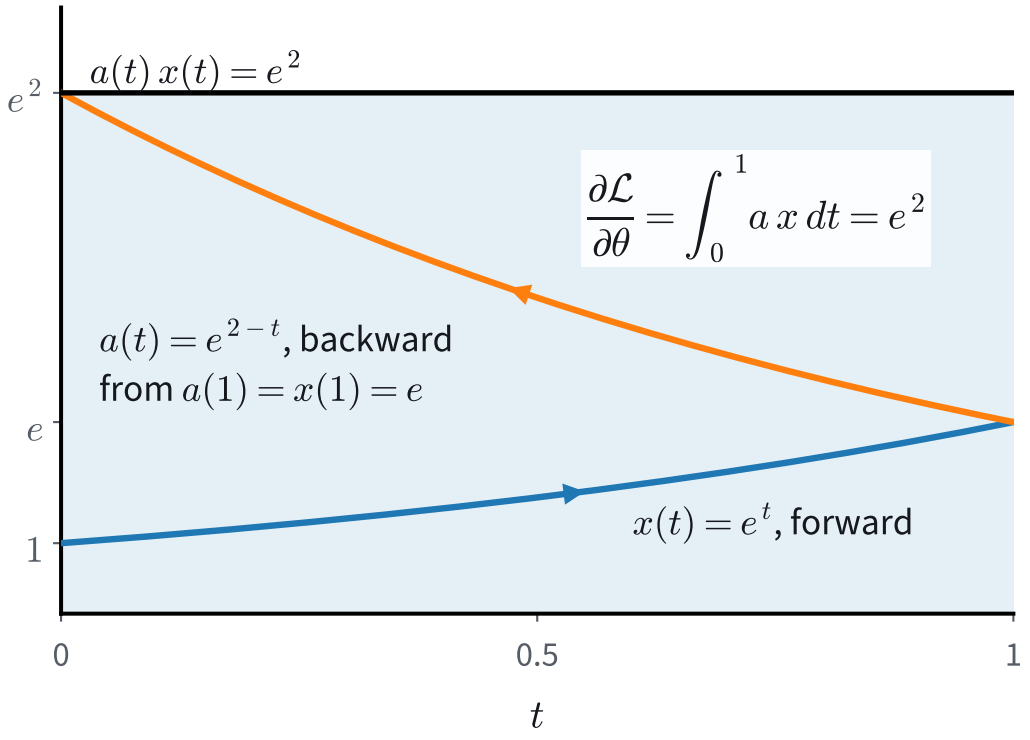


Figure 38.9: The scalar adjoint for $\dot{x} = \theta x$ at $\theta = 1$, $x_0 = 1$, $T = 1$: the state grows forward, the adjoint is integrated backward from the loss gradient $a(1) = x(1)$, and their product is constant; the parameter gradient is the shaded area, $Tx(T)^2 = e^2$.

$$\delta\dot{\mathbf{x}} = \frac{\partial \mathbf{f}}{\partial \mathbf{x}}\,\delta\mathbf{x} + \frac{\partial \mathbf{f}}{\partial \theta}\,\delta\theta, \qquad \delta\mathbf{x}(0) = \mathbf{0}.$$

Define $\mathbf{a}(t)$ by the backward ODE Equation 38.17 and differentiate the pairing $\mathbf{a}^\top \delta\mathbf{x}$:

$$\begin{aligned}\frac{d}{dt}\left(\mathbf{a}^\top \delta\mathbf{x}\right) &= \dot{\mathbf{a}}^\top \delta\mathbf{x} + \mathbf{a}^\top \delta\dot{\mathbf{x}} \\ &= -\mathbf{a}^\top \frac{\partial \mathbf{f}}{\partial \mathbf{x}}\,\delta\mathbf{x} + \mathbf{a}^\top \frac{\partial \mathbf{f}}{\partial \mathbf{x}}\,\delta\mathbf{x} + \mathbf{a}^\top \frac{\partial \mathbf{f}}{\partial \theta}\,\delta\theta = \mathbf{a}^\top \frac{\partial \mathbf{f}}{\partial \theta}\,\delta\theta.\end{aligned}$$

The state-coupling terms cancel, and the adjoint ODE was defined so that they would. Integrate from 0 to T, using $\delta\mathbf{x}(0) = \mathbf{0}$ at the left endpoint:

$$\mathbf{a}(T)^\top \delta\mathbf{x}(T) = \int_0^T \mathbf{a}(t)^\top \frac{\partial \mathbf{f}}{\partial \theta}\,dt\;\delta\theta.$$

The left side is $\nabla \ell(\mathbf{x}(T))^\top \delta\mathbf{x}(T) = \delta\mathcal{L}$, the first-order change of the loss; the right side is linear in $\delta\theta$ with coefficient Equation 38.18. It remains to identify $\mathbf{a}$ with the adjoint state. Run the same argument with $\delta\theta = \mathbf{0}$ and a nudge $\delta\mathbf{x}(s)$ injected at time s: the pairing $\mathbf{a}^\top \delta\mathbf{x}$ is then constant on $[s, T]$, so $\delta\mathcal{L} = \mathbf{a}(T)^\top \delta\mathbf{x}(T) = \mathbf{a}(s)^\top \delta\mathbf{x}(s)$, and the solution of Equation 38.17 is $\partial\mathcal{L}/\partial\mathbf{x}(s)$. ■

The backward ODE is linear in $\mathbf{a}$, with the transposed Jacobian as its matrix. Each of its steps computes $\mathbf{a}^\top \partial\mathbf{f}/\partial\mathbf{x}$, one vector–Jacobian product, which costs one reverse-mode pass through $\mathbf{f}$ (Section 34.5). The same pass returns $(\partial\mathbf{f}/\partial\theta)^\top \mathbf{a}$, the integrand of the parameter gradient, however many parameters there are. (A ReLU field is only piecewise C^1, so the proposition does not cover it, although automatic differentiation still differentiates the unrolled solver piece by piece. Smooth activations such as tanh or GELU satisfy the hypothesis.)

38.3.2.3 Backpropagation as the Discrete Adjoint

Backpropagation through the Euler program can now be set beside the adjoint ODE, line by line. To keep the formulas short, let the field be autonomous and write $\mathbf{J} = \partial\mathbf{f}/\partial\mathbf{x}$ for its Jacobian. Differentiate the program $\mathbf{x}_{n+1} = \mathbf{x}_n + h\,\mathbf{f}(\mathbf{x}_n, \theta)$ by the chain rule, with $\mathbf{a}_n = \partial\mathcal{L}/\partial\mathbf{x}_n$. The Jacobian of one step is $\mathbf{I} + h\mathbf{J}(\mathbf{x}_n)$, so

$$\begin{aligned}\mathbf{a}_n &= \big(\mathbf{I} + h\mathbf{J}(\mathbf{x}_n)\big)^\top \mathbf{a}_{n+1}, \qquad \mathbf{a}_N = \nabla \ell(\mathbf{x}_N), \\ \frac{\partial \mathcal{L}}{\partial \theta} &= \sum_{n=0}^{N-1} h \left(\frac{\partial \mathbf{f}}{\partial \theta}(\mathbf{x}_n)\right)^\top \mathbf{a}_{n+1}.\end{aligned} \tag{38.19}$$

Compare this with one explicit Euler step of the adjoint ODE Equation 38.17 from t_{n+1} back to t_n:

$$\mathbf{a}_n = \mathbf{a}_{n+1} + h\,\mathbf{J}(\mathbf{x}_{n+1})^\top \mathbf{a}_{n+1}.$$

The two updates differ only in where the Jacobian is evaluated: at the start $\mathbf{x}_n$ of the forward step or at its end $\mathbf{x}_{n+1}$. The sum in Equation 38.19 is a rectangle rule for the integral Equation 38.18. So backpropagation through a solver is a *discrete adjoint*, and a solver applied to the adjoint ODE is the *continuous adjoint*, discretized. The literature calls the two routes discretize-then-optimize, which is the "discretize, then differentiate" of our training run, and optimize-then-discretize (Kidger 2022). The first is the exact gradient of the program, the second approximates the gradient of the flow, and for Euler the two gradients differ at first order in h. Of everything in this subsection, this is the statement to keep.

The scalar example shows both gradients at once. For $\dot{x} = \theta x$ the Euler program is $x_N = (1+h\theta)^N x_0$, so $\partial x_N/\partial\theta = Nh(1+h\theta)^{N-1}x_0 = T\,x_N/(1+h\theta)$ and the discrete adjoint gives $\partial\mathcal{L}/\partial\theta = T\,x_N^2/(1+h\theta)$. With $\theta = T = x_0 = 1$ this is 6.116 at $N = 10$, 7.244 at $N = 100$ and 7.374 at $N = 1000$, against the exact $e^2 = 7.389$. The gaps 1.27, 0.145 and 0.0148 shrink tenfold per tenfold N.

The spiral field of Section 38.1, $\dot{\mathbf{x}} = \mathbf{A}\mathbf{x}$ with $\mathbf{A} = \left(\begin{smallmatrix} -0.5 & -1 \\ 1 & -0.5 \end{smallmatrix}\right)$, gives the same check in two dimensions with every ingredient in closed form. For a linear field the Jacobian is $\mathbf{A}$ at every point, so the two recursions displayed above are one and the same. What is left to measure is the error that both have against the flow, where the states, the adjoint and the sum are all Euler's.

Take θ to be the entries of $\mathbf{A}$ and $\mathcal{L} = \frac{1}{2}\|\mathbf{x}(T)\|^2$. The trajectory is $e^{\mathbf{A}t}\mathbf{x}_0$, the adjoint is $\mathbf{a}(t) = e^{\mathbf{A}^\top(T-t)}\mathbf{x}(T)$ (the solution of $\dot{\mathbf{a}} = -\mathbf{A}^\top\mathbf{a}$), and since $\partial\mathbf{f}/\partial A_{ij} = x_j\mathbf{e}_i$, the gradient Equation 38.18 becomes $\partial\mathcal{L}/\partial\mathbf{A} = \int_0^T \mathbf{a}(t)\,\mathbf{x}(t)^\top dt$.

The cell writes the backward sweep Equation 38.19 by hand, for $T = 2$. Its first line of output tests the sweep against central finite differences of the same program at $N = 1000$ steps, which tells us only that the sweep is coded correctly. The distance that matters is the one to the gradient of the exact flow. For it the cell evaluates the continuous adjoint integral by the trapezoid rule on 2000 intervals, with $\mathbf{a}$ and $\mathbf{x}$ from the matrix exponential, and compares it with the discrete adjoint at three step counts.

```
def loss_and_grad_unrolled(A, x0, T, N):
    """L = ||x_N||^2 / 2 through N unrolled Euler steps; dL/dA by hand-written
    reverse mode, the discrete adjoint."""
    h, xs = T / N, [x0]
    for _ in range(N):
        xs.append(xs[-1] + h * (A @ xs[-1]))   # forward sweep, tape the states
    a, gA = xs[-1].copy(), np.zeros_like(A)    # a_N = dL/dx_N = x_N
    for n in range(N - 1, -1, -1):             # backward sweep: one VJP per step
        gA += h * np.outer(a, xs[n])           # dL/dA += h a_{n+1} x_n^T
        a = a + h * (A.T @ a)                  # a_n = (I + hA)^T a_{n+1}
    return 0.5 * xs[-1] @ xs[-1], gA

T = 2.0
loss = lambda A: loss_and_grad_unrolled(A, x0, T, 1000)[0]
g_fd = np.array([(loss(A + 1e-5 * E) - loss(A - 1e-5 * E)) / 2e-5
                 for E in np.eye(4).reshape(4, 2, 2)]).reshape(2, 2)
gap_fd = np.abs(loss_and_grad_unrolled(A, x0, T, 1000)[1] - g_fd).max()
print(f'N =  1000: |backward sweep - central differences| = {gap_fd:.1e}')
n_quad = 2000                                  # continuous adjoint, trapezoid rule
```

```
ts = np.linspace(0.0, T, n_quad + 1)
xT = expm_eig(A, T) @ x0
AX = np.stack([np.outer(expm_eig(A.T, T - s) @ xT, expm_eig(A, s) @ x0)
               for s in ts])                    # a(s) x(s)^T
g_cont = (T / n_quad) * (AX[1:-1].sum(axis=0) + 0.5 * (AX[0] + AX[-1]))
for N in (100, 1000, 10000):
    gap = np.abs(loss_and_grad_unrolled(A, x0, T, N)[1] - g_cont).max()
    print(f'N = {N:5d}: |discrete - continuous adjoint|     = {gap:.1e}')
```

```
N =  1000: |backward sweep - central differences| = 1.2e-10
N =   100: |discrete - continuous adjoint|     = 2.9e-02
N =  1000: |discrete - continuous adjoint|     = 2.8e-03
N = 10000: |discrete - continuous adjoint|     = 2.8e-04
```

The sweep matches the finite differences to about 10^{-10}, the accuracy of the differences themselves. Its gap to the gradient of the flow falls from 2.9×10^{-2} to 2.8×10^{-3} to 2.8×10^{-4}, tenfold per tenfold N, as in the scalar case.

38.3.2.4 Memory and Drift

The adjoint ODE needs $\mathbf{x}(t)$ at every backward step, because the Jacobian is evaluated there. There are two ways to supply it. Store the forward states, as backpropagation does, which costs memory proportional to N. Or re-integrate $\dot{\mathbf{x}} = \mathbf{f}$ backward from $\mathbf{x}(T)$ alongside $\mathbf{a}$, which keeps the memory constant in N at the cost of one more field evaluation per step. The second option sounds too good, and the smallest case shows why: it amplifies errors.

For $\dot{x} = -\lambda x$ with $\lambda > 0$, the backward integration solves $\dot{x} = +\lambda x$. An absolute error δ in the stored $x(T)$ therefore becomes $e^{\lambda T}\delta$ by the time it reaches $t = 0$.

How large is δ? Picture a state with two modes. One decays at the rate $\lambda = 50$, the fast rate of the stiff example of Section 38.2.3, and the other hardly moves. Let the fast mode point along a diagonal, so that every coordinate contains some of it. By time T the fast mode has died out, the coordinates that are stored are of size one, and double precision rounds each of them at about 10^{-16}. Rounding knows nothing of modes: part of the error lies along the fast direction, so $\delta \approx 10^{-16}$. The reconstructed start is then off by 2.7×10^{-11} at $T = 0.25$, by 7×10^{-6} at $T = 0.5$ and by 5×10^{5} at $T = 1$, larger than the state itself. A double carries about sixteen digits, and a factor of $e^{50} \approx 5 \times 10^{21}$ is five digits more than it has. Exercise 3 measures this growth.

For the spiral, whose eigenvalues have real part -0.5, the factor over $T = 2$ is only e. A strongly contracting forward flow is a strongly expanding backward one, so re-integration saves memory at the cost of drift. For stiff fields *checkpointing* (Section 34.5.2.5) is the sturdier choice. It stores the state only at a few checkpoints and recomputes the states in between forward from the checkpoint before them, so nothing is ever integrated against the flow. With a checkpoint every $\sqrt{N}$ steps the backward sweep holds about $2\sqrt{N}$ states, 63 instead of 1000 for $N = 1000$, at the cost of one extra forward pass.

That makes three routes to a gradient. Which would we use? Backpropagation through the solver, with checkpoints when memory is short: it returns the exact gradient of the program that is actually run. The continuous adjoint is what that gradient converges to, and the formula to reason with. Re-integrating the state is a last resort, for fields that are not stiff.

38.3.3 Continuous Normalizing Flows

A Neural ODE moves points. A generative model trained by maximum likelihood must also say how probable each of its outputs is, so we now follow a density along the flow.

38.3.3.1 Density Along a Flow

Pushing samples through a flow turns it into a generator: draw $\mathbf{x}(0)$ from a simple density p_0 and integrate. Training such a generator by maximum likelihood needs the density of what comes out. A *normalizing flow* is an invertible network trained in this way, and its density follows from the change-of-variables formula of Section 36.2: for an invertible differentiable map $\mathbf{g}$ with Jacobian $\mathbf{J_g}$,

$$\log p_{\text{out}}(\mathbf{g}(\mathbf{x})) = \log p_{\text{in}}(\mathbf{x}) - \log \left|\det \mathbf{J_g}(\mathbf{x})\right| . \tag{38.20}$$

The log-density drops by the logarithm of the factor by which $\mathbf{g}$ stretches volume near $\mathbf{x}$. A normalizing flow built from many layers needs one log-determinant per layer and sample. A general determinant costs $O(d^3)$ operations, about 2×10^{10} for the $d = 3072$ of a 32×32 colour image. Discrete flows therefore restrict their layers to triangular Jacobians, whose determinant is the product of the diagonal, or to identity-plus-low-rank Jacobians, such as $\det(\mathbf{I} + \mathbf{u}\mathbf{v}^\top) = 1 + \mathbf{v}^\top\mathbf{u}$. Both determinants cost $O(d)$.

In continuous time the formula simplifies. Start with one dimension. The field $\dot{x} = \theta x$ on $\mathbb{R}$ has the flow $\Phi_t(x) = e^{\theta t}x$, which stretches every interval by $e^{\theta t}$. A density carried along is divided by the stretch, so $\log p_t(x(t)) = \log p_0(x_0) - \theta t$ and $\frac{d}{dt}\log p_t(x(t)) = -\theta$. The number θ is the 1×1 Jacobian of the field, and therefore also its trace.

The spiral is the running example in two dimensions. Its flow $e^{\mathbf{A}t} = e^{-t/2}e^{t\mathbf{K}}$ is a rotation followed by a shrink by $e^{-t/2}$ (Section 38.1). A disc is carried to a disc of $e^{-t/2}$ times its radius, so its area is multiplied by e^{-t}, and the density along each trajectory rises by e^{t}. Here $\operatorname{tr}\mathbf{A} = -1$, so the rate of change of the log-density is again minus the trace. Figure 38.10 draws this beside a field that preserves area and one that expands it: the darkest disc of the spiral, at $t = 1.2$, has $e^{-1.2} \approx 0.30$ of the area of the lightest.

38.3.3.2 The Instantaneous Change of Variables

Both examples were linear, and in both the log-density changed at the rate minus the trace. The general case needs one fact about determinants: over a short step the flow map is close to the identity, and the determinant of a matrix close to the identity is governed by its trace.

Lemma. *For any square matrix* $\mathbf{J}$, $\det(\mathbf{I} + h\mathbf{J}) = 1 + h\operatorname{tr}\mathbf{J} + O(h^2)$.

Proof. The permutation formula (Section 33.2.3.2) writes the determinant as a signed sum, over all permutations, of products with one entry from every row and every column. Any permutation other than the identity picks at least two off-diagonal entries, each of size $O(h)$, so it contributes $O(h^2)$. The identity permutation contributes $\prod_i (1 + hJ_{ii}) = 1 + h\sum_i J_{ii} + O(h^2)$. ■

Proposition (instantaneous change of variables). *Let* $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, t)$ *with* $\mathbf{f}$ *continuously differentiable in* $(\mathbf{x}, t)$ *and globally Lipschitz in* $\mathbf{x}$*, so that each flow map* Φ_t *is a* C^1 *bijection with a* C^1 *inverse (the solution of a* C^1 *field is* C^1 *in its initial value (Hairer et al. 1993), and the inverse*

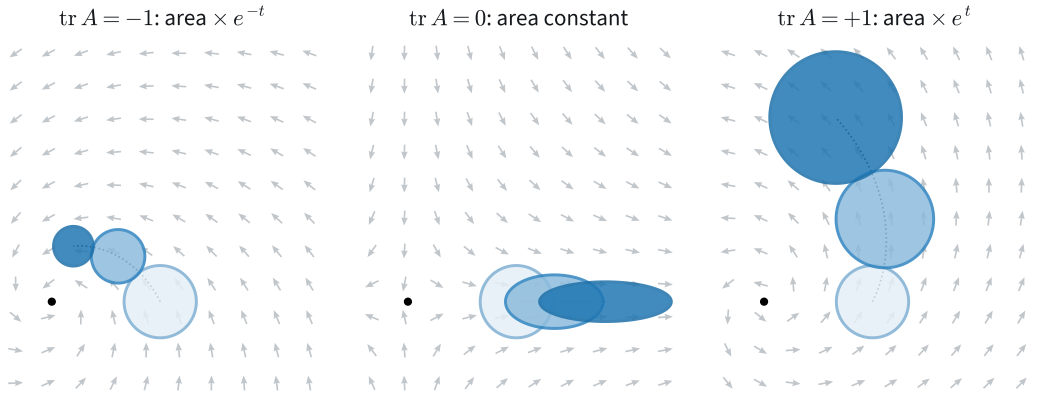


Figure 38.10: A disc of samples carried by the exact flows of three linear fields, shown at $t = 0, 0.6$ and 1.2 (darker is later); the dotted line is the path of the centre. The density along a trajectory changes by the factor $e^{-t\,\mathrm{tr}\,\mathbf{A}}$: it rises where the flow contracts (left, the spiral), stays constant under the area-preserving saddle although the shape changes (middle), and falls where the flow expands (right).

runs the field backward). Let $\mathbf{x}(0) \sim p_0$ *for a positive differentiable density* p_0*. Then the density* p_t *of* $\mathbf{x}(t)$ *exists and is positive, and along each trajectory* (Chen et al. 2018)

$$\frac{d}{dt} \log p_t(\mathbf{x}(t)) = -\,\mathrm{tr}\left(\frac{\partial \mathbf{f}}{\partial \mathbf{x}}\right)(\mathbf{x}(t), t). \tag{38.21}$$

Proof. The density exists by Equation 38.20 applied to the bijection Φ_t, and it is positive because p_0 is and the Jacobian of a map with a differentiable inverse is nonsingular. Over a short interval the flow advances by the map $\Phi_{t \to t+h}(\mathbf{x}) = \mathbf{x} + \int_t^{t+h} \mathbf{f}(\mathbf{x}(s), s)\, ds$. Differentiating in $\mathbf{x}$ and using the continuity of $\mathbf{J}$, its Jacobian is $\mathbf{I} + h\mathbf{J} + o(h)$, where $\mathbf{J} = \partial \mathbf{f}/\partial \mathbf{x}$ is evaluated at $(\mathbf{x}(t), t)$ and $o(h)$ stands for terms that vanish faster than h. Apply Equation 38.20 to this one map and expand with the Lemma, which tolerates the perturbation: replacing $h\mathbf{J}$ by $h\mathbf{J} + o(h)$ changes the determinant only by $o(h)$. For small h the determinant is close to 1, hence positive, so the absolute value can be dropped:

$$\begin{aligned} \log p_{t+h}(\mathbf{x}(t+h)) &= \log p_t(\mathbf{x}(t)) - \log \det(\mathbf{I} + h\mathbf{J} + o(h)) \\ &= \log p_t(\mathbf{x}(t)) - h\,\mathrm{tr}\,\mathbf{J} + o(h), \end{aligned}$$

using $\log(1 + u) = u + O(u^2)$. Subtract $\log p_t(\mathbf{x}(t))$, divide by h, and let $h \to 0$. ■

The hypotheses look heavier than they are. They ask only that every flow map be a smooth bijection, which a field with smooth Lipschitz activations such as tanh provides. The content of the proposition is the one line with the trace.

The trace of the Jacobian is the *divergence* $\nabla \cdot \mathbf{f}$ of the field, the local rate at which the flow expands volume. Where the field spreads samples apart ($\mathrm{tr}\,\mathbf{J} > 0$) the log-density along each trajectory falls, and where it compresses them, it rises. Integrating Equation 38.21 along a trajectory gives

$$\log p_T(\mathbf{x}(T)) = \log p_0(\mathbf{x}(0)) - \int_0^T \operatorname{tr}\!\left(\frac{\partial \mathbf{f}}{\partial \mathbf{x}}\right) dt. \tag{38.22}$$

The per-layer sum of log-determinants of a discrete flow has become an integral of traces. A model that computes it, by integrating the pair $(\mathbf{x}, \log p)$ with any ODE solver, is a **continuous normalizing flow** (CNF) (Chen et al. 2018). It needs no restriction on the architecture of $\mathbf{f}$. To evaluate the density of a given point $\mathbf{x}(T)$, integrate the field backward from it to $\mathbf{x}(0)$, accumulate the trace on the way, and read off p_0 there. Likelihoods of diffusion models are computed in this way (Section 17.6). Nothing in the formula prefers a direction of time: the generators of Section 38.6 and Section 38.7 put the noise at the final time and the data at $t = 0$, and the same formula serves them.

For a linear field the trace integral is $t \operatorname{tr}\mathbf{A}$, and Equation 38.22 compared with Equation 38.20 for the map $e^{\mathbf{A}t}$ says $\log\det e^{\mathbf{A}t} = t \operatorname{tr}\mathbf{A}$. For a diagonalizable $\mathbf{A} = \mathbf{V}\Lambda\mathbf{V}^{-1}$ the identity also follows in one line from the eigenvalues,

$$\det e^{\mathbf{A}t} = \det\left(\mathbf{V} e^{\Lambda t}\mathbf{V}^{-1}\right) = \prod_j e^{\Lambda_{jj} t} = e^{t \operatorname{tr}\mathbf{A}}, \tag{38.23}$$

and the comparison of the two formulas gives it for every $\mathbf{A}$.

The proposition follows one trajectory at a time. Stated for the whole density at once, the same law is the continuity equation of fluid mechanics, which Section 38.5 derives together with the term that noise adds.

38.3.3.3 Checking the Trace Integral

A linear field is a weak test of the proposition: its trace is constant, and the integral is just $t \operatorname{tr}\mathbf{A}$. A nonlinear field is a real one, because its trace changes along the path. Take the planar field $\mathbf{f}(\mathbf{x}) = \mathbf{W}_2 \tanh(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$ with 8 hidden units and random weights. Its Jacobian and trace are explicit:

$$\mathbf{J} = \mathbf{W}_2 \operatorname{diag}(\mathbf{s})\, \mathbf{W}_1, \qquad \mathbf{s} = 1 - \tanh^2(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1), \qquad \operatorname{tr}\mathbf{J} = \sum_i s_i \,(\mathbf{W}_1\mathbf{W}_2)_{ii},$$

using the cyclic property $\operatorname{tr}\left(\mathbf{W}_2 \operatorname{diag}(\mathbf{s})\, \mathbf{W}_1\right) = \operatorname{tr}\left(\operatorname{diag}(\mathbf{s})\, \mathbf{W}_1\mathbf{W}_2\right)$. The cell integrates the pair $(\mathbf{x}, \int \operatorname{tr}\mathbf{J}\, dt)$ with RK4 to $T = 1$ and keeps the path. It prints the smallest and the largest trace met on the way, and then compares the integral with $\log|\det D\Phi_T|$, where the Jacobian $D\Phi_T$ of the flow map is computed from central finite differences of the flow map itself.

```
n_hidden = 8
W1, b1 = rng.standard_normal((n_hidden, 2)), rng.standard_normal(n_hidden)
W2 = rng.standard_normal((2, n_hidden)) / np.sqrt(n_hidden)
field = lambda x: W2 @ np.tanh(W1 @ x + b1)
trJ = lambda x: np.sum((1 - np.tanh(W1 @ x + b1) ** 2)
                       * np.einsum('ij,ji->i', W1, W2))
aug = lambda z: np.append(field(z[:2]), trJ(z[:2]))  # d/dt (x, int tr J)
xa, N = rng.standard_normal(2), 200
```

```
zs = [np.append(xa, 0.0)]                       # RK4 path of (x, int tr J)
for _ in range(N):
    zs.append(rk4_step(aug, zs[-1], 1.0 / N))
zT, tr_path = zs[-1], [trJ(z[:2]) for z in zs]
D = np.zeros((2, 2))                            # D Phi_1 by central differences
for j in range(2):
    e = np.zeros(2); e[j] = 1e-5
    D[:, j] = (solve(rk4_step, field, xa + e, 1.0, N)
               - solve(rk4_step, field, xa - e, 1.0, N)) / 2e-5
logdet = np.log(abs(np.linalg.det(D)))
print(f'tr J along the path: from {min(tr_path):+.3f} to {max(tr_path):+.3f}')
print(f'trace integral {zT[2]:+.6f}  log|det D Phi| {logdet:+.6f}  '
      f'gap {abs(zT[2] - logdet):.1e}')
```

```
tr J along the path: from +0.889 to +1.013
trace integral +0.970187  log|det D Phi| +0.970187  gap 8.3e-11
```

The trace does move: the first line gives the interval that it covers along the path. How wide the interval is depends on the draw of the weights. A tenth or more is typical, and in some draws the interval contains zero, so that the flow expands area on one stretch of the path and compresses it on another. Yet the integral and the log-determinant agree to eight digits or better, the accuracy of the solver and of the finite differences. Nothing in the proposition used linearity.

38.3.3.4 The Hutchinson Trace Estimator

The formula for $\operatorname{tr}\mathbf{J}$ above used the structure of a one-layer field. For a general network the trace needs the d diagonal entries of $\mathbf{J}$, which cost d reverse-mode passes, one per coordinate. The trace has already replaced an $O(d^3)$ determinant by d passes. A random projection replaces the d passes by one, and returns an unbiased but noisy estimate (Hutchinson 1989).

Proposition (Hutchinson estimator). *For any matrix $\mathbf{M} \in \mathbb{R}^{d\times d}$ and any random vector $\epsilon \in \mathbb{R}^d$ with $\mathbb{E}[\epsilon\epsilon^\top] = \mathbf{I}$, such as a standard Gaussian or a Rademacher vector with independent entries ± 1 of equal probability,*

$$\operatorname{tr}(\mathbf{M}) = \mathbb{E}\left[\epsilon^\top \mathbf{M}\epsilon\right]. \tag{38.24}$$

Proof. $\mathbb{E}[\epsilon^\top\mathbf{M}\epsilon] = \mathbb{E}[\operatorname{tr}(\mathbf{M}\epsilon\epsilon^\top)] = \operatorname{tr}(\mathbf{M}\,\mathbb{E}[\epsilon\epsilon^\top]) = \operatorname{tr}(\mathbf{M})$, by the cyclic property of the trace and its linearity, which lets it commute with the expectation. ■

The estimator touches $\mathbf{J}$ only through $\epsilon^\top\mathbf{J}$, one vector–Jacobian product, which is one reverse-mode pass through $\mathbf{f}$ with ϵ as the seed; no Jacobian is formed. For the one-hidden-layer field the product is explicit,

$$\epsilon^\top\mathbf{J} = \left((\epsilon^\top\mathbf{W}_2)\odot\mathbf{s}\right)\mathbf{W}_1,$$

and it costs about as much as evaluating $\mathbf{f}$. A CNF evaluates one such product with every evaluation of the field, so the likelihood costs one additional backward pass per field evaluation. This is the method that Grathwohl et al. (2019) named FFJORD.

The proposition says that the average over probes converges to the trace. It does not say how fast, and for a single probe that is the whole question. With $\mathbf{S} = (\mathbf{J} + \mathbf{J}^\top)/2$, one Rademacher probe has variance $2\sum_{i \neq j} S_{ij}^2$ and one Gaussian probe $2\sum_{i,j} S_{ij}^2$, the same sum with the diagonal included (Exercise 5). The trace sees only the diagonal of $\mathbf{J}$, and the noise of a Rademacher probe only what lies off it.

Let us try the estimator on a field in $d = 50$ whose trace we can predict: a random one-hidden-layer part with 64 tanh units and a pull toward the origin, $\mathbf{f}(\mathbf{x}) = \mathbf{W}_2 \tanh(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1) - \mathbf{x}$. Its Jacobian is $\mathbf{W}_2 \operatorname{diag}(\mathbf{s})\, \mathbf{W}_1 - \mathbf{I}$, so the pull contributes -50 to the trace and the random part a number of order one. The pull also adds $-\epsilon^\top \epsilon$ to every estimate. For a Rademacher probe this is -50 exactly, and all the noise comes from the random part. For a Gaussian probe $\epsilon^\top \epsilon$ is itself random, and the two variance formulas say what that costs. They differ by the diagonal term $2\sum_i S_{ii}^2$, and here every $S_{ii} = J_{ii}$ is close to -1, so the variance of a Gaussian probe should exceed that of a Rademacher probe by about $2 \cdot 50 = 100$. The cell draws $m = 1000$ probes of each kind and computes each $\epsilon^\top \mathbf{J}$ by reverse mode.

```
d, n_hidden, m = 50, 64, 1000
W1 = rng.standard_normal((n_hidden, d)) / np.sqrt(d)
b1 = 0.1 * rng.standard_normal(n_hidden)
W2 = rng.standard_normal((d, n_hidden)) / np.sqrt(n_hidden)
x = rng.standard_normal(d)
s = 1 - np.tanh(W1 @ x + b1) ** 2
tr_tanh = np.sum(s * np.einsum('ij,ji->i', W1, W2))  # trace of W2 diag(s) W1
E = rng.choice([-1.0, 1.0], size=(m, d))         # Rademacher probes, one per row
E_gauss = rng.standard_normal((m, d))            # Gaussian probes, one per row
```

We now evaluate the reverse-mode vector–Jacobian product $\epsilon_k^\top \mathbf{J}$ at `x` for all $m = 1000$ Rademacher (`E`) and Gaussian (`E_gauss`) probes and compare the resulting trace estimates $\epsilon_k^\top \mathbf{J} \epsilon_k$ and their single-probe standard deviations against the exact trace `tr_tanh - d`:

```
W1t, b1t, W2t = (torch.tensor(w) for w in (W1, b1, W2))
field = lambda x: W2t @ torch.tanh(W1t @ x + b1t) - x
_, vjp = torch.func.vjp(field, torch.tensor(x))      # reverse mode at x
for name, P in (('Rademacher', E), ('Gaussian', E_gauss)):
    P = torch.tensor(P)
    rows = torch.func.vmap(vjp)(P)[0]                # row k: eps_k^T J
    est = (rows * P).sum(dim=1)                      # eps_k^T J eps_k
    sd = est.std(correction=0)
    print(f'{name:10s}: exact {tr_tanh - d:+6.2f}, mean of {m} probes '
          f'{est.mean():+6.2f} +- {sd / m ** 0.5:.2f}, one probe sd {sd:.2f}')
```

```
Rademacher: exact -51.30, mean of 1000 probes -51.23 +- 0.16, one probe sd 4.99
Gaussian  : exact -51.30, mean of 1000 probes -51.16 +- 0.37, one probe sd 11.62
```

Both averages lie within a few of their standard errors of the exact trace, as unbiased estimates should. One Rademacher probe has a standard deviation of about 5, a tenth of the trace: a single backward pass locates the trace to ten percent, where the exact value takes 50 passes. One Gaussian probe has a standard deviation of about 11, and $11^2 - 5^2 = 96$, close to the predicted 100. Rademacher probes are never the noisier kind, and here they are much the quieter.

The ten percent belongs to this field, not to the estimator. Without the pull every Rademacher estimate would shift by exactly $+50$: the standard deviation would still be 5, the trace would be of order one, and a single probe would say next to nothing about it. Which of the two cases a model in high dimension resembles is the subject of Exercise 5.

A likelihood needs more than one trace. It needs the integral $\int_0^T \operatorname{tr} \mathbf{J}\, dt$ along a trajectory, one estimate per solver step, and the probes can be supplied in two ways: one probe held for the whole solve, or a fresh one at every step. What should we expect? A held probe sees the sum $\sum_n h\,\mathbf{J}(\mathbf{x}_n)$ as one matrix and estimates its trace once, so its noise is that of a single estimate whatever N is. Fresh probes make independent errors, each weighted by $h = 1/N$, so the noise of their sum falls like $1/\sqrt{N}$. The next cell follows one Euler trajectory of the same field to $T = 1$ with $N = 10$, 100 and 1000 steps and repeats each policy 200 times with Rademacher probes, taking $\epsilon^\top \mathbf{J}$ from the explicit product above.

```
field = lambda x: W2 @ np.tanh(W1 @ x + b1) - x
diag_W1W2 = np.einsum('ij,ji->i', W1, W2)
for N in (10, 100, 1000):
    h, xs = 1 / N, [x]
    for _ in range(N - 1):                    # Euler trajectory x_0, ..., x_{N-1}
        xs.append(xs[-1] + h * field(xs[-1]))
    S = 1 - np.tanh(np.array(xs) @ W1.T + b1) ** 2  # s at every step, (N, n_hidden)
    exact = h * (S @ diag_W1W2).sum() - d     # h * sum_n tr J(x_n)
    held = np.broadcast_to(rng.choice([-1.0, 1.0], (200, 1, d)), (200, N, d))
    fresh = rng.choice([-1.0, 1.0], (200, N, d))
    row = f'N = {N:4d}: exact {exact:+.2f}'
    for name, P in (('held probe', held), ('fresh probes', fresh)):
        est = h * np.einsum('rnh,nh,rnh->r', P @ W2, S, P @ W1.T) - d  # eps^T eps = d
        row += f' | {name} {est.mean():+.2f}, sd {est.std():.2f}'
    print(row)
```

```
N =   10: exact -51.90 | held probe -52.11, sd 5.69 | fresh probes -51.73, sd 1.71
N =  100: exact -51.93 | held probe -51.35, sd 5.98 | fresh probes -51.94, sd 0.63
N = 1000: exact -51.93 | held probe -51.28, sd 5.88 | fresh probes -51.92, sd 0.19
```

The exact integral is again close to -50, and the means of both policies agree with it to within sampling error: the estimate is unbiased either way. The spreads are as predicted. A held probe has a standard deviation of about 5 to 6 at every N. Fresh probes come to between 1.5 and 1.8 at $N = 10$, about 0.5 to 0.6 at $N = 100$ and just under 0.2 at $N = 1000$: in each row, the held value divided by $\sqrt{N}$.

Fresh probes are about thirty times quieter at $N = 1000$ and cost the same. Why would anyone hold the probe? Because a held probe leaves the solver an ordinary differential equation. With ϵ fixed, the estimate $\epsilon^\top \mathbf{J}(\mathbf{x})\epsilon$ is a deterministic function of the state, so any solver applies, an adaptive one can control its error, and the adjoint equations give the gradient. With a fresh probe at every evaluation the right-hand side is noise, and a solver that compares two evaluations to judge its step would be comparing two random draws. Our cell could use fresh probes because its Euler steps were fixed in advance. FFJORD holds the probe, and keeps the noise.

Training can live with that noise, because training never needs an accurate likelihood, only an unbiased one. The estimate enters the log-likelihood linearly, so the estimated log-

likelihood and its gradient are unbiased too, and a stochastic gradient asks for nothing more (Section 35.4). The noise of the probes is averaged over the minibatch and over iterations, as the noise of the minibatch itself is. What one held probe cannot give is the likelihood of a single point. An error of 5 in a log-likelihood is a factor of $e^5 \approx 150$ in the density, so for that one averages many probes, draws fresh ones along a fixed grid of steps, or computes the exact trace.

The only noise in this section sat in our probes. The trajectories themselves were deterministic. A diffusion model adds fresh noise to the state at every step, and increments that scale like $\sqrt{h}$ break the Taylor expansions this section relied on. Integrating such an equation takes a calculus of its own (Section 38.4).

38.3.4 Summary

A residual block is an Euler step, and one field with shrinking steps gives a Neural ODE, whose exact flow is invertible and whose step count is left to a solver. Both gains have limits: on $\mathbb{R}$ a flow preserves order, so $x \mapsto -x$ needs a second dimension, and a field trained through ten Euler steps fits the ten-step program, where one trained through RK4 fits the flow at four times the field evaluations. The gradient of a loss on $\mathbf{x}(T)$ is the integral of $(\partial \mathbf{f}/\partial\theta)^\top \mathbf{a}$ along the trajectory, where the adjoint $\mathbf{a}$ solves a linear ODE backward from $\nabla \ell(\mathbf{x}(T))$; backpropagation through Euler is exact for the program and within $O(h)$ of this gradient. Re-integrating the state backward saves memory but amplifies a stored error by $e^{\lambda T}$ for a mode that decays at rate λ. Along a trajectory $\frac{d}{dt}\log p_t = -\operatorname{tr}\mathbf{J}$, a trace where a discrete flow needs a determinant. Hutchinson's estimator obtains the trace from one vector–Jacobian product per probe, without bias but with noise that training averages out and a single likelihood does not.

38.3.5 Exercises

1. code **What a flow cannot represent.** ● Two trajectories of a field with Lipschitz constant L satisfy $e^{-Lt}\|\mathbf{x}-\mathbf{y}\| \le \|\Phi_t(\mathbf{x})-\Phi_t(\mathbf{y})\| \le e^{Lt}\|\mathbf{x}-\mathbf{y}\|$ (Exercise 2 of Section 38.1), and Section 38.3.1.2 showed that no flow on $\mathbb{R}$ reverses order (Dupont et al. 2019).
    1. Let $d = 1$. Use the lower bound to give a second proof that Φ_T is strictly increasing, and show that $\Phi_T(1) - \Phi_T(-1) \ge 2e^{-LT}$. Which of the two statements, order or distance, does the proposition of the text give, and which needs the bound?
    2. The training example carries the circle points $(0, \pm 1)$, which are 2 apart, to $(1.5, \pm 0.4)$, which are 0.8 apart. Show that the exact time-T flow of a field with Lipschitz constant L can do this only if $LT \ge \ln 2.5 \approx 0.92$, whereas the long axis, stretched from 2 to 2.4, gives only $LT \ge \ln 1.2$. A one-hidden-layer tanh field has $L \le \|\mathbf{W}_2\|_2\|\mathbf{W}_1\|_2$ because tanh is 1-Lipschitz. Compute this bound for the RK4-trained field and check the inequality.
    3. Embed $\mathbb{R}$ in $\mathbb{R}^2$ as $x \mapsto (x, 0)$. Show that every linear field $\dot{\mathbf{x}} = \mathbf{C}\mathbf{x}$ whose time-one map sends $(x, 0)$ to $(-x, 0)$ for every x has $e^{\mathbf{C}} = -\mathbf{I}$. *Hint: an eigenvalue -1 of $e^{\mathbf{C}}$ comes from an eigenvalue μ of $\mathbf{C}$ with $e^{\mu} = -1$, which is not real.* Find such a field whose trajectories are ellipses rather than circles.
    4. Change the training cell to fit the map $x \mapsto -x$ on 64 evenly spaced points of $[-1, 1]$ with a field of input and output dimension 1 and $N = 10$ Euler steps. What does the trained model output, and why does the loss settle at a value you

can predict before training? Repeat with a field of dimension 2, inputs $(x, 0)$, and the loss on the first coordinate.

5. Train the one-dimensional model once more with $N = 1$ and $h = 1$, and evaluate it with $N = 1$ and with $N = 100$ steps of size $h = 0.01$. Relate the two results to the proposition and to the condition $hL < 1$.

2. **Adjoint with a running loss.** ● The adjoint equations Equation 38.17 and Equation 38.18 treat a loss on the endpoint. Regularizers such as the kinetic energy $\int_0^T \|\mathbf{f}(\mathbf{x}(t), \theta, t)\|^2 \, dt$ of Finlay et al. (2020) accumulate a cost along the trajectory. Let

$$\mathcal{L} = \ell_T(\mathbf{x}(T)) + \int_0^T \ell(\mathbf{x}(t), \theta, t) \, dt.$$

   1. Extend the variational proof of Equation 38.17 to show that the adjoint acquires a source term, $\dot{\mathbf{a}} = -(\partial \mathbf{f}/\partial \mathbf{x})^\top \mathbf{a} - \nabla_{\mathbf{x}} \ell$ with $\mathbf{a}(T) = \nabla \ell_T(\mathbf{x}(T))$, and that $\partial \mathcal{L}/\partial \theta = \int_0^T \left((\partial \mathbf{f}/\partial \theta)^\top \mathbf{a} + \nabla_\theta \ell \right) dt$.
   2. Finlay et al. (2020) instead append a coordinate E with $\dot{E} = \ell$ and $E(0) = 0$ to the state, and add $E(T)$ to the endpoint loss. Apply Equation 38.17 to the extended state $(\mathbf{x}, E)$ and show that it reproduces (a).
   3. For the kinetic energy, $\ell = \|\mathbf{f}\|^2$. Write $\nabla_{\mathbf{x}} \ell$ as a vector–Jacobian product. How many reverse-mode passes through $\mathbf{f}$ does one step of the adjoint ODE then require?

3. code **Reconstructing the trajectory backward.** ● The continuous adjoint can avoid storing the forward states by integrating $\mathbf{x}(t)$ backward from $\mathbf{x}(T)$ alongside $\mathbf{a}(t)$. Section 38.3.2 found that a mode which decays at the rate λ then amplifies an error δ in the stored $\mathbf{x}(T)$ to $e^{\lambda T}\delta$.

   1. Let $\mathbf{R}$ be the rotation by 0.6 radians and $\mathbf{C} = \mathbf{R}\,\mathrm{diag}(-50, -1)\,\mathbf{R}^\top$. Integrate $\dot{\mathbf{x}} = \mathbf{C}\mathbf{x}$ from $(1, 1)$ to time T with 2000 RK4 steps (`solve` with `rk4_step`), then integrate backward from the result for the same time, with a negative horizon. Report the reconstruction error for $T = 0.25$, 0.5, 1 and 2, and compare its growth with $e^{50T}\delta$. Which δ fits? Why is it nearer 10^{-17} than the 10^{-16} of the text? *Hint: how large are the coordinates of* $\mathbf{x}(T)$*?*
   2. Repeat with $\mathrm{diag}(-50, -1)$ itself. From $T = 0.5$ on it reconstructs the start far more accurately than $\mathbf{C}$ does. Why?
   3. Activation checkpointing (Section 5.7.3.3) stores the state at the start of each segment of the trajectory. A hybrid scheme reconstructs the states within a segment by integrating backward from the checkpoint at its end. For the matrix $\mathbf{C}$ of (a), what is the longest segment duration for which the drift $e^{50T}\delta$ with $\delta = 10^{-16}$ stays below 10^{-8}?

4. **Log-determinant of a residual step.** ●● The instantaneous change of variables Equation 38.21 is a limit of infinitesimal steps. A finite residual step $F(\mathbf{x}) = \mathbf{x} + \mathbf{g}(\mathbf{x})$ needs its exact log-determinant, as in the invertible residual networks of Behrmann et al. (2019) .

   1. Let $\mathbf{g}$ be Lipschitz with constant $c < 1$. For $d = 1$ the one-line argument of Section 38.3.1.2 (an Euler step is increasing when $hL < 1$) shows that F is strictly increasing, hence injective. For general d, write $F(\mathbf{x}) = \mathbf{y}$ as a fixed-point equation for $\mathbf{x}$ and use the Banach fixed-point theorem, as in the proof of Picard–Lindelöf,

to show that F is a bijection of $\mathbb{R}^d$. How fast does the fixed-point iteration converge?

2. Show that $\|\mathbf{J_g}\| \le c$, where $\mathbf{J_g}$ is the Jacobian of $\mathbf{g}$, and use the eigenvalues of $\mathbf{J_g}$ to show that

$$\log\det(\mathbf{I}+\mathbf{J_g}) = \sum_{k=1}^{\infty} \frac{(-1)^{k+1}}{k} \operatorname{tr}(\mathbf{J_g}^k).$$

3. An Euler step has $\mathbf{g} = h\mathbf{f}$ and $\mathbf{J_g} = h\mathbf{J}$. Compare $\log\det(\mathbf{I}+h\mathbf{J})$ with the term $h \operatorname{tr}\mathbf{J}$ that Equation 38.22 accumulates per step. How large is the difference per step, and how large is it in total over T/h steps?

4. The Hutchinson estimator Equation 38.24 estimates $\operatorname{tr}(\mathbf{J_g}^k)$ by $\epsilon^\top \mathbf{J_g}^k \epsilon$. How many vector–Jacobian products per probe does the series truncated after K terms cost?

5. code **Variance of the Hutchinson estimator.** ● The estimator Equation 38.24 estimates $\operatorname{tr}\mathbf{M}$ by $\epsilon^\top\mathbf{M}\epsilon$ for a single probe ϵ. Let $\mathbf{S} = (\mathbf{M}+\mathbf{M}^\top)/2$, so that $\epsilon^\top\mathbf{M}\epsilon = \epsilon^\top\mathbf{S}\epsilon$.

   1. Show that the variance of $\epsilon^\top\mathbf{M}\epsilon$ is $2\|\mathbf{S}\|_F^2$ for Gaussian probes and $2\sum_{i\neq j} S_{ij}^2$ for Rademacher probes. For which matrices is a single Rademacher probe exact?
   2. Form the Jacobian $\mathbf{J} = \mathbf{W}_2 \operatorname{diag}(\mathbf{s})\, \mathbf{W}_1 - \mathbf{I}$ of the field with the pull explicitly, from the weights and the point of the Hutchinson experiment (it is 50×50). Evaluate both variances and compare their square roots with the two one-probe standard deviations that the experiment printed. How many probes does each distribution need for a standard error of 0.01?
   3. In a CNF, $\mathbf{M}$ is the Jacobian $\partial\mathbf{f}/\partial\mathbf{x}$ in dimension d. Suppose that all entries have comparable magnitude. How does the relative standard error $\sqrt{\operatorname{Var}}/|\operatorname{tr}\mathbf{M}|$ of one probe scale with d when the diagonal entries share one sign, and when their signs are random?
   4. In training, each of the B points of a minibatch has its own trajectory and its own held probe. How does the noise of the average log-likelihood depend on B? For which B does it match the noise that one point has with fresh probes at $N = 100$ steps?

6. code **Discrete and continuous adjoints.** ●● Backpropagation through a solver differentiates the executed program, whereas a solver applied to the adjoint ODE Equation 38.17 approximates the gradient of the exact flow (Kidger 2022). Compare the two recursions step by step for an autonomous field with Jacobian $\mathbf{J}(\mathbf{x}) = \partial\mathbf{f}/\partial\mathbf{x}$ and forward states $\mathbf{x}_n$.

   1. For forward Euler, the discrete adjoint Equation 38.19 computes $\mathbf{a}_n = (\mathbf{I} + h\mathbf{J}(\mathbf{x}_n))^\top \mathbf{a}_{n+1}$, whereas an explicit Euler step of the adjoint ODE from t_{n+1} back to t_n computes $\mathbf{a}_n = \mathbf{a}_{n+1} + h\mathbf{J}(\mathbf{x}_{n+1})^\top \mathbf{a}_{n+1}$. Bound the difference of the two updates for a smooth $\mathbf{f}$. For a linear field the two coincide, and the text said that the gap to the gradient of the flow is then Euler's own error. Make this precise for the scalar program $x_N = (1+h\theta)^N x_0$: show that the recursion computes the gradient $T\,x_N^2/(1+h\theta)$ that the text found, and that this equals $T\,x(T)^2(1-h\theta(1+\theta T)) + O(h^2)$.
   2. Differentiate one Heun step $\mathbf{x}_{n+1} = \mathbf{x}_n + \frac{h}{2}[\mathbf{f}(\mathbf{x}_n)+\mathbf{f}(\mathbf{u}_n)]$ with $\mathbf{u}_n = \mathbf{x}_n + h\mathbf{f}(\mathbf{x}_n)$ to obtain its discrete adjoint recursion. Show that this recursion is a Heun step of

the adjoint ODE from t_{n+1} to t_n in which $\mathbf{J}(\mathbf{x}_{n+1})$ is replaced by $\mathbf{J}(\mathbf{u}_n)$. At which order in h do the two updates differ?

3. For a linear field $\mathbf{f}(\mathbf{x}) = \mathbf{A}\mathbf{x}$, one RK4 step multiplies the state by $R(h\mathbf{A})$, where $R(z) = 1 + z + z^2/2 + z^3/6 + z^4/24$ is the stability function of RK4 (Section 38.2; Exercise 2 there studies it on the imaginary axis). Show that the discrete adjoint of RK4 is an RK4 step of the adjoint ODE.
4. Replace the Euler step of `loss_and_grad_unrolled` by an RK4 step, and compute the gradient of the unrolled loss by central finite differences instead of a hand-written backward sweep. Measure its gap to the continuous-adjoint gradient `g_cont` for $N = 5, 10, 20, 40$ and 80 steps. The trapezoid rule that computes `g_cont` uses `n_quad = 2000` intervals and is accurate only to about 10^{-7}, so refine it first. Which slope do you measure? Which order does the gradient computed through a solver of order p inherit?

7. code **Liouville's formula.** ●● The instantaneous change of variables followed a density along a trajectory. This exercise proves it a second way, through the Jacobian $\mathbf{M}(t) = D\Phi_t(\mathbf{x}_0)$ of the flow map itself.

   1. Differentiate $\frac{d}{dt}\Phi_t(\mathbf{x}_0) = \mathbf{f}(\Phi_t(\mathbf{x}_0), t)$ with respect to $\mathbf{x}_0$ to obtain the variational equation $\dot{\mathbf{M}} = \mathbf{J}(\mathbf{x}(t), t)\,\mathbf{M}$ with $\mathbf{M}(0) = \mathbf{I}$.
   2. Prove Jacobi's formula $\frac{d}{dt}\det\mathbf{M} = \operatorname{tr}(\mathbf{M}^{-1}\dot{\mathbf{M}})\det\mathbf{M}$ for invertible $\mathbf{M}(t)$. *Hint: write* $\det\mathbf{M}(t+h) = \det\mathbf{M}(t)\,\det\left(\mathbf{I} + h\mathbf{M}^{-1}\dot{\mathbf{M}} + O(h^2)\right)$ *and use the Lemma.*
   3. Conclude that $\log|\det D\Phi_T(\mathbf{x}_0)| = \int_0^T \operatorname{tr}\mathbf{J}(\mathbf{x}(t), t)\,dt$, and derive Equation 38.22 from the discrete change of variables Equation 38.20 applied to the bijection Φ_T. For a linear field the result is the formula Equation 38.23.
   4. Integrate $\dot{\mathbf{M}} = \mathbf{J}\mathbf{M}$ with RK4 alongside the 8-unit tanh field of Section 38.3.3, in the cell that checks the trace integral (before the Hutchinson experiment reuses the names of its weights), and compare $\log|\det\mathbf{M}(1)|$ with the trace integral and the log-determinant that cell prints.

38.4 Stochastic Differential Equations

Every trajectory in the previous sections was fixed by its start, and a flow never forgets where it came from: two starts always reach two different ends. The diffusion models of Chapter 17 are built on a process that does forget. Their *forward process* mixes new noise into its state, a little at a time, until nothing of the start remains, and generation, which runs it backward, has a known place to begin. Such a process solves a differential equation too, but not one that ordinary calculus can handle.

Let us begin with a case small enough to follow by hand. Take one number, $x_0 = 2$. At each step keep 99% of it and add an independent Gaussian draw of standard deviation $\sqrt{0.02} \approx 0.14$:

$$x_{k+1} = 0.99\,x_k + \sqrt{0.02}\,\xi_k, \qquad \xi_k \sim \mathcal{N}(0, 1).$$

After 400 steps the mean is $2 \cdot 0.99^{400} \approx 0.036$, and the variance is close to 1, where the 2% that a step removes balances the 0.02 that it adds. Any other start ends in the same place, a draw from nearly $\mathcal{N}(0, 1)$. One feature of the recurrence looks odd. If it describes a process in continuous time with time step $h = 0.01$, then the decay $0.99 = 1 - h$ changes

x by an amount of order h, but the noise has size $\sqrt{h}$, not h. This choice is forced. Noise of size h would vanish as $h \to 0$, while noise of size $\sqrt{h}$ survives, and its square is of order h, the same order as the drift. Ordinary calculus discards squares of small changes, so this one fact requires a new calculus.

The limit of such noise is Brownian motion, named after the botanist Robert Brown, who in 1827 watched particles from pollen grains jitter in water. Norbert Wiener made it a mathematical object in 1923, and Kiyoshi Itô built its calculus in the 1940s (Itô 1944) on one rule: the squares of short Brownian increments add up to the elapsed time, so the chain rule gains a second-derivative term.

We build Brownian motion from coin flips and the calculus from that rule. Then we solve the process behind the recurrence, the forward process of Section 17.2. Given the start, its distribution at every time is a Gaussian, so a diffusion model can noise its data to any level in one draw. Simulation, and the measurement of its error, can wait for a second reading. Only the rule for squares is hard to believe, and its proof takes six lines.

38.4.1 Brownian Motion

38.4.1.1 Noise as a Forward Process

Before building any machinery, let us check what a forward process must do. Starting from data distributed as $p_0 = p_{\text{data}}$, it must end at one known distribution, whatever the data looked like,

$$p_0 = p_{\text{data}} \longrightarrow p_T \approx \mathcal{N}(\mathbf{0}, \mathbf{I}),$$

so that generation always knows where to start. No fixed flow does this for two datasets at once. A flow is an invertible map Φ (Section 38.1), so if it carried two distributions p and q to the same one, applying Φ^{-1} would show $p = q$.

Let us try three processes on two "datasets" that are as simple as datasets can be: point masses at $x_0 = 2$ and at $x_0 = -2$.

- **Drift alone**, $\dot{x} = -x$. The solution $x(t) = x_0 e^{-t}$ puts the two starts at $\pm 2e^{-3} = \pm 0.0996$ at $t = 3$. They approach the point 0, which is not a Gaussian, and at every finite time they are still two distinct points.
- **Noise alone**, $dX = \sqrt{2}\, dW$, which means $X_t = x_0 + \sqrt{2}\, W_t$ for a noise W whose variance grows linearly in time (the next subsection constructs it). The distribution of X_t, its *law*, is $\mathcal{N}(x_0, 2t)$: each cloud keeps its center at ± 2 while its standard deviation $\sqrt{2t}$ grows without bound (2.45 at $t = 3$). The start is forgotten only relative to this growing width, and there is no limiting law.
- **Both together**, $dX = -X\, dt + \sqrt{2}\, dW$. From either start the law at $t = 3$ is $\mathcal{N}(\pm 0.0996,\ 0.9975)$, as Section 38.4.3 proves. The two laws are nearly indistinguishable from each other and from $\mathcal{N}(0, 1)$.

Figure 38.11 draws the three. Only the first fails outright. The second is usable, and is in use. Once its variance $2t$ is far larger than that of the data, the start hardly matters. Processes of this kind are called *variance-exploding*, because without a limiting law they must make their final variance huge. The third is the running example of this section, and the recurrence of the opening simulates it. Its drift pulls each start toward zero while the noise refills the variance that the drift removes, and the two balance at variance 1.

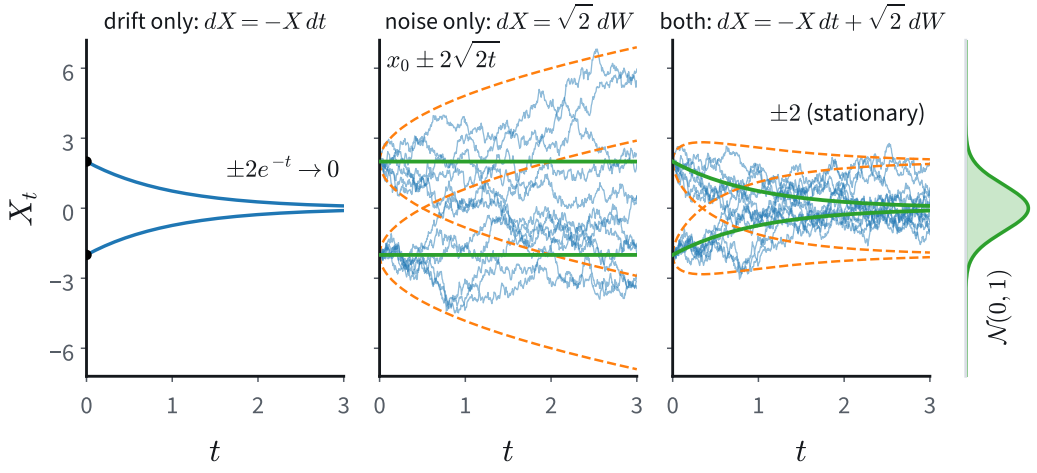


Figure 38.11: One failure, one usable process and the one we use, from the two starts $x_0 = \pm 2$. Drift alone sends every start to the point 0. Noise alone keeps each start as the center of a cloud whose two-standard-deviation envelope $x_0 \pm 2\sqrt{2t}$ (dashed) grows without bound. Both together carry every start into the same limiting law $\mathcal{N}(0,1)$, drawn sideways at the right. Green lines are the means.

In general, a forward process of this kind is a deterministic vector field plus a noise term. For data in $\mathbb{R}^d$ we write

$$d\mathbf{X} \;=\; \underbrace{\mathbf{f}(\mathbf{X}, t)\, dt}_{\text{drift}} \;+\; \underbrace{g(t)\, d\mathbf{W}}_{\text{diffusion}}, \tag{38.25}$$

which reads: over a short time dt, the state moves by the velocity $\mathbf{f}$ times dt and receives a random kick whose size is set by the noise amplitude g. The drift steers the mean, and the diffusion injects variance. So far the kick $d\mathbf{W}$ is only a symbol. We give it a meaning in one coordinate, and return to $\mathbb{R}^d$ only at the end of the Itô calculus.

38.4.1.2 From Random Walks to the Wiener Process

We put an object behind the symbol by taking the limit of the simplest random process there is, a walk driven by coin flips. The limit explains the square root in our recurrence.

Fix a step duration Δt and let a walker take independent steps of size $\pm\sqrt{\Delta t}$, each sign chosen by a fair coin. After k steps, at time $t = k\,\Delta t$, the position is a sum of k independent mean-zero steps of variance Δt apiece. Variances of independent terms add (Section 36.2.2.1), so

$$\mathbb{E}\big[W_t^{(\Delta t)}\big] = 0, \qquad \mathrm{Var}\big(W_t^{(\Delta t)}\big) = k\,\Delta t = t,$$

whatever the step size. With $\Delta t = 0.01$, for instance, 100 steps of ± 0.1 give variance $100 \cdot 0.01 = 1$ at time 1. Steps of ± 0.01 on the same clock would give variance 0.01, and steps of ± 1 would give 100. In general, steps of size $c\,\Delta t^{a}$ give variance $c^2\, t\, \Delta t^{2a-1}$ at time

t, which tends to 0 for $a > \frac{1}{2}$ and to ∞ for $a < \frac{1}{2}$. Only the square root leaves a finite nonzero variance in the limit, which is why it returns in the noise term of every numerical scheme below.

As $\Delta t \to 0$ the number of steps before time t grows without bound, and the central limit theorem (Section 36.3.2.3) makes the position Gaussian: a sum of many independent mean-zero terms with total variance t is approximately $\mathcal{N}(0, t)$.

The limit process is called **Brownian motion** or the **Wiener process** (Wiener 1923). It is defined by four properties:

1. $W_0 = 0$;
2. **independent increments**: for $s < t$, the increment $W_t - W_s$ is independent of the entire path up to time s;
3. **Gaussian increments**: $W_t - W_s \sim \mathcal{N}(0,\, t - s)$;
4. continuous sample paths.

That a process with these properties exists is Wiener's theorem, and that the rescaled walks converge to it is Donsker's theorem; we take both as given. The process spreads with standard deviation $\sqrt{t}$ forever (Figure 38.12), and its increments have the form

$$\Delta W \;=\; W_{t+\Delta t} - W_t \;=\; \sqrt{\Delta t}\,\xi, \qquad \xi \sim \mathcal{N}(0,1), \tag{38.26}$$

a standard normal draw scaled by $\sqrt{\Delta t}$.

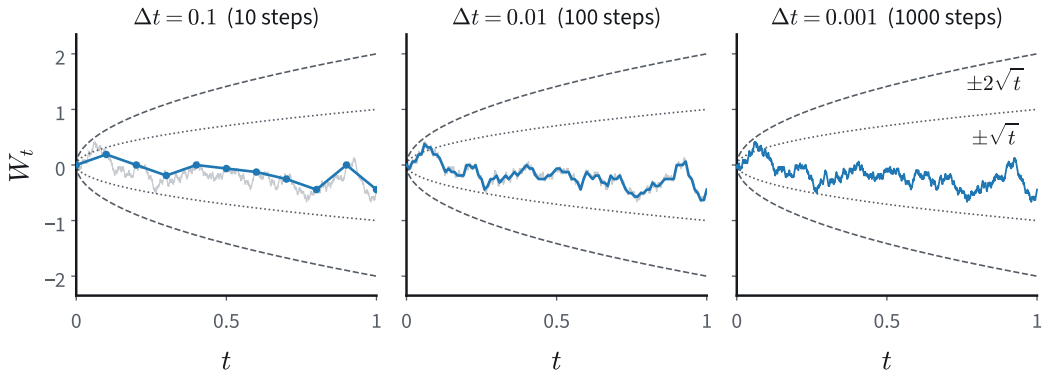


Figure 38.12: One coin-flip walk of 1000 steps $\pm\sqrt{0.001}$, viewed on meshes $\Delta t = 0.1, 0.01$ and 0.001 (faint gray: the full path). Each view is itself a random walk with step variance Δt; refining the mesh fills in detail without changing the large-scale path. At each fixed time the spread follows the $\pm\sqrt{t}$ and $\pm 2\sqrt{t}$ envelopes.

Independent increments also fix how the values of W at two times are related. For $s \le t$ write $W_t = W_s + (W_t - W_s)$. The increment is independent of W_s and has mean zero, so $\operatorname{Cov}(W_s, W_t) = \mathbb{E}[W_s^2] = s = \min(s, t)$: the walk carries its position at time s forward and adds noise that is uncorrelated with it. The formula is worth more than its one-line proof suggests. Jointly Gaussian values are determined by their means and covariances, so a process with jointly Gaussian values at any finite set of times, mean zero, covariance $\min(s, t)$ and continuous paths is a Brownian motion.

The same increments make the path rough. Brownian paths are continuous but nowhere differentiable, a theorem of Paley, Wiener, and Zygmund (Paley et al. 1933). The increment scaling Equation 38.26 gives the heuristic reason:

$$\left|\frac{\Delta W}{\Delta t}\right| = \frac{|\xi|}{\sqrt{\Delta t}} \longrightarrow \infty \quad \text{in probability as } \Delta t \to 0,$$

so a difference quotient grows without bound as the step shrinks. This is not a proof (it rules out a derivative at each fixed time, while "nowhere" concerns all times at once), but it shows the obstruction.

Rescaling time by a factor c and space by $\sqrt{c}$ returns a Brownian motion (Exercise 2), so zooming in on a path shows the same roughness at every scale, where a differentiable curve would flatten into its tangent. Hence dW/dt does not exist, and dW has to be given a meaning inside integrals. The Itô calculus supplies that meaning.

38.4.1.3 Simulation of Brownian Paths

The increment form Equation 38.26 is also a simulation recipe: take cumulative sums of $\sqrt{\Delta t}\,\xi$ steps. We generate $20{,}000$ paths on $[0, 1]$ in 500 steps and put a question to them. At any one time $W_t \sim \mathcal{N}(0, t)$, so 95.4% of the paths lie inside $\pm 2\sqrt{t}$, the outer envelope of Figure 38.12. What fraction stays inside the envelope at all times? It is worth a guess before running the cell.

```
rng = np.random.default_rng()
T_bm, n_bm, n_paths = 1.0, 500, 20000
dt_bm = T_bm / n_bm
dW_bm = np.sqrt(dt_bm) * rng.standard_normal((n_paths, n_bm))
W_bm = np.cumsum(dW_bm, axis=1)              # W at t = dt, 2 dt, ..., 1
band = 2 * np.sqrt(dt_bm * np.arange(1, n_bm + 1))  # 2 sqrt(t) on the grid
inside = np.abs(W_bm) < band
print(f'inside +-2 sqrt(t) at t = 1:   {inside[:, -1].mean():.3f}'
      '   (Gaussian: 0.954)')
for stride in (10, 1):                       # same paths, coarse then fine grid
    print(f'inside at all {n_bm // stride} grid times: '
          f'{inside[:, stride - 1::stride].all(axis=1).mean():.3f}')
```

```
inside +-2 sqrt(t) at t = 1:   0.954   (Gaussian: 0.954)
inside at all 50 grid times: 0.697
inside at all 500 grid times: 0.535
```

At the single time $t = 1$ the fraction is about 95%, as it must be. But only about seven paths in ten are inside at each of 50 equally spaced times, and only a little over half at all 500. A band that holds at each time does not hold for a path: every look is another chance to find the path outside. Watched at every instant, no path stays inside, since near $t = 0$ a Brownian path crosses $\pm 2\sqrt{t}$ again and again (the law of the iterated logarithm, which we do not prove). A statement about the law at one time is much weaker than a statement about whole paths, and the two return as the weak and the strong error of a simulation (Section 38.4.4.2).

38.4.2 Itô Calculus

38.4.2.1 Quadratic Variation

Ordinary calculus stops its Taylor expansions at first order because the square of a small change is negligible. For Brownian motion this fails, and the result of this subsection says by how much. It is the one on which the rest of the section rests.

Take a partition $0 = t_0 < t_1 < \cdots < t_n = t$ with increments $\Delta t_i = t_{i+1} - t_i$ and mesh $h_{\max} = \max_i \Delta t_i$, and add up the squared increments of a path along it. For a differentiable path x with $|x'| \le M$, the mean value theorem gives $|\Delta x_i| \le M\,\Delta t_i$, so

$$\sum_i (\Delta x_i)^2 \;\le\; M^2 \sum_i \Delta t_i^2 \;\le\; M^2\, h_{\max}\, t \;\longrightarrow\; 0$$

as the mesh shrinks: a smooth increment squared is of order Δt^2. A Brownian increment squared is of order Δt, since $\mathbb{E}[(\Delta W_i)^2] = \Delta t_i$, and these add up to t however fine the mesh (Figure 38.13). The sum is random, but its fluctuations vanish in the limit.

Proposition (quadratic variation of Brownian motion). *As the mesh $h_{\max} \to 0$,*

$$Q_n \;=\; \sum_{i=0}^{n-1} \left(W_{t_{i+1}} - W_{t_i}\right)^2 \;\longrightarrow\; t \quad \text{in mean square:} \quad \mathbb{E}\big[(Q_n - t)^2\big] \to 0. \tag{38.27}$$

Proof. Each increment is $\mathcal{N}(0, \Delta t_i)$, so $\mathbb{E}[(\Delta W_i)^2] = \Delta t_i$ and $\mathbb{E}[Q_n] = \sum_i \Delta t_i = t$ for every partition. For the spread, a Gaussian has fourth moment $3\,\Delta t_i^2$, so $\operatorname{Var}\big((\Delta W_i)^2\big) = 3\Delta t_i^2 - \Delta t_i^2 = 2\Delta t_i^2$; the increments are independent, so their variances add:

$$\mathbb{E}\big[(Q_n - t)^2\big] = \operatorname{Var}(Q_n) = \sum_i 2\,\Delta t_i^2 \;\le\; 2\,h_{\max} \sum_i \Delta t_i = 2\,h_{\max}\,t \;\longrightarrow\; 0,$$

and mean-square convergence follows. ■

The statement deserves a second look: every increment is random, and in the limit the sum of their squares is not. On a uniform grid with n intervals of $[0, 1]$ the proof gives the sum a standard deviation of $\sqrt{2/n}$, which is 0.35 for $n = 16$ and 0.003 for $n = 2^{18}$.

The proposition is usually written as the **Itô multiplication table**, the working rules for products of differentials:

$$(dW)^2 = dt, \qquad dW\,dt = 0, \qquad (dt)^2 = 0. \tag{38.28}$$

The first rule is Equation 38.27 in shorthand, and it is a statement about sums. A single squared increment is $\Delta t\,\xi^2$, which is rarely close to Δt: about one time in four it is below a tenth of Δt, and about one time in six it is above twice Δt. Only the sum over many steps settles, as the proof showed. (For the coin-flip walk every squared step equals Δt, and nothing needs to settle.) The other two rules record that $dW\,dt$, of size $\Delta t^{3/2}$, and $(dt)^2$ vanish faster than Δt and contribute nothing in the limit.

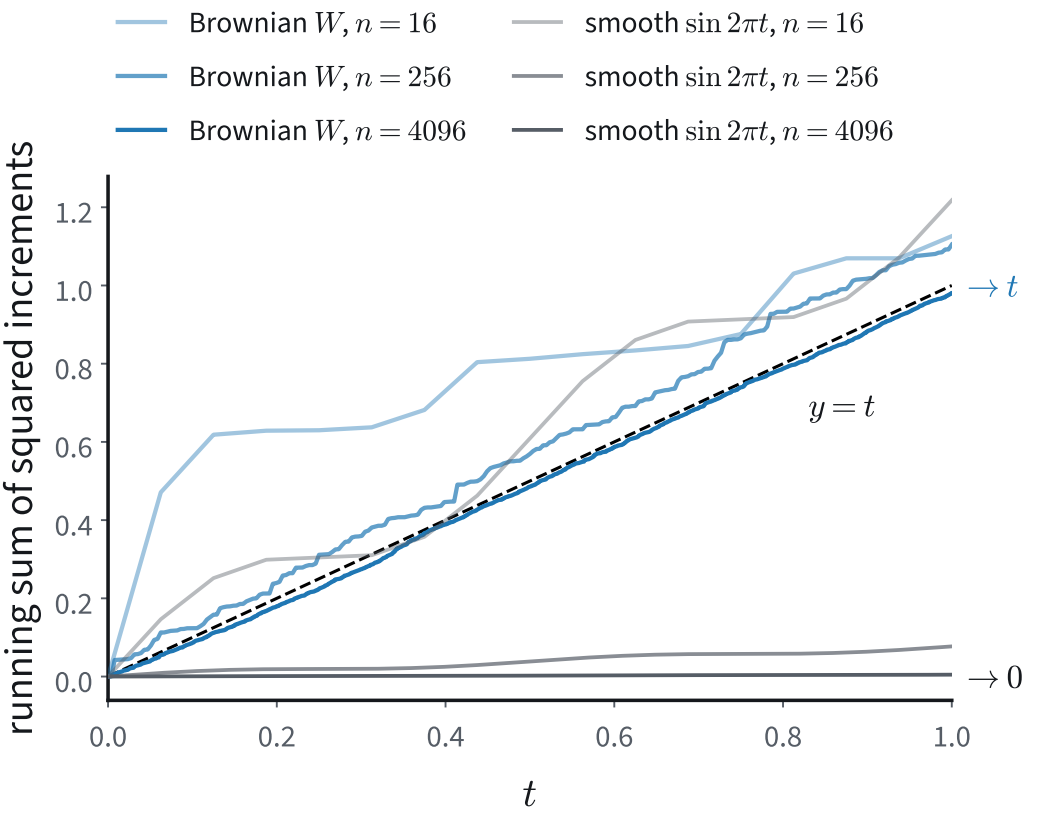


Figure 38.13: Running sums of squared increments along one Brownian path for meshes of $n = 16$, 256, and 4096 intervals (darker shades are finer meshes). The sums approach $y = t$, whereas the sums for the smooth path $\sin 2\pi t$ (gray) shrink like $2\pi^2/n$.

At fine meshes the fluctuations are so small that a single path shows the limit. We sample a path on 2^{18} intervals of $[0, 1]$, view it on coarser subgrids, and print its sum of squares next to the predicted standard deviation. For comparison we print the same sum for the smooth path $x(t) = \sin 2\pi t$ next to $2\pi^2/n$, the mesh $1/n$ times $\int_0^1 x'(t)^2\,dt = 2\pi^2$.

```
n_fine = 2**18
dW_qv = np.sqrt(1.0 / n_fine) * rng.standard_normal(n_fine)
W_qv = np.concatenate([[0.0], np.cumsum(dW_qv)])
print(f'{"n":>7} {"QV of W":>8} {"sqrt(2/n)":>10} {"QV of sin":>10}'
      f' {"2 pi^2/n":>9}')
for k in range(4, 19, 2):
    n = 2**k
    W_n = W_qv[::n_fine // n]                # the same path on a coarser grid
    s_n = np.sin(2 * np.pi * np.linspace(0, 1, n + 1))
    print(f'{n:7d} {np.sum(np.diff(W_n)**2):8.4f} {np.sqrt(2 / n):10.4f} '
          f'{np.sum(np.diff(s_n)**2):10.6f} {2 * np.pi**2 / n:9.6f}')
```

```
      n  QV of W  sqrt(2/n)  QV of sin  2 pi^2/n
     16   1.1551     0.3536   1.217927  1.233701
     64   0.9103     0.1768   0.308177  0.308425
    256   0.8957     0.0884   0.077102  0.077106
   1024   0.9560     0.0442   0.019277  0.019277
   4096   1.0044     0.0221   0.004819  0.004819
  16384   0.9893     0.0110   0.001205  0.001205
  65536   1.0007     0.0055   0.000301  0.000301
 262144   1.0017     0.0028   0.000075  0.000075
```

The Brownian column scatters around 1 with the standard deviation $\sqrt{2/n}$ that the proof predicts. On 16 intervals it typically lands between about 0.5 and 1.6, and at $n = 2^{18}$ it is within about 0.01 of 1. The smooth column is still 1.22 at $n = 16$, but it shrinks like $2\pi^2/n$ and agrees with that prediction to three digits from $n = 256$ on.

38.4.2.2 The Itô Integral

The rough path forces us to define integrals against dW directly. We start with the simplest one, $\int_0^t W\,dW$, and find that the usual Riemann-sum definition does not single out an answer.

For a smooth integrator, Riemann sums with the integrand evaluated at the left endpoint, the right endpoint, or the midpoint of each interval all converge to the same limit. Here they do not. With W itself as the integrand, the right-endpoint sum exceeds the left-endpoint sum by

$$\sum_i W_{t_{i+1}}\,\Delta W_i - \sum_i W_{t_i}\,\Delta W_i = \sum_i \left(W_{t_{i+1}} - W_{t_i}\right)\Delta W_i = \sum_i (\Delta W_i)^2 \;\longrightarrow\; t,$$

by the proposition just proved. The two discretizations of "the same" integral disagree by a finite, deterministic amount. Let us look at one path with 1000 steps.

```
n_lr = 1000
dt_lr = 1.0 / n_lr
t_lr = np.linspace(0, 1, n_lr + 1)
dW_lr = np.sqrt(dt_lr) * rng.standard_normal(n_lr)
W_lr = np.concatenate([[0.0], np.cumsum(dW_lr)])
left = np.sum(W_lr[:-1] * dW_lr)              # integrand at the left endpoint
right = np.sum(W_lr[1:] * dW_lr)              # integrand at the right endpoint
print(f'left sum {left:.4f}   right sum {right:.4f}')
print(f'right - left {right - left:.4f}   sum of (dW)^2 {np.sum(dW_lr**2):.4f}')
```

```
left sum -0.3614   right sum 0.6124
right - left 0.9738   sum of (dW)^2 0.9738
```

The left sum is a random number that changes from run to run. The right sum exceeds it by the sum of squared increments in every printed digit, as the identity above says it must. What the run adds is the size of the gap. It usually lies within about 0.1 of 1 (its standard deviation is $\sqrt{2/1000} \approx 0.045$), and a finer mesh would not close it.

Both sums have limits, and we can find them with what we have. Their total telescopes, because $\left(W_{t_{i+1}} + W_{t_i}\right) \Delta W_i = W_{t_{i+1}}^2 - W_{t_i}^2$ adds up to W_t^2 on every partition, and their difference tends to t. Hence the left sum tends to $\frac{1}{2}\left(W_t^2 - t\right)$ and the right sum to $\frac{1}{2}\left(W_t^2 + t\right)$, in mean square. Ordinary calculus would answer $\frac{1}{2}W_t^2$, which is neither limit but their average. We must choose an evaluation point, and the choice changes the answer. Which of the two sums should be the integral?

Itô's choice is the left endpoint, so that the integrand never sees the coming increment. A fair game explains why. Think of ΔW_i as the outcome of the i-th fair bet and of H_i as the stake placed on it. A gambler must choose the stake from what is known at time t_i, before the bet is settled, and the total winnings are $\sum_i H_i \, \Delta W_i$. Because the increment ΔW_i is independent of everything known at t_i, including H_i, each bet has expected winnings $\mathbb{E}[H_i \, \Delta W_i] = \mathbb{E}[H_i] \, \mathbb{E}[\Delta W_i] = 0$, so no betting strategy has positive expected winnings. A stake that could peek at the outcome would make money on average. The right-endpoint sum places such a stake, and its mean is indeed t, not zero.

The definition writes this rule down. An integrand is *predictable* if its value on each interval is decided before the increment over that interval is drawn. A **simple predictable process** H is constant on each interval, $H_s = H_i$ for $s \in (t_i, t_{i+1}]$, where H_i is determined by the path up to time t_i and $\mathbb{E}[H_i^2] < \infty$. Its **Itô integral** is the sum of the winnings,

$$\int_0^t H_s \, dW_s \;=\; \sum_i H_i \left(W_{t_{i+1}} - W_{t_i}\right). \tag{38.29}$$

The fair-game argument shows that it has mean zero, $\mathbb{E}\left[\int_0^t H \, dW\right] = 0$. Its variance is computable as well.

Proposition (Itô isometry). *For a simple predictable H,*

$$\mathbb{E}\left[\left(\int_0^t H_s \, dW_s\right)^2\right] \;=\; \int_0^t \mathbb{E}\big[H_s^2\big] \, ds. \tag{38.30}$$

Proof. Expand the square of the sum in Equation 38.29. The step that needs care is the cross terms. A cross term with $i < j$ contains the factor ΔW_j, which is independent of everything known at time t_j, including the product $H_i\,\Delta W_i\,H_j$. So the expectation factorizes: $\mathbb{E}\big[H_i\,\Delta W_i\,H_j\,\Delta W_j\big] = \mathbb{E}\big[H_i\,\Delta W_i\,H_j\big]\,\mathbb{E}[\Delta W_j] = 0$. A diagonal term factorizes the same way into $\mathbb{E}\big[H_i^2\big]\,\mathbb{E}\big[(\Delta W_i)^2\big] = \mathbb{E}\big[H_i^2\big]\,\Delta t_i$. Summing the diagonal gives $\sum_i \mathbb{E}[H_i^2]\,\Delta t_i = \int_0^t \mathbb{E}[H_s^2]\,ds$. ■

The isometry also carries the definition beyond step processes, to integrands such as $H = W$ itself. It says that $H \mapsto \int H\,dW$ preserves the norm $\|H\|^2 = \int_0^t \mathbb{E}[H_s^2]\,ds$. So if simple predictable processes converge to H in this norm, their integrals converge in mean square, and the limit is taken as the definition of $\int_0^t H\,dW$. The zero mean and the isometry survive the limit and hold for every predictable H of finite norm, and the left-endpoint sums of a continuous integrand of this kind converge to its integral. We do not prove these statements; see Øksendal (2003), chapter 3.

The integral we started from now has a value, the limit of its left-endpoint sums:

$$\int_0^t W_s\,dW_s = \tfrac{1}{2}\left(W_t^2 - t\right). \tag{38.31}$$

This is the answer of ordinary calculus minus $t/2$. The $-t/2$ gives it the mean zero that a fair game must have, since $\mathbb{E}[W_t^2] = t$. The isometry can be checked on it too. It turns the variance of a noise integral into an ordinary integral of a second moment, here $\int_0^1 \mathbb{E}[W_s^2]\,ds = \int_0^1 s\,ds = \frac{1}{2}$ at $t = 1$, and the closed form agrees: W_1^2 is a squared standard normal, with variance 2, so $\mathrm{Var}(\frac{1}{2}(W_1^2 - 1)) = \frac{1}{4}\cdot 2 = \frac{1}{2}$.

The other evaluation points define other integrals. The midpoint choice, or the average of the left and right sums, defines the *Stratonovich integral* (Stratonovich 1966). It obeys the ordinary chain rule, as the average $\frac{1}{2}W_t^2$ above suggests, but gives up the zero mean, because its integrand sees half of the coming increment. This book uses Itô's convention, for two reasons: the zero mean makes expectations easy, and the simplest simulation step, which evaluates the coefficients at the start of the step, is an Itô sum (Exercise 6).

38.4.2.3 Drift, Diffusion, and Solutions

With the integral in hand, the forward-process template Equation 38.25 acquires a meaning. A **stochastic differential equation** is written

$$dX = f(X,t)\,dt + g(X,t)\,dW, \tag{38.32}$$

and the differential form is shorthand for the integral equation

$$X_t = X_0 + \int_0^t f(X_s,s)\,ds + \int_0^t g(X_s,s)\,dW_s,$$

an ordinary integral plus an Itô integral. Over a short step, with the coefficients frozen at its start, $\mathbb{E}[dX] = f\,dt$ and $\mathrm{Var}(dX) = g^2\,dt$. So the **drift** f is the conditional mean velocity,

and the **diffusion** coefficient g sets the rate g^2 at which variance is pumped in. With $g \equiv 0$ the equation is the ODE of Section 38.1, and with $f \equiv 0$ and a constant g it is the noise-only process $X_t = X_0 + g\,W_t$ of Figure 38.11.

What is a solution? For an ODE it is one curve. Here it is one curve for each Brownian path. If f and g are Lipschitz in x and grow at most linearly, $|f(x,t)|+|g(x,t)| \le C(1+|x|)$, and X_0 is independent of W with $\mathbb{E}[X_0^2] < \infty$, then the start and the Brownian path determine the solution path uniquely (Øksendal (2003), Theorem 5.2.1). The proof is a stochastic version of the Picard iteration of Section 38.1.1.3, with the isometry in place of the integral bound. All the randomness therefore sits in W. Running the equation many times from the same start with fresh noise gives a cloud of paths like those of Figure 38.11, and what a generative model cares about is the law of the cloud at each time.

One distinction will matter when we simulate. Noise is **additive** if its amplitude depends on time only, $g = g(t)$, as in the template Equation 38.25 and in every diffusion-model SDE of this chapter, the running example among them. It is **multiplicative** if the amplitude depends on the state, $g = g(X,t)$, as for a stock price whose fluctuations are proportional to its level. The difference looks cosmetic, but it decides how accurately a simulated path follows the true one (Section 38.4.4.2).

38.4.2.4 Itô's Lemma

To compute with an SDE we need the chain rule: if X solves an SDE, what equation does a function $\phi(X_t, t)$ solve? The ordinary chain rule would give $d(W^2) = 2W\,dW$, and we know that this is wrong, since $\int_0^t 2W\,dW = W_t^2 - t$ by Equation 38.31. Something is missing.

One step of a walk shows where the missing term comes from, and what it is for other functions. Take $\phi(x) = x^2$ and one step $\pm\sqrt{\Delta t}$ from x, each sign with probability $\frac{1}{2}$. The average of ϕ after the step is

$$\tfrac{1}{2}\left[(x+\sqrt{\Delta t})^2 + (x-\sqrt{\Delta t})^2\right] = x^2 + \Delta t = \phi(x) + \tfrac{1}{2}\phi''(x)\,\Delta t,$$

since $\phi'' = 2$. The rise comes from the square of the step, which for a coin flip is Δt every time, and which the ordinary chain rule throws away.

With $x = 0.7$ and $\Delta t = 0.25$ the step lands at 0.2 or 1.2, where ϕ is 0.04 or 1.44. The average 0.74 lies above $\phi(0.7) = 0.49$ by $0.25 = \Delta t$ (Figure 38.14). A convex function gains from symmetric noise, and because a squared Brownian increment is of order dt, the gain per unit time, $\frac{1}{2}\phi''$, does not vanish in the limit.

The general formula comes from the same computation. Let X solve $dX = f\,dt + g\,dW$ and let $\phi(x,t)$ be smooth. A small change of ϕ, expanded to second order because first order is no longer enough, is

$$d\phi = \phi_t\,dt + \phi_x\,dX + \tfrac{1}{2}\phi_{xx}\,(dX)^2 + \cdots.$$

Substitute $dX = f\,dt + g\,dW$ and multiply out $(dX)^2$ with the table Equation 38.28:

$$(dX)^2 = f^2(dt)^2 + 2fg\,dt\,dW + g^2(dW)^2 = g^2\,dt.$$

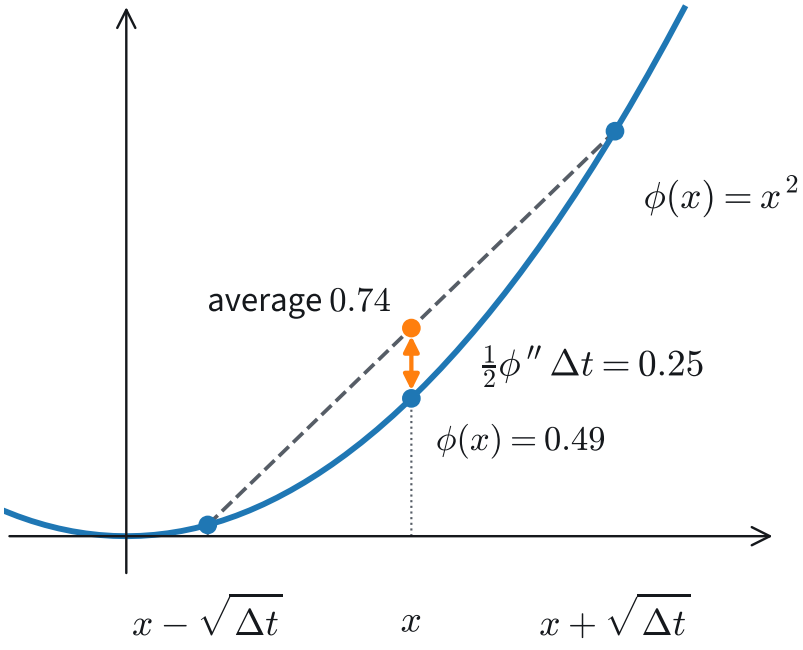


Figure 38.14: The Itô correction as Jensen's gap. One step of $\pm\sqrt{\Delta t}$ from x moves a convex ϕ to an average that lies above $\phi(x)$ by $\frac{1}{2}\phi''(x)\,\Delta t$; here $\phi = x^2$, $x = 0.7$, $\Delta t = 0.25$, and the gap is 0.25.

The drift and cross terms vanish, and the noise term survives at first order. Collecting terms gives the chain rule of stochastic calculus. What we have given is a heuristic derivation, so we state the result as a theorem without proof.

Theorem (Itô's lemma). *Let X solve $dX = f\,dt + g\,dW$ under the conditions of the previous subsection, and let $\phi \in C^{1,2}$, once continuously differentiable in time and twice in the state. Then $Y_t = \phi(X_t, t)$ satisfies*

$$d\phi \;=\; \left(\phi_t + f\,\phi_x + \tfrac{1}{2} g^2\,\phi_{xx}\right) dt \;+\; g\,\phi_x\,dW. \tag{38.33}$$

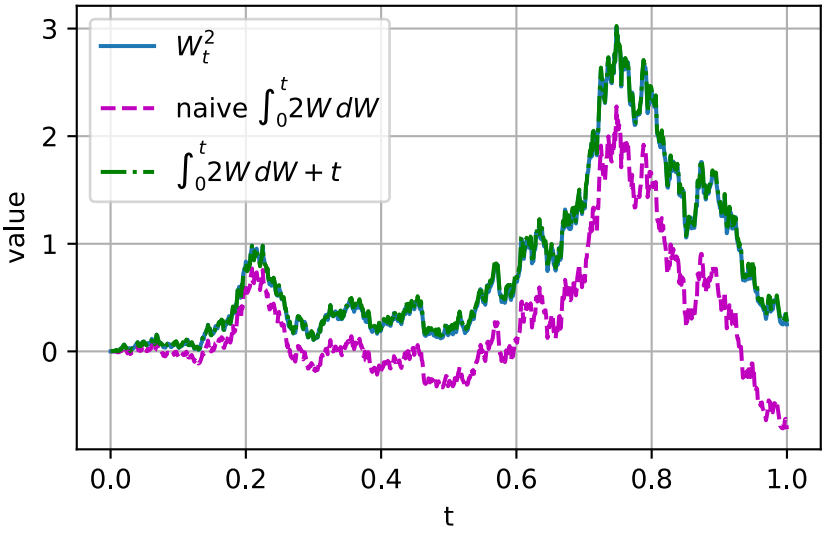
This is the working formula of the subject: the ordinary chain rule, plus half the second derivative times the rate at which variance arrives. The new term $\frac{1}{2}g^2\phi_{xx}\,dt$ is called the **Itô correction**. The proof has to control the remainder terms of the Taylor expansion in mean square (Øksendal (2003), chapter 4), and it needs both derivatives in the state: the lemma does not apply as stated to a function with a kink, such as $\phi(x) = |x|$.

Example 1: W^2. Take $X = W$ (so $f = 0$, $g = 1$) and $\phi(x) = x^2$. The lemma gives $d(W^2) = 2W\,dW + dt$, with the dt that the ordinary chain rule dropped. In integrated form this is $W_t^2 = \int_0^t 2W_s\,dW_s + t$, which is Equation 38.31 again. One line of the lemma has replaced the telescoping sums.

The correction can be watched as it accumulates. On any grid the gap between W_t^2 and the left-endpoint sum for $\int_0^t 2W\,dW$ telescopes, $W_t^2 - \sum_{t_i<t} 2W_{t_i}\Delta W_i = \sum_{t_i<t}(\Delta W_i)^2$, so it is the running quadratic variation and should follow the line t. We reuse the path of the left and right sums.

```
ito = np.concatenate([[0.0], np.cumsum(2 * W_lr[:-1] * dW_lr)])  # int 2W dW
gap = W_lr**2 - ito
print(f'final gap W_1^2 - int 2W dW = {gap[-1]:.4f}   (Ito correction: t = 1)')
print(f'max |gap_t - t| along the path = {np.abs(gap - t_lr).max():.4f}')
d2l.plot(t_lr, [W_lr**2, ito, ito + t_lr], 't', 'value',
         legend=['$W_t^2$', 'naive $\\int_0^t 2W\\,dW$',
                 '$\\int_0^t 2W\\,dW + t$'], figsize=(5, 3))
```

```
final gap W_1^2 - int 2W dW = 0.9738   (Ito correction: t = 1)
max |gap_t - t| along the path = 0.0364
```



The curve labeled naive is the Itô integral $\int_0^t 2W\,dW$, which the ordinary chain rule would take for W_t^2. It sags below W_t^2, and adding t puts it back on top. Along the whole path the gap usually stays within about 0.1 of t, since by the proof of Equation 38.27 its standard deviation at time t is $\sqrt{2t\,\Delta t} \le 0.045$ on this grid. At $t = 1$ the gap is the sum of squared increments that the earlier cell printed.

Example 2: geometric Brownian motion. The SDE $dX = \mu X\,dt + \nu X\,dW$ with $X_0 > 0$ models a quantity that grows at mean rate μ with noise proportional to its size. It is Samuelson's model of a stock price (Samuelson 1965), on which the option-pricing formula of Black and Scholes rests (Black and Scholes 1973). Its noise is multiplicative. Apply the lemma to $\phi(x) = \log x$, with $\phi' = 1/x$, $\phi'' = -1/x^2$, and $g^2 = \nu^2 x^2$. The logarithm is smooth only for $x > 0$, so assume for the moment that X stays positive:

$$d\log X = \left(\mu - \tfrac{1}{2}\nu^2\right)dt + \nu\,dW, \qquad \text{so} \qquad X_t = X_0 \exp\!\left(\left(\mu - \tfrac{1}{2}\nu^2\right)t + \nu W_t\right). \tag{38.34}$$

The formula we obtain is positive, and the lemma applied to it as a function of W_t and t confirms that it solves the equation, which removes the assumption. The $-\frac{1}{2}\nu^2$ is the correction with $\phi'' < 0$: a concave function loses from symmetric noise. Noise therefore lowers the typical growth rate $\mu - \frac{1}{2}\nu^2$ below the mean growth rate μ, and Exercise 3 shows how far the mean and the median of X_t drift apart. We will measure simulation errors against this exact solution.

In several dimensions the pattern is the same, and we take the result as given (Øksendal 2003). For $d\mathbf{X} = \mathbf{f}\,dt + \mathbf{G}\,d\mathbf{W}$ with $\mathbf{X} \in \mathbb{R}^d$, a matrix $\mathbf{G}$, and a Brownian motion $\mathbf{W}$ whose coordinates are independent scalar Brownian motions, the multiplication table reads $dW_i\,dW_j = \delta_{ij}\,dt$, and a twice continuously differentiable ϕ satisfies

$$d\phi(\mathbf{X}) \;=\; \nabla\phi \cdot d\mathbf{X} \;+\; \tfrac{1}{2}\sum_{i,j} \left(\mathbf{G}\mathbf{G}^{\top}\right)_{ij} \frac{\partial^2 \phi}{\partial x_i\,\partial x_j}\,dt. \tag{38.35}$$

The correction weights each second derivative by the covariance rate $\mathbf{G}\mathbf{G}^{\top}$ of the noise. The diffusion-model SDEs of this chapter treat every coordinate alike, $\mathbf{G} = g(t)\,\mathbf{I}$, and for them it reduces to $\frac{1}{2}g(t)^2 \sum_i \partial^2\phi/\partial x_i^2 \; dt$, a multiple of the Laplacian of ϕ. The next section needs the lemma in this form.

38.4.3 The Ornstein–Uhlenbeck Process

The running example now gets its name and its solution. The Ornstein–Uhlenbeck (OU) process (Uhlenbeck and Ornstein 1930) is

$$dX = -\theta X\,dt + g\,dW, \qquad \theta > 0, \tag{38.36}$$

the decay ODE $\dot{x} = -\theta x$ of Section 38.1.2 with noise of constant amplitude g added. Its authors wrote it for the velocity of a particle that is kicked by molecules and slowed by friction. The drift pulls excursions back toward zero at rate θ, and the noise keeps producing new ones (Figure 38.15). The recurrence of the opening is this process with $\theta = 1$ and $g = \sqrt{2}$, advanced on a grid with step $h = 0.01$ by freezing the drift over each step: one step multiplies by $1 - \theta h = 0.99$ and adds noise of standard deviation $g\sqrt{h} = \sqrt{0.02}$.

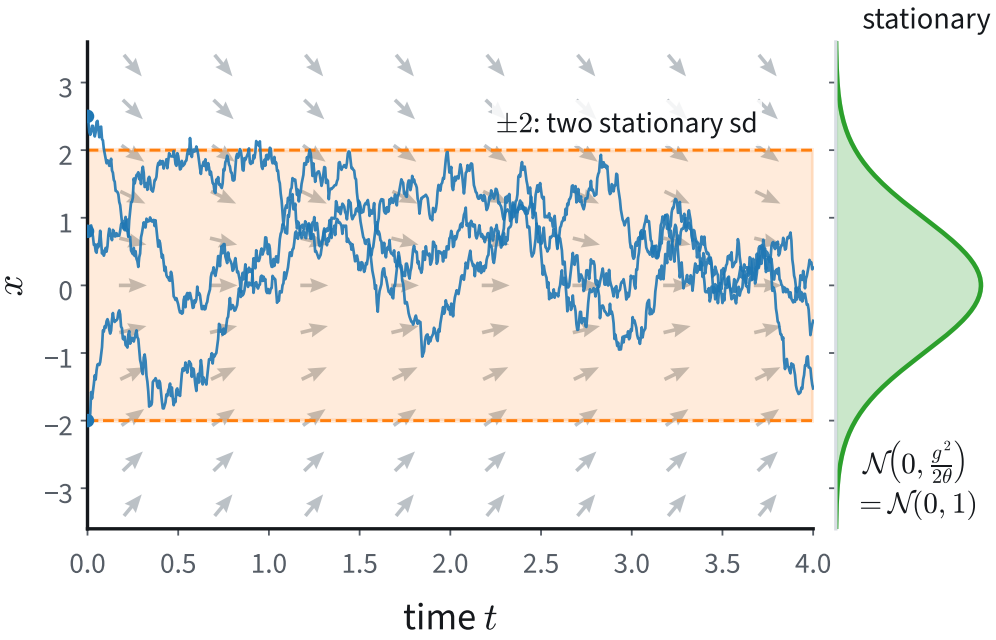


Figure 38.15: Mean reversion. The drift $-\theta x$ (arrows) pulls every path toward 0 at rate $\theta = 1$ while noise of amplitude $g = \sqrt{2}$ keeps it moving; paths from three starts forget them within a few multiples of $1/\theta$ and then fluctuate inside the band of two standard deviations of the limiting law, $\pm 2\sqrt{g^2/2\theta} = \pm 2$ (labeled stationary). Right: its density.

38.4.3.1 The Transition Kernel

The OU process is one of the few SDEs whose law is known in closed form at every time, and this is why diffusion models are built on it. The law of X_t given the start $X_0 = x_0$, read as a function of the start, is called the *transition kernel* (no relation to the kernels of a convolution). For the OU process it is Gaussian. Of all the results of this section, this is the one the rest of the book uses most.

Proposition (OU transition kernel). *For the OU process Equation 38.36,*

$$X_t \mid X_0 = x_0 \;\sim\; \mathcal{N}\!\left(x_0\, e^{-\theta t},\; \frac{g^2}{2\theta}\left(1 - e^{-2\theta t}\right)\right). \tag{38.37}$$

Proof. The integrating factor $e^{\theta t}$ solved the decay equation with an input in Section 38.1.2.3 (by the variation of constants formula), and it works when the input is noise. Apply Itô's lemma Equation 38.33 to $\phi(x,t) = e^{\theta t}x$, for which $\phi_t = \theta e^{\theta t}x$, $\phi_x = e^{\theta t}$, and $\phi_{xx} = 0$, so there is no correction:

$$d(e^{\theta t}X_t) = (\theta e^{\theta t}X_t - \theta e^{\theta t}X_t)\,dt + e^{\theta t}g\,dW_t = g\,e^{\theta t}\,dW_t.$$

The drift cancels, and integrating from 0 to t and multiplying by $e^{-\theta t}$ gives

$$X_t = X_0\, e^{-\theta t} + g\int_0^t e^{-\theta(t-s)}\,dW_s. \tag{38.38}$$

The remaining integral has a deterministic integrand, so it is a limit of weighted sums of independent Gaussian increments and is Gaussian with mean zero (we grant that mean-square limits of Gaussians are Gaussian). The Itô isometry Equation 38.30 gives its variance:

$$\operatorname{Var}(X_t \mid X_0) = g^2\int_0^t e^{-2\theta(t-s)}\,ds = \frac{g^2}{2\theta}\left(1 - e^{-2\theta t}\right). \qquad \blacksquare$$

The solution Equation 38.38 reads as a fading memory. The start decays at rate θ, and a noise kick from time s survives with weight $e^{-\theta(t-s)}$: recent noise counts, old noise is forgotten. In the recurrence a kick from k steps ago survives with weight $0.99^k \approx e^{-0.01k}$. Every later section, and Section 17.2 in the main text, writes the kernel with two coefficients, the mean factor and the standard deviation,

$$\alpha_t = e^{-\theta t}, \qquad \sigma_t^2 = \frac{g^2}{2\theta}\left(1 - e^{-2\theta t}\right), \qquad \text{so that} \qquad X_t \mid X_0 = x_0 \sim \mathcal{N}(\alpha_t x_0,\, \sigma_t^2). \tag{38.39}$$

The data enter only through the mean $\alpha_t x_0$, and the noise only through the variance σ_t^2. For the running example ($\theta = 1$, $g = \sqrt{2}$, $\sigma_t^2 = 1 - e^{-2t}$) started at $x_0 = 2$, the kernel is $\mathcal{N}(1.213,\, 0.795^2)$ at $t = 0.5$, $\mathcal{N}(0.736,\, 0.930^2)$ at $t = 1$, and $\mathcal{N}(0.037,\, 1.000^2)$ at $t = 4$. Because the kernel is Gaussian at every time and for every start, everything about the noised state given its start is known in closed form, including the gradient of its log-density, $-(x - \alpha_t x_0)/\sigma_t^2$. Generation will turn out to depend on this gradient.

The recurrence is only a discretization of the process, so its cloud should follow the kernel closely but need not follow it exactly. We run it as a plain loop on $10{,}000$ copies of $x_0 = 2$, save the cloud at five times, and print its mean and standard deviation next to $2\alpha_t$ and σ_t, together with the fraction of the cloud inside $2\alpha_t \pm 2\sigma_t$.

```
theta_ou, g_ou, h_ou = 1.0, np.sqrt(2.0), 0.01
X_ou = np.full(10000, 2.0)                    # 10,000 copies of x_0 = 2
snap_steps = {round(t / h_ou): t for t in (0.5, 1.0, 1.5, 2.0, 4.0)}
X_ou_snap = {}
for k in range(1, 401):
    xi = rng.standard_normal(X_ou.size)
    X_ou = (1 - theta_ou * h_ou) * X_ou + g_ou * np.sqrt(h_ou) * xi
    if k in snap_steps:
        X_ou_snap[snap_steps[k]] = X_ou.copy()
print(f'{"t":>4} {"mean":>6} {"2alpha_t":>8} {"std":>6} {"sigma_t":>7}'
      f' {"inside":>7}')
for t, X in X_ou_snap.items():
    alpha_t = np.exp(-theta_ou * t)
    sigma_t = np.sqrt(g_ou**2 / (2 * theta_ou) * (1 - alpha_t**2))
    inside = np.mean(np.abs(X - 2 * alpha_t) < 2 * sigma_t)
    print(f'{t:4.1f} {X.mean():6.3f} {2 * alpha_t:8.3f} {X.std():6.3f} '
          f'{sigma_t:7.3f} {inside:7.3f}')
```

```
   t   mean 2alpha_t    std sigma_t  inside
 0.5  1.209    1.213  0.803   0.795   0.951
 1.0  0.733    0.736  0.943   0.930   0.955
 1.5  0.433    0.446  0.968   0.975   0.957
 2.0  0.259    0.271  0.992   0.991   0.953
 4.0  0.042    0.037  1.012   1.000   0.952
```

Means and standard deviations agree with the kernel within two or three standard errors (about 0.01 for a mean and 0.007 for a standard deviation from $10{,}000$ samples). About 95% of the cloud lies inside the band at every time, as for any Gaussian. The cloud that began as the single point 2 widens quickly and then stops widening. At this sample size the discretization cannot be seen.

38.4.3.2 The Stationary Distribution

A forward process must forget its start, and the kernel Equation 38.37 shows that the OU process does. As $t \to \infty$ the mean $x_0 e^{-\theta t}$ decays to zero and the variance approaches $g^2/(2\theta)$, so

$$X_t \mid X_0 = x_0 \;\longrightarrow\; \mathcal{N}\left(0,\, \frac{g^2}{2\theta}\right) \quad \text{in distribution, for every } x_0.$$

This is the endpoint, the same for all data, that Section 38.4.1.1 asked for. The limit is more than an endpoint. A process started in it stays in it, and a law with this property is called *stationary*. To see it, let $X_0 \sim \mathcal{N}(0, g^2/2\theta)$ be independent of W. By Equation 38.38, X_t is then the sum of $X_0 e^{-\theta t}$ and an independent Gaussian of variance σ_t^2, hence Gaussian with mean zero and variance

$$e^{-2\theta t}\,\frac{g^2}{2\theta}+\frac{g^2}{2\theta}\,(1-e^{-2\theta t})=\frac{g^2}{2\theta}.$$

The decay of the start removes as much variance as the noise adds. How long does the forgetting take? Its time scale is $1/\theta$, the *relaxation time*, and after a few multiples of it $e^{-\theta t}$ is negligible. For the running example $g^2/2\theta = 1$ and $1/\theta = 1$, so the stationary law is $\mathcal{N}(0,1)$, and by $t = 4$ the mean had fallen from 2 to 0.037.

The recurrence has a stationary law of its own, and it is slightly different. One step maps a variance v to $(1-\theta h)^2 v + g^2 h$. From $v = 1$ a step of the running example gives $0.9801 + 0.02 = 1.0001$, slightly more than 1, and the fixed point of the map is

$$v_\infty = \frac{g^2 h}{1-(1-\theta h)^2} = \frac{0.02}{0.0199} = 1.005 \qquad (h = 0.01).$$

The recurrence therefore settles half a percent above the exact variance 1. No cloud of $10{,}000$ points can show half a percent, but the pencil can. This is the first sign of a discretization error, and Section 38.4.4.2 shows that it shrinks in proportion to h.

38.4.3.3 The Variance-Preserving Normalization

The stationary variance $g^2/2\theta$ is a free parameter, and diffusion models fix it to 1 by choosing

$$g^2 = 2\theta \qquad \Longrightarrow \qquad X_t \mid X_0 = x_0 \;\longrightarrow\; \mathcal{N}(0,1).$$

The running example makes this choice. With it $\sigma_t^2 = 1-\alpha_t^2$, so one coefficient describes the kernel.

So far the start was a single number x_0. Let it now be a draw X_0 from the data. In distribution, the solution Equation 38.38 then reads $X_t = \alpha_t X_0 + \sqrt{1-\alpha_t^2}\,\epsilon$ with a standard Gaussian ϵ independent of X_0. The law of this X_t, which is the kernel averaged over the data, is called the *marginal* at time t. Most papers write it with another letter. They follow the denoising diffusion probabilistic model (DDPM) of Ho et al. (2020), the discrete-time diffusion model of Section 17.2, which writes $\bar\alpha_t$ for our $\alpha_t^2 = e^{-2\theta t}$. The main text does the same, and so do we:

$$X_t \;=\; \sqrt{\bar\alpha_t}\;X_0 \;+\; \sqrt{1-\bar\alpha_t}\;\epsilon, \qquad \epsilon \sim \mathcal{N}(0,1) \text{ independent of } X_0. \tag{38.40}$$

The marginal is a weighted mix of data and fresh noise, with weights whose squares add to one. Suppose the data have zero mean and unit variance. Taking variances of the two independent terms gives

$$\mathrm{Var}(X_t) = \bar\alpha_t \mathrm{Var}(X_0) + (1-\bar\alpha_t) = \bar\alpha_t + (1-\bar\alpha_t) = 1 \qquad \text{for all } t.$$

This is what **variance-preserving** (VP) means. For unit-variance data the marginal variance equals 1 at every time, not only in the limit. The process moves variance from the data to the noise, a fraction $\bar\alpha_t$ still carried by the data and $1-\bar\alpha_t$ already replaced by noise, without

changing the total. The name is accurate for unit-variance data only. For data of variance s^2 the marginal variance drifts from s^2 to 1 (Exercise 5), so the scale of the data matters. Images scaled to $[-1, 1]$, as in Section 17.1, have variance below one, while latent diffusion models rescale their latents to about unit variance (Rombach et al. 2022).

The variance is preserved; the shape of the distribution is not. Take data that are ± 1 with probability $\frac{1}{2}$ each, the outcome of a coin flip. Their fourth moment is 1, against 3 for a standard Gaussian, and along the process it climbs as $\mathbb{E}[X_t^4] = 3 - 2\bar{\alpha}_t^2$, which is 2.264 at $t = 0.25$ and 2.963 at $t = 1$ for the running example (same exercise). The coin turns into a bell curve, and its variance never moves.

Diffusion models usually let the rate vary with time. The **VP-SDE** of score-based diffusion (Y. Song, Sohl-Dickstein, et al. 2021),

$$dX = -\tfrac{1}{2}\beta(t)\, X\, dt + \sqrt{\beta(t)}\, dW, \tag{38.41}$$

is an OU process with time-dependent rate $\theta(t) = \beta(t)/2$ and noise amplitude $g(t)^2 = \beta(t)$; the running example is the constant schedule $\beta = 2$. It keeps $g^2 = 2\theta$ at every instant, so the variance stays at 1 along the whole schedule, and its marginal is Equation 38.40 with $\bar{\alpha}_t = e^{-\int_0^t \beta(s)\, ds}$ (Exercise 7 proves that its transitions are exact Gaussians). This is the variance-preserving path of Section 17.2, the forward process that a diffusion model learns to reverse.

38.4.4 Simulation

38.4.4.1 The Euler–Maruyama Scheme

Most SDEs have no closed-form solution, so we need to simulate them. Our recurrence already shows how. To simulate Equation 38.32 on a grid $t_n = n\,\Delta t$ we freeze the coefficients over each step, as forward Euler does for ODEs (Section 38.2), and add a Gaussian kick whose size the increment form Equation 38.26 dictates. The **Euler–Maruyama** update (Maruyama 1955) is

$$\hat{X}_{n+1} = \hat{X}_n + f(\hat{X}_n, t_n)\,\Delta t + g(\hat{X}_n, t_n)\,\sqrt{\Delta t}\,\xi_n, \qquad \xi_n \sim \mathcal{N}(0,1) \text{ i.i.d.}, \tag{38.42}$$

where $\hat{X}_n$ approximates X_{t_n} and $\hat{X}_0 = X_0$. The kick scales as $\sqrt{\Delta t}$, since any other exponent makes the noise vanish or explode in the limit (Section 38.4.1.2), and with $g = 0$ the scheme is forward Euler.

For the OU process the step reads

$$\hat{X}_{n+1} = (1 - \theta\,\Delta t)\,\hat{X}_n + g\sqrt{\Delta t}\,\xi_n = 0.99\,\hat{X}_n + 0.1414\,\xi_n, \tag{38.43}$$

where the second form is the running example at $\Delta t = h = 0.01$, the recurrence we have been running all along.

For this process we could have done better. With $g^2 = 2\theta$ the kernel Equation 38.37 over one interval is $\mathcal{N}(e^{-\theta\Delta t}x, 1 - e^{-2\theta\Delta t})$, and sampling from it is the forward step of DDPM (Section 17.2), $x_k = \sqrt{1-\beta_k}\,x_{k-1} + \sqrt{\beta_k}\,\epsilon_k$ with $\beta_k = 1 - e^{-2\theta\Delta t}$. DDPM's step is

therefore exact at every step size, and Euler–Maruyama is its first-order approximation. At $\theta = 1$ and $\Delta t = 0.01$ the exact step multiplies by $e^{-0.01} = 0.99005$ and adds noise of variance $\beta_k = 1 - e^{-0.02} = 0.0198$, against 0.99 and 0.02. The two betas should not be confused. The β_k of the step is the fraction of the variance that one step replaces by noise. It is about $\beta(t)\,\Delta t$ for the rate $\beta(t)$ of Equation 38.41, here 0.0198 against $2 \cdot 0.01$. Exact steps need a linear equation (Exercise 7). A general drift has no transition law in closed form, and the approximate scheme is all we have.

The helper below implements Equation 38.42 for a whole batch of paths. It takes the increments ΔW as an argument, so that two simulations can be driven by the same noise. We need that at once: with the noise switched on, "error" itself needs a definition, and there are two.

```
def euler_maruyama(f, g, x0, t, dW):
    """Simulate dX = f dt + g dW on the grid t for a batch of paths."""
    X = np.empty((len(x0), len(t)))
    X[:, 0] = x0
    for k in range(len(t) - 1):
        h = t[k + 1] - t[k]
        X[:, k + 1] = (X[:, k] + f(X[:, k], t[k]) * h
                       + g(X[:, k], t[k]) * dW[:, k])
    return X
```

38.4.4.2 Strong and Weak Convergence

A simulated path is random, and so is the true one. The two definitions of error differ in what they compare: paths or laws, the two kinds of statement that the band experiment told apart.

Let $N = T/\Delta t$ be the number of steps, so that $\hat{X}_N$ approximates X_T. Drive the scheme and the exact solution with the same Brownian increments. The **strong error** $\mathbb{E}\,|\hat{X}_N - X_T|$ measures pathwise tracking: does the simulated trajectory follow the true trajectory driven by that noise? The **weak error** $|\mathbb{E}\,\varphi(\hat{X}_N) - \mathbb{E}\,\varphi(X_T)|$, for smooth test functions φ, measures only whether the law at time T comes out right. A generative model needs weak accuracy, since it only uses the samples at the final time. Strong accuracy matters when paths are coupled across resolutions, as in multilevel Monte Carlo, which makes most of its estimate on cheap coarse paths and corrects it with a few expensive fine ones (Exercise 8). Figure 38.16 shows the two notions side by side.

A scheme has strong order p if its strong error is at most $C\,\Delta t^p$, and weak order p likewise. The orders of Euler–Maruyama are classical.

Theorem (orders of Euler–Maruyama; stated without proof). *Let f, g, and φ be smooth, with the growth conditions and the moments of X_0 that Kloeden and Platen (1992) make precise.*

1. *Euler–Maruyama has strong order $\frac{1}{2}$: $\mathbb{E}\,|\hat{X}_N - X_T| \le C\,\Delta t^{1/2}$.*
2. *If the noise is additive, $g = g(t)$, the strong order is 1: $\mathbb{E}\,|\hat{X}_N - X_T| \le C\,\Delta t$.*
3. *The weak order is 1, for additive and multiplicative noise alike: $|\mathbb{E}\,\varphi(\hat{X}_N) - \mathbb{E}\,\varphi(X_T)| \le C\,\Delta t$.*

A count of single-step errors explains the exponents. It is not a proof: it adds the errors of the steps as if each stayed where it was made, and the proofs must also control how the later

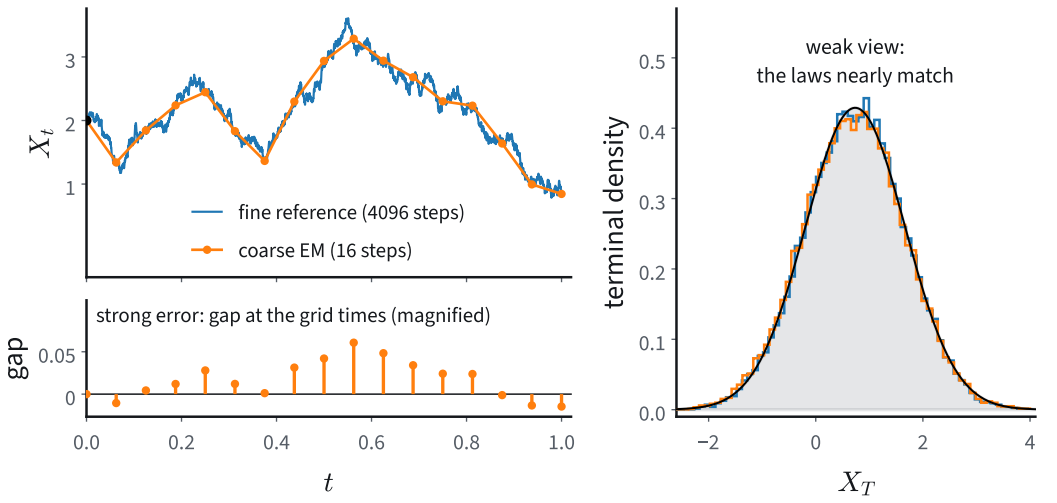


Figure 38.16: Strong versus weak convergence of Euler–Maruyama on the running example from $X_0 = 2$ on $[0, 1]$. Left: one coarse path (16 steps, orange) against a fine reference (4096 steps, blue) driven by the same Brownian increments; the strip below magnifies their gap at the coarse grid times, which is what the strong error measures. Right: the terminal histograms of 20,000 paths at the two step sizes nearly match each other and the exact law (black), although at 16 steps the variance is about 4% too large.

dynamics carry an early error forward.

Euler–Maruyama freezes g over the step, but g moves with the path. At a time s inside the step the path has moved by about $g\,(W_s - W_{t_n})$, so the noise coefficient there is about $g + g\,\partial_x g\,(W_s - W_{t_n})$. The largest term that the scheme leaves out is the integral of the second part against dW, which is Equation 38.31 for the Brownian motion $W_s - W_{t_n}$ that starts afresh at t_n:

$$g\,\partial_x g \int_{t_n}^{t_{n+1}} \left(W_s - W_{t_n}\right) dW_s = \tfrac{1}{2}\, g\,\partial_x g\,\left((\Delta W)^2 - \Delta t\right).$$

Milstein's scheme puts this term back (Milstein 1975). Its random factor is the rule for squares seen a step at a time: $(\Delta W)^2 = \Delta t$ is true in sums, and off by something of mean zero and size Δt in every step. The $N = T/\Delta t$ steps contribute uncorrelated terms of this kind, which add like a random walk to $\sqrt{N}\,\Delta t = \sqrt{T\,\Delta t}$. That is order $\frac{1}{2}$.

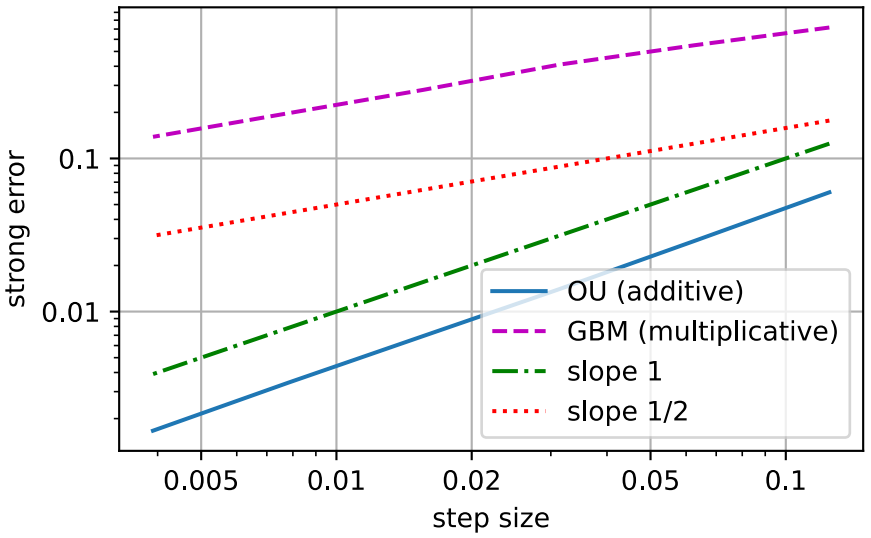
For additive noise $\partial_x g = 0$ removes this term. The next omitted stochastic terms have size $\Delta t^{3/2}$ per step and add to $\sqrt{N}\,\Delta t^{3/2} = O(\Delta t)$, as do the deterministic remainders, $N\,\Delta t^2 = O(\Delta t)$, and the order is 1. For the weak error only means matter. Every omitted stochastic term has mean zero, the bias of one step is of order Δt^2, and N steps give $O(\Delta t)$, whatever the noise.

We measure both strong orders on the same noise. Geometric Brownian motion ($\mu = \nu = 1$, $X_0 = 1.5$) is compared with its exact solution Equation 38.34, and the running example from $X_0 = 2$ with a reference on a 4096-step grid, since the noise integral in its exact

solution Equation 38.38 could itself be evaluated only on a grid. Coarse increments are sums of consecutive fine ones, so every resolution sees the same Brownian path.

```
T, n_ref, n_em = 1.0, 2**12, 2000
mu_g, nu_g = 1.0, 1.0                       # GBM: multiplicative noise
ou_f, ou_g = lambda x, s: -theta_ou * x, lambda x, s: g_ou
gbm_f, gbm_g = lambda x, s: mu_g * x, lambda x, s: nu_g * x
x0_ou, x0_gbm = np.full(n_em, 2.0), np.full(n_em, 1.5)
dW_f = np.sqrt(T / n_ref) * rng.standard_normal((n_em, n_ref))
t_f = np.linspace(0, T, n_ref + 1)
ou_ref = euler_maruyama(ou_f, ou_g, x0_ou, t_f, dW_f)[:, -1]
gbm_exact = 1.5 * np.exp((mu_g - nu_g**2 / 2) * T + nu_g * dW_f.sum(axis=1))
dts, errs_ou, errs_gbm = [], [], []
for n in [2**k for k in range(3, 9)]:        # 8 to 256 steps
    dW = dW_f.reshape(n_em, n, n_ref // n).sum(axis=2)
    t = np.linspace(0, T, n + 1)
    ou = euler_maruyama(ou_f, ou_g, x0_ou, t, dW)[:, -1]
    gbm = euler_maruyama(gbm_f, gbm_g, x0_gbm, t, dW)[:, -1]
    dts.append(T / n)
    errs_ou.append(np.abs(ou - ou_ref).mean())
    errs_gbm.append(np.abs(gbm - gbm_exact).mean())
print('strong-order slopes:  OU (additive) '
      f'{np.polyfit(np.log(dts), np.log(errs_ou), 1)[0]:.2f},'
      '  GBM (multiplicative) '
      f'{np.polyfit(np.log(dts), np.log(errs_gbm), 1)[0]:.2f}')
d2l.plot(dts, [errs_ou, errs_gbm, np.array(dts), 0.5 * np.sqrt(dts)],
         'step size', 'strong error', xscale='log', yscale='log',
         legend=['OU (additive)', 'GBM (multiplicative)',
                 'slope 1', 'slope 1/2'], figsize=(5, 3))
```

```
strong-order slopes:  OU (additive) 1.03,  GBM (multiplicative) 0.48
```



The measured slopes are close to 1 for the additive-noise OU process and close to $\frac{1}{2}$ for the multiplicative-noise geometric Brownian motion, the two strong orders of the theorem.

The two curves have different heights because the processes have different scales. What can be compared is how they fall: each halving of the step halves the OU error and divides the error of geometric Brownian motion by about $\sqrt{2}$.

For the weak order no sampling is needed. Euler–Maruyama applied to the OU process is a linear Gaussian recursion: if $\hat{X}_n$ is Gaussian, so is $\hat{X}_{n+1}$. So the simulated marginal is Gaussian at every step, with mean and variance

$$m_{n+1} = (1 - \theta\,\Delta t)\,m_n, \qquad v_{n+1} = (1 - \theta\,\Delta t)^2\,v_n + g^2\,\Delta t.$$

The mean is multiplied by $1 - \theta\Delta t$ per step where the exact solution multiplies by $e^{-\theta\Delta t}$ (0.900 against 0.905 at $\Delta t = 0.1$). The variance recursion sums a geometric series, $v_n = g^2\Delta t\,(1 - a^{2n})/(1 - a^2)$ with $a = 1 - \theta\Delta t$ and $v_0 = 0$, whose limit is the fixed point v_∞ found earlier. Evaluating these closed forms at $T = 1$ from $X_0 = 2$ for six step counts gives the errors of the simulated law with no sampling error to blur their slopes.

```
mean_T, var_T = 2 * np.exp(-1), 1 - np.exp(-2)  # exact OU kernel at T = 1
print(f'{"n":>4} {"EM mean":>8} {"EM var":>7}'
      f'   exact {mean_T:.4f}, {var_T:.4f}')
dts, errs = [], []
for n in [10, 20, 40, 80, 160, 320]:
    h = 1 / n
    a_h = 1 - theta_ou * h                  # one-step mean multiplier
    m = 2 * a_h**n                          # EM mean after n steps
    v = g_ou**2 * h * (1 - a_h**(2 * n)) / (1 - a_h**2)   # geometric series
    print(f'{n:4d} {m:8.4f} {v:7.4f}')
    dts.append(h)
    errs.append([abs(m - mean_T), abs(v - var_T)])
s_m, s_v = np.polyfit(np.log(dts), np.log(np.array(errs)), 1)[0]
print(f'weak-order slopes:  mean {s_m:.2f},  variance {s_v:.2f}')
```

```
   n  EM mean  EM var   exact 0.7358, 0.8647
  10   0.6974  0.9247
  20   0.7170  0.8938
  40   0.7265  0.8791
  80   0.7311  0.8718
 160   0.7335  0.8682
 320   0.7346  0.8664
weak-order slopes:  mean 1.01,  variance 1.01
```

At 10 steps the simulated law has mean 0.697 and variance 0.925, against the exact 0.736 and 0.865 (the kernel $\mathcal{N}(0.736,\,0.930^2)$ from before), and at 320 steps it has 0.735 and 0.866. Both fitted slopes are 1.01: halving the step halves the error of the simulated law, which is weak order 1.

The same computation for geometric Brownian motion also gives weak order 1, so multiplicative noise costs half an order in the paths and nothing in the law. A scheme that looks more accurate can even converge to the wrong equation. Heun's method, which averages the noise coefficient over the step, converges to the Stratonovich solution instead of the Itô one (Exercise 6).

Simulation produces samples of the forward process, and the kernel gives its law from a single start. Generation asks for more. Data are not a single point: the marginal p_t of the

noised data is the kernel averaged over all starts, and for real data, which we know only through samples, it has no closed form. Yet running the process backward will require the gradient of $\log p_t$, the quantity that we could write down for one start, and so far we do not even have an equation that p_t obeys.

38.4.5 Summary

A forward process must forget its start, which no deterministic flow can do, so it is driven by noise. Noise that survives the limit of small steps has size $\sqrt{\Delta t}$, and the squares of its increments add up to the elapsed time, $(dW)^2 = dt$. An integral against dW therefore depends on where its integrand is evaluated, and Itô's left endpoint gives it mean zero and the isometry $\mathbb{E}[(\int H\,dW)^2] = \int \mathbb{E}[H^2]\,ds$. For the same reason the chain rule gains the correction $\frac{1}{2}g^2\phi_{xx}\,dt$. With these tools the Ornstein–Uhlenbeck process $dX = -\theta X\,dt + g\,dW$ is solved: given its start x_0 it is Gaussian at every time, with mean $x_0e^{-\theta t}$ and variance $\frac{g^2}{2\theta}(1-e^{-2\theta t})$, and with $g^2 = 2\theta$ it is the variance-preserving forward process of diffusion models. Equations without a closed form are simulated by Euler–Maruyama, Euler's step plus a $\sqrt{\Delta t}$ kick, which follows paths at order 1 for additive noise and $\frac{1}{2}$ for multiplicative noise and reproduces laws at order 1 for both.

38.4.6 Exercises

1. **Infinite total variation.** The total variation of a path x along a partition $0 = t_0 < t_1 < \cdots < t_n = t$ is $V_n = \sum_i |x(t_{i+1}) - x(t_i)|$. Pathwise integrals $\int H\,dx$, defined as limits of sums $\sum_i H(s_i)\,\Delta x_i$ with arbitrary evaluation points $s_i \in [t_i, t_{i+1}]$, exist for every continuous H when V_n stays bounded as the mesh shrinks.
    1. Let $x = W$ and use the uniform partition with $\Delta t_i = t/n$. Compute $\mathbb{E}[V_n]$ and $\mathrm{Var}(V_n)$, using $\mathbb{E}|\xi| = \sqrt{2/\pi}$ for $\xi \sim \mathcal{N}(0,1)$, and conclude that $V_n \to \infty$ in probability as $n \to \infty$.
    2. Let x be continuous with $V_n \le V$ for every partition. Show that $\max_i |\Delta x_i| \to 0$ as the mesh shrinks, and deduce $\sum_i (\Delta x_i)^2 \le \max_i |\Delta x_i| \cdot V \to 0$: such a path has quadratic variation zero. Why does Equation 38.27 then give a second proof that Brownian paths have unbounded total variation?
    3. Let x be continuous with bounded total variation. Show that the left-endpoint sums $\sum_i x(t_i)\,\Delta x_i$ and the right-endpoint sums $\sum_i x(t_{i+1})\,\Delta x_i$ both converge to $\frac{1}{2}\,(x(t)^2 - x(0)^2)$. Compare with the two limits for $x = W$ in Section 38.4.2.2. Which step of your argument fails for a Brownian path?
2. code **Brownian scaling and exit times.**
    1. For $c > 0$ let $\tilde{W}_t = W_{ct}/\sqrt{c}$. Check properties 1–4 of Section 38.4.1.2 and conclude that $\tilde{W}$ is a Brownian motion. Which property of the $\pm\sqrt{\Delta t}$ walk does this make exact?
    2. Let τ_a be the first time $|W_t|$ reaches a level $a > 0$. Use (a) to show that τ_a has the same distribution as $a^2\tau_1$, so that $\mathbb{E}[\tau_a] = a^2\,\mathbb{E}[\tau_1]$.
    3. Granting $\mathbb{E}[\tau_1] = 1$ (it follows from $\mathbb{E}[W_\tau^2] = \mathbb{E}[\tau]$, a consequence of Equation 38.31 and the zero mean of the Itô integral up to a stopping time), simulate $4{,}000$ walks with Gaussian steps of variance $\Delta t = 10^{-3}$ and estimate $\mathbb{E}[\tau_a]$ for $a = 0.5, 1, 2$, with Monte Carlo standard errors. The estimates tend to run

slightly above a^2, most clearly for small a. Explain the sign of the bias from the way a discrete walk crosses a level, and say how it scales with a and Δt.

4. In the experiment of the text a little over half of the paths stay inside $\pm 2\sqrt{t}$ at all 500 grid times. Use (a) to show that this fraction depends only on the number n of grid times and not on the length of the interval. Measure it for $n = 50$, 500, and 5000. By what factor does it fall each time n grows tenfold, and what does that say about the limit $n \to \infty$?

3. **Geometric Brownian motion and the $-\nu^2/2$ correction.** Consider $dX = \mu X\, dt + \nu X\, dW$ with $X_0 > 0$ and its solution, given by Equation 38.34.

   1. Using $\mathbb{E}\left[e^{\lambda Z}\right] = e^{\lambda^2 v/2}$ for $Z \sim \mathcal{N}(0, v)$, show that $\mathbb{E}[X_t] = X_0 e^{\mu t}$ and $\mathbb{E}[X_t^2] = X_0^2 e^{(2\mu+\nu^2)t}$. Find the median of X_t.
   2. Take $\mu = 0.1$ and $\nu = 0.6$. What happens to the mean, to the median, and to a typical path as $t \to \infty$? How can these statements hold at the same time?
   3. For which ν, at a fixed $\mu > 0$, does the median stop growing? Check the condition for the parameters of (b) on 100 paths simulated with `euler_maruyama` to $t = 20$ at $\Delta t = 10^{-2}$: what fraction of the paths ends below X_0, and what fraction does the exact solution Equation 38.34 predict?

   Adapted from Simo Särkkä and Arno Solin, Applied Stochastic Differential Equations, Exercise 4.3.

4. code **Strong order and Milstein's correction.** As Section 38.4.4.2 explains, Euler–Maruyama omits the Milstein term $\frac{1}{2} g\, \partial_x g\, \left((\Delta W)^2 - \Delta t\right)$, and its strong order is $\frac{1}{2}$ unless that term vanishes.

   1. By Equation 38.34, $Y = \log X$ satisfies an SDE with additive noise. Apply `euler_maruyama` to that SDE with the increments `dW_f` and the step sizes of the strong-order experiment, exponentiate the endpoints, and measure the strong error against `gbm_exact`. Explain the result.
   2. Use Itô's lemma to show that for $dX = f(X)\, dt + g(X)\, dW$ with $g > 0$, the variable $Y = \psi(X)$ with $\psi' = 1/g$ has unit additive noise, and find its drift.
   3. Milstein's scheme (Milstein 1975) adds that term to the Euler–Maruyama step,

$$\hat{X}_{n+1} = \hat{X}_n + f\, \Delta t + g\, \Delta W_n + \tfrac{1}{2}\, g\, \partial_x g\, \left((\Delta W_n)^2 - \Delta t\right), \tag{38.44}$$

   with f, g, and $\partial_x g$ evaluated at $(\hat{X}_n, t_n)$. Implement it for the geometric Brownian motion of the strong-order experiment, and measure its strong-order slope on the same increments and step sizes. By what factor does it reduce the strong error at $\Delta t = 2^{-8}$?

5. **Variance preservation, exactly.** Let X_t follow Equation 38.40 with zero-mean data X_0.

   1. Assume unit-variance data and show that $\mathbb{E}[X_t^4] = 3 + \bar{\alpha}_t^2\, \left(\mathbb{E}[X_0^4] - 3\right)$. Specialize to Rademacher data $X_0 = \pm 1$ to obtain the prediction $3 - 2\bar{\alpha}_t^2$ of the text.
   2. The excess kurtosis $\mathbb{E}[X^4]/\mathrm{Var}(X)^2 - 3$ is zero for every Gaussian. At what rate in t does it decay for X_t? Compare this rate with the rate at which the conditional mean $\mathbb{E}[X_t \mid X_0]$ forgets the data.

3. Suppose instead that the data have variance s^2. Find $\mathrm{Var}(X_t)$. For $s = 2$ and $\theta = 1$, how long must the process run before $\mathrm{Var}(X_T)$ is within 1% of the variance of the reference $\mathcal{N}(0, 1)$?

6. code **Itô versus Stratonovich.** ● The left- and right-endpoint sums for $\int_0^t W\,dW$ add up to W_t^2 on every partition (Section 38.4.2.2), so their average equals $\frac{1}{2}W_t^2$. Sums of this kind define the Stratonovich integral (Stratonovich 1966), which obeys the ordinary chain rule.

   1. Heun's method of Section 38.2 carries over to SDEs if the noise term is treated like a second slope. The predictor is the Euler–Maruyama step $\tilde{X} = \hat{X}_n + f(\hat{X}_n)\,\Delta t + g(\hat{X}_n)\,\Delta W_n$, and the corrector averages both coefficients:

$$\hat{X}_{n+1} = \hat{X}_n + \tfrac{1}{2}\left[f(\hat{X}_n) + f(\tilde{X})\right]\Delta t + \tfrac{1}{2}\left[g(\hat{X}_n) + g(\tilde{X})\right]\Delta W_n. \quad (38.45)$$

      Implement it for the geometric Brownian motion of the strong-order experiment, on its increments `dW_f` and step sizes. Compare its endpoints with the exact Itô solution Equation 38.34 and with $X_0 \exp(\mu T + \nu W_T)$. To which of the two does the scheme converge?

   2. Expand the corrector's noise term to first order in $\tilde{X} - \hat{X}_n$, and use $(\Delta W_n)^2 \approx \Delta t$ to show that the scheme behaves like Euler–Maruyama for an Itô SDE with a modified drift. Find that drift and check that it explains the limit in (a).

   3. Why do the two conventions give the same solution for every SDE of the form Equation 38.25?

7. **Exact transitions of linear SDEs.** ● Consider the linear SDE $dX = -\theta(t)X\,dt + g(t)\,dW$ with continuous θ and g, and let $\Theta(t) = \int_0^t \theta(u)\,du$.

   1. Apply Itô's lemma to $\phi(x, t) = e^{\Theta(t)}x$ and show that, given X_s, the state X_t at a later time t is Gaussian with mean $e^{-(\Theta(t)-\Theta(s))}X_s$ and variance $\int_s^t e^{-2(\Theta(t)-\Theta(u))}\,g(u)^2\,du$.
   2. Specialize (a) to the VP-SDE Equation 38.41 and show that a transition of any length maps a unit-variance state to a unit-variance state. Conversely, assume for the linear SDE of (a) that $\mathrm{Var}(X_t) = 1$ for every $t \geq 0$ whenever $\mathrm{Var}(X_0) = 1$, and show that $g(t)^2 = 2\,\theta(t)$ at every t.
   3. Consider the semilinear SDE $dX = (-\theta X + q(X))\,dt + g\,dW$ with constants $\theta, g > 0$ and a smooth function q. The exponential Euler–Maruyama step holds q at the value $q(\hat{X}_n)$ over a step of length Δt and solves the remaining linear SDE exactly, by the method of (a). Write the step. Show that it agrees with the Euler–Maruyama step Equation 38.42 to first order in Δt and that it is exact when q is constant. For $q \equiv 0$, Euler–Maruyama damps the mean only for $\Delta t < 2/\theta$ (Section 38.2.3). What limits the step size of the exponential step?
   4. Exact steps have zero weak error at every step size. Which property of the drift and the diffusion of the VP-SDE makes them available? The generative sampler of a diffusion model simulates, backward in time, an SDE whose drift is $-\frac{1}{2}\beta(t)\,x - \beta(t)\,\nabla \log p_t(x)$: the VP drift plus a multiple of $\nabla \log p_t$, the *score* of the forward marginal, which is a nonlinear function of x for non-Gaussian

data. Why can its steps not be made exact? Exponential integrators for diffusion samplers (Zhang and Chen 2023) apply the idea of (c) to the deterministic counterpart of this sampler, an ODE whose drift has the same two parts with half the score term (Section 38.5). Which part of such a drift does the step treat exactly, and which part does it approximate?

8. [extended] **Multilevel Monte Carlo.** ●● Estimating $\mathbb{E}[\varphi(X_T)]$ by averaging over Euler–Maruyama paths has two sources of error, the weak bias of the scheme and the sampling noise. Multilevel Monte Carlo (Michael B. Giles 2008) reduces the cost by coupling paths across step sizes, which is one of the uses of strong accuracy named in Section 38.4.4.2. Throughout, the mean-square strong error $\mathbb{E}\big[(\hat{X}_N - X_T)^2\big]^{1/2}$ has the same orders as the strong error of that section.

   1. Plain Monte Carlo averages $\varphi(\hat{X}_N)$ over M independent paths at a cost of $MT/\Delta t$ steps. Assume a weak bias $c_1\,\Delta t$ and a variance v of $\varphi(\hat{X}_N)$. Show that a mean squared error below ε^2 costs order ε^{-3} steps at the best choice of M and Δt.
   2. Let P_l be φ of the Euler–Maruyama endpoint with step $h_l = T\,2^{-l}$, and write $\mathbb{E}[P_L] = \mathbb{E}[P_0] + \sum_{l=1}^{L} \mathbb{E}[P_l - P_{l-1}]$. Each term is estimated from M_l independent pairs in which the coarse path uses the sums of consecutive pairs of fine increments, as in the strong-order experiment. For Lipschitz φ, bound $V_l = \mathrm{Var}(P_l - P_{l-1})$ by the mean-square strong error. Take $\varphi(x) = x$, and predict how V_l scales with h_l for the OU process and the geometric Brownian motion of that experiment, with its parameters. Measure V_l for $l = 1, \dots, 8$ with `euler_maruyama`, and fit the slopes on the finer levels.
   3. Minimize the total cost $\sum_l M_l/h_l$ subject to $\sum_l V_l/M_l \le \varepsilon^2/2$ with a Lagrange multiplier. Choose the finest level L so that the bias stays below $\varepsilon/\sqrt{2}$. Assuming $V_l \propto h_l^r$, show that the cost is of order ε^{-2} for $r = 2$ and of order $\varepsilon^{-2}(\log \varepsilon)^2$ for $r = 1$.
   4. Implement both estimators for $\mathbb{E}[X_T] = X_0 e^{\mu T}$ of the geometric Brownian motion of the strong-order experiment, with M_l from
      (c) and your measured V_l. For $\varepsilon \in \{0.2, 0.1, 0.05, 0.02\}$, record the total number of steps each estimator uses and check its error over a few repetitions. Plot ε^2 times the cost against ε. Repeat with the Milstein scheme of Exercise 4c in place of Euler–Maruyama. `euler_maruyama` stores every path in full, so simulate large M_l in batches.
   5. Repeat the measurement of (b) for the discontinuous $\varphi(x) = \mathbf{1}(x > X_0)$. Why does V_l decay more slowly, and what does the slower decay do to the cost?

38.5 The Fokker–Planck Equation and Probability Flow

The previous section simulated the Ornstein–Uhlenbeck process, the stochastic differential equation $dX = -X\,dt + \sqrt{2}\,dW$ that pulls a particle toward the origin while kicking it with noise. Every run drew different paths, and no single path can be predicted. Yet something repeats from run to run. Start a hundred thousand particles at $x = 2$ and look at their histogram at time $t = 0.5$: on every run it is the same bell curve, centered near 1.21 with a standard deviation near 0.80. The randomness of the paths averages out, and the density of the cloud moves deterministically. Which equation moves it?

Adriaan Fokker and Max Planck wrote that equation for particles that feel a steady force and random kicks (Fokker 1914; Planck 1917), and Andrey Kolmogorov showed that it governs the density of every diffusion (Kolmogorov 1931). It has two terms: the drift carries the density along, and the noise smooths it. The smoothing can also be written as transport, a flow that moves probability from where the density is high toward where it is low. Seen at the level of the density, noise acts like a deterministic velocity.

This matters for generative models. A diffusion model generates data by running a noising process backward. No map takes the endpoint of a noisy path back to its start, since many starts lead to the same endpoint. But we know how the density of the whole cloud changed, and we can build processes whose density changes the other way. One is an ordinary differential equation, obtained by replacing the noise with the outward flow; its particles have the same histogram as the stochastic process at every time. The other is again a stochastic process, and it turns noise into data. In both, the drift and the amount of noise are ours to choose. One quantity is not: the direction in which the log-density increases, at every point and time.

We check each step on the Ornstein–Uhlenbeck process and on a two-mode "data" density for which that direction has a closed form. For real data it is unknown, and this is the question we leave open. If the derivations fade, three displays should stay: the Fokker–Planck equation Equation 38.52, the identity Equation 38.63 that turns its smoothing term into transport, and the family of reverse processes Equation 38.71. A reader at home with the divergence theorem can skip Section 38.5.2.1, and the likelihood formula and the bias computation at the end can wait for a second reading.

38.5.1 From Paths to Densities

38.5.1.1 Paths and Marginals

Every SDE in this section is

$$d\mathbf{X} = \mathbf{f}(\mathbf{X}, t)\, dt + g(t)\, d\mathbf{W},$$

with a drift $\mathbf{f} : \mathbb{R}^d \times [0, T] \to \mathbb{R}^d$, a scalar noise amplitude $g(t)$, a d-dimensional Brownian motion $\mathbf{W}$, and a random start $\mathbf{X}_0 \sim p_0$. Over a short step the particle moves by $\mathbf{f}\, dt$ plus an independent Gaussian kick of variance $g^2\, dt$ in each coordinate.

One realization of the Brownian motion produces one path $\mathbf{X}_t$, different on every run. The object of this section is instead the density $p_t(\mathbf{x})$ of the random variable $\mathbf{X}_t$, called the **time-marginal**. It averages over both sources of randomness, the draw of $\mathbf{X}_0$ and the noise along the way, and this averaging makes it deterministic.

Our running process is the Ornstein–Uhlenbeck (OU) process of Section 38.4.3, $dX = -\theta X\, dt + g\, dW$. Started from a point $X_0 = x_0$, its marginal is the Gaussian kernel, which we write as

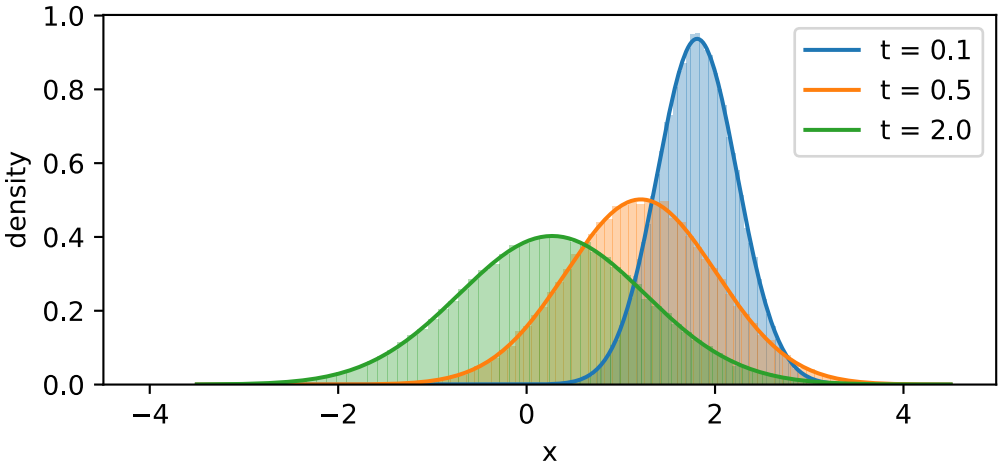
$$X_t \sim \mathcal{N}(\alpha_t x_0,\ \sigma_t^2), \qquad \alpha_t = e^{-\theta t}, \quad \sigma_t^2 = \frac{g^2}{2\theta}\left(1 - e^{-2\theta t}\right). \tag{38.46}$$

We take $\theta = 1$ and $g = \sqrt{2}$ throughout, so the variance saturates at $g^2/(2\theta) = 1$ and the stationary law is $\mathcal{N}(0, 1)$. The drift moves the center of the cloud and the noise widens it; the Fokker–Planck equation will turn both actions into operators on p_t.

38.5.1.2 Simulated Marginals

The SDE section compared the moments of such a cloud with the kernel. Here we look at the whole histogram: $100{,}000$ particles start at $x_0 = 2$ and move by Euler–Maruyama steps of size $\Delta t = 0.002$ (Section 38.4.4.1), and at $t = 0.1$, 0.5 and 2 the cloud is drawn against its exact Gaussian density.

```
theta, g = 1.0, np.sqrt(2.0)            # OU: dX = -theta X dt + g dW

def ou_mean_var(x0, t, theta=theta, g=g):
    """Mean and variance of X_t started at x0: N(alpha_t x0, sigma_t^2)."""
    alpha = np.exp(-theta * t)
    return alpha * x0, g**2 / (2 * theta) * (1 - alpha**2)

rng = np.random.default_rng()
n, dt, snap_at = 100_000, 0.002, {50: 0.1, 250: 0.5, 1000: 2.0}  # step -> t
x, snaps = np.full(n, 2.0), {}          # every particle starts at x0 = 2
for k in range(1, max(snap_at) + 1):    # Euler--Maruyama
    x += -theta * x * dt + g * np.sqrt(dt) * rng.standard_normal(n)
    if k in snap_at:
        snaps[snap_at[k]] = x.copy()

grid = np.linspace(-3.5, 4.5, 400)
d2l.set_figsize((6, 2.5))
for (t, cloud), c in zip(snaps.items(), ('C0', 'C1', 'C2')):
    m, v = ou_mean_var(2.0, t)
    d2l.plt.hist(cloud, bins=80, density=True, alpha=0.35, color=c)
    d2l.plt.plot(grid, np.exp(-(grid - m)**2 / (2 * v)) / np.sqrt(2 * np.pi * v),
                 color=c, label=f't = {t}')
d2l.plt.xlabel('x'); d2l.plt.ylabel('density'); d2l.plt.legend();
```



At each of the three times the histogram lies on its curve. By Equation 38.46 the curves have mean $2e^{-t}$ and standard deviation $\sqrt{1 - e^{-2t}}$; they are drawn from the formula, not fitted to the cloud. At $t = 2$ the mean is still $2e^{-2} \approx 0.27$, about a quarter of a standard deviation away from the stationary $\mathcal{N}(0, 1)$: forgetting a start takes time. This is one run, but the

three curves it matches contain no randomness. A second run would differ in every path and match the same curves. It is time to find the equation that moves them.

38.5.2 The Fokker–Planck Equation

A path cannot be differentiated in time, but the average of a smooth function of $\mathbf{X}_t$ over all paths can. Turning a statement about averages into a statement about the density requires moving derivatives from the function onto the density, and that needs three facts about fields in $\mathbb{R}^d$.

38.5.2.1 Divergence, Laplacian, and Integration by Parts

All three are built from the gradient of Section 34.2, and each has a physical meaning that we will need again. Let $\mathbf{u} : \mathbb{R}^d \to \mathbb{R}^d$ be a smooth vector field, which we picture as the velocity of a fluid, and $\psi : \mathbb{R}^d \to \mathbb{R}$ a smooth scalar field.

Divergence. We want to know how much fluid leaves a small box per unit time. The **divergence** of $\mathbf{u}$ is the scalar field

$$\nabla \cdot \mathbf{u} = \sum_{i=1}^{d} \frac{\partial u_i}{\partial x_i}. \tag{38.47}$$

It is the net rate at which fluid leaves an infinitesimal box around $\mathbf{x}$, divided by the volume of the box: positive at a source, negative at a sink. Two plane fields show the range. The expanding field $\mathbf{u} = (x_1, x_2)$ has $\nabla \cdot \mathbf{u} = 1 + 1 = 2$, so every box of area A loses fluid at the rate $2A$, wherever it sits. The rotation $\mathbf{u} = (-x_2, x_1)$ has $\nabla \cdot \mathbf{u} = 0 + 0 = 0$: fluid circulates, and no box gains or loses any. A zero divergence therefore does not mean a zero field. Figure 38.17 checks both statements on one box.

Summing the outflow of many small boxes cancels every interior face, which leaves the flux through the outer boundary. This is the **divergence theorem**: for a bounded region Ω with outward unit normal $\mathbf{n}$,

$$\int_{\Omega} \nabla \cdot \mathbf{u} \, d\mathbf{x} = \oint_{\partial\Omega} \mathbf{u} \cdot \mathbf{n} \, dS. \tag{38.48}$$

The total divergence inside a region equals the net flow out through its boundary. In one dimension it is the fundamental theorem of calculus, $\int_a^b u'(x)\,dx = u(b) - u(a)$. We use the theorem without proof; it is proved in any text on multivariable calculus.

Laplacian. We want a measure of how a function compares with its surroundings. The **Laplacian** is the divergence of the gradient (in front of a function, Δ is this operator and not an increment as in Δt),

$$\Delta\psi = \nabla \cdot (\nabla\psi) = \sum_{i=1}^{d} \frac{\partial^2 \psi}{\partial x_i^2}. \tag{38.49}$$

It compares ψ at a point with its average over a small sphere of radius r around the point: that average is $\psi(\mathbf{x}) + \frac{r^2}{2d}\Delta\psi(\mathbf{x}) + O(r^4)$, which Exercise 1 proves. In one dimension, with

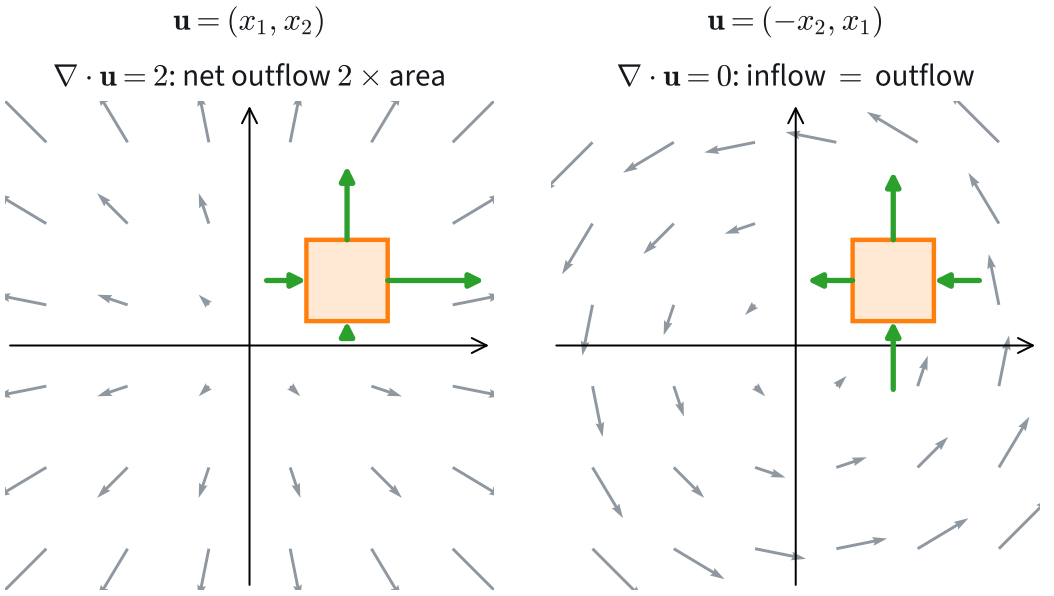


Figure 38.17: The divergence as net outflow, on the box of side 0.5 centered at $(0.6, 0.4)$. Each green arrow is the normal component of the field at the midpoint of a side, drawn away from the side for outflow and onto it for inflow. (a) For $\mathbf{u} = (x_1, x_2)$ fluid enters through the left and bottom sides and leaves through the right and top; the outflow exceeds the inflow by $2\times$ the area. (b) For the rotation $\mathbf{u} = (-x_2, x_1)$ the inflow through the right and bottom sides equals the outflow through the other two.

$\psi(x) = x^2$, the "sphere" is the pair of points $x \pm r$, the average is $\frac{1}{2}\big((x-r)^2 + (x+r)^2\big) = x^2 + r^2$, and $\frac{r^2}{2}\psi'' = r^2$, so the formula holds with no remainder (at $x = 1$, $r = 0.3$ the average is 1.09). So $\Delta\psi > 0$ means that ψ lies below its neighborhood average. A zero Laplacian does not mean a constant function: $\psi = x_1^2 - x_2^2$ has $\Delta\psi = 2 - 2 = 0$ and equals its average over every circle.

Integration by parts. This is the d-dimensional form of the one-dimensional rule of Section 34.6.2.3, and it is the tool that moves a derivative from one factor onto the other. If ψ and $\mathbf{u}$ are smooth and $\psi\,\mathbf{u}$ decays fast enough at infinity (in this section, because ψ vanishes outside a bounded set or because both factors have Gaussian tails), then

$$\int \psi\,(\nabla \cdot \mathbf{u})\; d\mathbf{x} = -\int \nabla\psi \cdot \mathbf{u}\; d\mathbf{x}. \tag{38.50}$$

The derivative moves from $\mathbf{u}$ onto ψ, and the sign flips.

Proof. The product rule gives $\nabla \cdot (\psi\mathbf{u}) = \nabla\psi \cdot \mathbf{u} + \psi\,\nabla \cdot \mathbf{u}$. Integrate both sides over a ball of radius R and let $R \to \infty$. By the divergence theorem the left side is the flux of $\psi\mathbf{u}$ through the sphere of radius R, which tends to zero under the decay assumption. ■

Applied twice, first with $\mathbf{u} = \nabla p$ and then with the roles exchanged, it moves a whole Laplacian across the integral sign, and the two signs cancel: $\int \psi\,\Delta p = -\int \nabla\psi \cdot \nabla p = \int p\,\Delta\psi$.

38.5.2.2 From Itô's Lemma to the PDE

We now derive the equation for p_t. Throughout we assume that $\mathbf{f}$ is continuously differentiable in $\mathbf{x}$ and continuous in t, that g is continuous, that p_t is smooth and positive, and that derivatives in t may be taken under the integral sign.

The idea is to look at the cloud through averages. If ϕ is a smooth bump equal to 1 on a small box and 0 outside a slightly larger one, then $\mathbb{E}[\phi(\mathbf{X}_t)]$ is roughly the probability of the box. Knowing how every such average changes over time determines how the density changes. So take any smooth **test function** $\phi : \mathbb{R}^d \to \mathbb{R}$ that vanishes outside a bounded set.

Itô's lemma (Section 38.4.2.4) is the chain rule for SDE paths. For $d\mathbf{X} = \mathbf{f}\,dt + \mathbf{G}\,d\mathbf{W}$ it adds to the ordinary chain rule the second-order term $\frac{1}{2}\sum_{ij}(\mathbf{G}\mathbf{G}^{\top})_{ij}\,\partial_i\partial_j\phi$, because the squared Brownian increment $(dW)^2$ is of size dt rather than negligible. With $\mathbf{G} = g(t)\,\mathbf{I}$ that term is $\frac{1}{2}g^2$ times the Laplacian of ϕ:

$$d\phi(\mathbf{X}_t) = \big(\nabla\phi \cdot \mathbf{f} + \tfrac{1}{2}g^2\Delta\phi\big)\,dt + g\,\nabla\phi \cdot d\mathbf{W}.$$

The function changes by a drift part, the chain rule plus the Laplacian correction, and a noise part. Now take expectations. The Itô integral of $g\nabla\phi$ has mean zero, because each Brownian increment is independent of the integrand at the start of that increment (Section 38.4.2.2). The noise part drops out:

$$\frac{d}{dt}\,\mathbb{E}[\phi(\mathbf{X}_t)] = \mathbb{E}\big[\nabla\phi \cdot \mathbf{f} + \tfrac{1}{2}g^2\Delta\phi\big]\,. \tag{38.51}$$

The average of ϕ changes at the average of the rate that Itô's lemma gave. Both sides are integrals against the marginal density. Writing $\mathbb{E}[\,\cdot\,] = \int (\cdot)\, p_t(\mathbf{x})\, d\mathbf{x}$ and moving d/dt under the integral,

$$\int \phi\, \partial_t p_t \, d\mathbf{x} = \int (\nabla\phi \cdot \mathbf{f})\, p_t \, d\mathbf{x} + \tfrac{1}{2} g^2 \int (\Delta\phi)\, p_t \, d\mathbf{x}.$$

The derivatives on the right still act on ϕ; we move them onto p_t with integration by parts Equation 38.50. Because ϕ vanishes outside a bounded set, no boundary terms appear and no decay of p_t is needed. The drift term needs one move, with $\psi = \phi$ and $\mathbf{u} = \mathbf{f} p_t$; the diffusion term needs two, and its signs cancel:

$$\int (\nabla\phi \cdot \mathbf{f})\, p_t \, d\mathbf{x} = -\int \phi\, \nabla \cdot (\mathbf{f} p_t)\, d\mathbf{x}, \qquad \int (\Delta\phi)\, p_t \, d\mathbf{x} = \int \phi\, \Delta p_t \, d\mathbf{x}.$$

Substituting both gives one integral against ϕ:

$$\int \phi \left(\partial_t p_t + \nabla \cdot (\mathbf{f} p_t) - \tfrac{1}{2} g^2 \Delta p_t \right) d\mathbf{x} = 0.$$

This holds for every test function ϕ. The bracket is continuous, and a continuous function that integrates to zero against every bump is zero (a bump placed where it is positive would give a positive integral). So the bracket vanishes pointwise, and p_t solves the **Fokker–Planck equation**, also called the *Kolmogorov forward equation*:

$$\partial_t p_t(\mathbf{x}) = -\nabla \cdot \big(\mathbf{f}(\mathbf{x}, t)\, p_t(\mathbf{x})\big) + \tfrac{1}{2} g(t)^2\, \Delta p_t(\mathbf{x}). \tag{38.52}$$

The first term is the drift's contribution, the second the noise's. For a matrix noise $\mathbf{G}\, d\mathbf{W}$ the second term becomes $\frac{1}{2} \sum_{ij} \partial_i \partial_j (a_{ij} p_t)$ with $\mathbf{a} = \mathbf{G}\mathbf{G}^\top$; Exercise 5 works out what changes. Risken (1996) collects the many variants of the equation used in physics.

The derivation gives one direction only: the marginal of the SDE solves the equation. Later we need the converse, that a solution which starts from p_0 *is* the marginal, and for this the equation must have only one solution. It does under mild conditions, which every drift we use satisfies: $g > 0$, and a drift that is Lipschitz in $\mathbf{x}$ uniformly in t, that is, with a slope in $\mathbf{x}$ bounded by one constant for all t. Among probability densities with a given initial law the solution is then unique (Bogachev et al. (2015) give precise statements). We use this uniqueness without proof, and say so where we do.

38.5.2.3 The Heat Equation and the Two Terms

Switch the drift off. The Fokker–Planck equation is then the **heat equation**,

$$\mathbf{f} = \mathbf{0}: \qquad \partial_t p_t = \tfrac{1}{2} g^2\, \Delta p_t. \tag{38.53}$$

The density rises wherever it lies below its neighborhood average. Before any general statement, we check it at one point. Take $g = 1$, $d = 1$ and $p_t = \mathcal{N}(0, t)$, the law of Brownian motion started at 0. With $p = e^{-x^2/(2t)}/\sqrt{2\pi t}$, direct differentiation gives

$\partial_t p = p\,(\frac{x^2}{2t^2} - \frac{1}{2t})$ and $\partial_{xx} p = p\,(\frac{x^2}{t^2} - \frac{1}{t})$. At $x = 0$, $t = 1$: $p = 0.399$, $\partial_t p = -0.1995$ and $\frac{1}{2}\partial_{xx} p = -0.1995$. The peak sinks at the rate the Laplacian predicts.

More generally, a Gaussian $\mathcal{N}(\mathbf{m}, v_t \mathbf{I})$ solves the heat equation when its variance grows as $v_t = v_0 + g^2 t$ for constant g, or as $v_t = v_0 + \int_0^t g(\tau)^2\, d\tau$ when g changes with time. This is the $\mathrm{Var}(W_t) = t$ law of Brownian motion. Noise alone is the *variance-exploding* family of Section 38.4.1.1, which diffusion models use with a growing $g(t)$.

Now read the two terms of Equation 38.52. The first, $-\nabla \cdot (\mathbf{f} p)$, is *advection*: it removes density wherever the drift carries more mass out of a small box than in. The second, $\frac{1}{2} g^2 \Delta p$, is *diffusion*: it raises the density where it lies below its neighborhood average and lowers it where it lies above. It is Itô's $(dW)^2 = dt$ correction, seen at the level of the density.

38.5.2.4 Gaussian Solutions of the OU Equation

For the OU process we know the answer in closed form, which lets us test the equation on a whole family of solutions. With $f(x) = -\theta x$ in one dimension, Equation 38.52 reads

$$\partial_t p = \theta\, \partial_x (x\, p) + \tfrac{1}{2} g^2\, \partial_{xx} p.$$

The unknown of a partial differential equation is a whole function, and one rarely finds it by hand. Here we are lucky. It turns out that for a Gaussian the equation collapses to two ordinary differential equations, one for the mean and one for the variance.

Proposition (Gaussian solutions of the OU equation). *Let $p(x,t) = \mathcal{N}(x; m(t), v(t))$ be a Gaussian density with differentiable mean $m(t)$ and variance $v(t) > 0$. Then p solves the OU Fokker–Planck equation if and only if*

$$\dot m = -\theta m, \qquad \dot v = g^2 - 2\theta v. \tag{38.54}$$

Proof. For a Gaussian, $\partial_x p = -\frac{x-m}{v}\, p$. Direct differentiation turns each side of the PDE into p times a quadratic polynomial in $(x - m)$:

$$\partial_t p = p\left[\frac{\dot m}{v}(x-m) + \frac{\dot v}{2v^2}\big((x-m)^2 - v\big)\right],$$

while, writing $x = (x-m) + m$ in the drift term,

$$\theta\, \partial_x (xp) + \tfrac{g^2}{2}\partial_{xx} p = p\left[-\frac{\theta m}{v}(x-m) + \Big(\frac{g^2}{2v^2} - \frac{\theta}{v}\Big)\big((x-m)^2 - v\big)\right].$$

The functions $1, (x-m), (x-m)^2$ are linearly independent, so the two sides agree for all x only when the coefficients match: $\dot m = -\theta m$ from the linear term and $\dot v = g^2 - 2\theta v$ from the quadratic term (the constant term repeats the latter). ■

The mean decays and the variance relaxes toward $g^2/(2\theta)$. Solving the two ODEs from $m(0) = m_0$, $v(0) = v_0$ gives

$$m(t) = m_0 e^{-\theta t}, \qquad v(t) = v_0 e^{-2\theta t} + \frac{g^2}{2\theta}(1 - e^{-2\theta t}).$$

For a point start, $v_0 = 0$, this is the kernel Equation 38.46 at every $t > 0$, so the transition density of Section 38.4.3 solves the Fokker–Planck equation.

The fixed point $\dot{m} = \dot{v} = 0$ is $m = 0$, $v = g^2/(2\theta)$: the stationary Gaussian is the steady state $\partial_t p = 0$ of the PDE (Exercise 3 derives it without assuming a Gaussian form). Setting $\theta = 0$ recovers the heat equation's $\dot{v} = g^2$.

The luck has a cause, and a limit. The drift is linear, so a Gaussian stays Gaussian and its first two moments form a closed system. For the double-well drift $f(x) = x - x^3$ of the same exercise, the equation for $\mathbb{E}[X^2]$ contains $\mathbb{E}[X^4]$, the one for $\mathbb{E}[X^4]$ contains $\mathbb{E}[X^6]$, and the chain never closes.

38.5.2.5 Linearity and Mixtures

Real data are not Gaussian either, so we need a density with more structure that we can still follow in closed form. With a linear drift a Gaussian mixture is one. Our running "data" density is

$$p_0 = \tfrac{1}{2}\,\mathcal{N}(-2,\, 0.25) + \tfrac{1}{2}\,\mathcal{N}(2,\, 0.25),$$

two well-separated modes of standard deviation 0.5. A draw from p_0 picks a component $\mathcal{N}(\mu_k, v_k)$ and then a point from it. By the kernel Equation 38.46, X_t is $\alpha_t X_0$ plus an independent $\mathcal{N}(0, \sigma_t^2)$ variable. Given the component, it is therefore a sum of two independent Gaussians, with mean $\alpha_t \mu_k$ and variance $\alpha_t^2 v_k + \sigma_t^2$, and averaging over the component gives the marginal: the mixture of the evolved components. With $\theta = 1$, $g = \sqrt{2}$ it is the mixture of $\mathcal{N}\big(\pm 2e^{-t},\, 1 - 0.75\, e^{-2t}\big)$, in closed form at every time.

The same fact can be read off the Fokker–Planck equation, which is linear in p. A mixture $w_1 p_t^{(1)} + w_2 p_t^{(2)}$ of solutions is therefore a solution, and by the uniqueness stated after Equation 38.52 it is the marginal for the mixed start. Under OU each Gaussian component follows the moment ODEs Equation 38.54 on its own.

Every experiment from here on uses this marginal, so we define it once, together with its CDF and a sampler. The parameters are explicit arguments, so the exercises can change them.

```
ws, mus, v0s = np.array([0.5, 0.5]), np.array([-2.0, 2.0]), np.array([0.25, 0.25])
erf = np.vectorize(math.erf)            # the error function, on arrays

def mixture_params(t, mus=mus, v0s=v0s, theta=theta, g=g):
    """Means and variances of the OU-evolved components (moment ODEs)."""
    alpha = np.exp(-theta * t)
    return alpha * mus, alpha**2 * v0s + g**2 / (2 * theta) * (1 - alpha**2)

def gauss(x, m, v):
    return np.exp(-(x - m)**2 / (2 * v)) / np.sqrt(2 * np.pi * v)

def p_t(x, t, ws=ws, mus=mus, v0s=v0s):
```

```
    """Exact marginal density of the mixture at time t."""
    m, v = mixture_params(t, mus, v0s)
    return sum(w * gauss(x, mk, vk) for w, mk, vk in zip(ws, m, v))

def mixture_cdf(x, t, ws=ws, mus=mus, v0s=v0s):
    m, v = mixture_params(t, mus, v0s)
    return sum(w * 0.5 * (1 + erf((x - mk) / np.sqrt(2 * vk)))
               for w, mk, vk in zip(ws, m, v))

def sample_p0(n, rng, ws=ws, mus=mus, v0s=v0s):
    k = rng.choice(len(ws), size=n, p=ws)   # component of each draw
    return mus[k] + np.sqrt(v0s[k]) * rng.standard_normal(n)
```

With the closed form in code, we can put it into the PDE directly. Both sides are evaluated on a grid at $t = 0.25$, 1 and 3, with ∂_t by a centered difference in time and the x-derivatives by `np.gradient`. Finite differences leave a residual even for an exact solution, so the cell uses two grids, the second twice as fine: the truncation error of these second-order stencils must fall by a factor of four.

```
def fp_residual(t, points, eps=1e-5):
    """Largest Fokker--Planck residual on a grid of `points`, and max|dp/dt|."""
    xs = np.linspace(-5.0, 5.0, points)
    dx = xs[1] - xs[0]
    dpdt = (p_t(xs, t + eps) - p_t(xs, t - eps)) / (2 * eps)
    p = p_t(xs, t)
    rhs = (np.gradient(theta * xs * p, dx)                # -d/dx(f p)
           + 0.5 * g**2 * np.gradient(np.gradient(p, dx), dx))
    res = (dpdt - rhs)[5:-5]            # trim the one-sided edge stencils
    return np.abs(res).max(), np.abs(dpdt).max()

for t in (0.25, 1.0, 3.0):
    (r1, dpdt), (r2, _) = fp_residual(t, 4001), fp_residual(t, 8001)
    print(f't={t}: max|dp/dt| = {dpdt:.1e}; max|residual| at 4001, 8001 points: '
          f'{r1:.1e}, {r2:.1e} (ratio {r1 / r2:.1f})')
```

```
t=0.25: max|dp/dt| = 4.4e-01; max|residual| at 4001, 8001 points: 4.4e-06, 1.1e-06 (ratio 4.0)
t=1.0: max|dp/dt| = 1.7e-01; max|residual| at 4001, 8001 points: 6.3e-07, 1.6e-07 (ratio 4.0)
t=3.0: max|dp/dt| = 3.2e-03; max|residual| at 4001, 8001 points: 1.2e-06, 3.0e-07 (ratio 4.0)
```

The residual is three to five orders of magnitude below $\partial_t p$ at each time (4×10^{-6} against 0.44 at $t = 0.25$; at $t = 3$ the density barely moves, and 10^{-6} stands against 3×10^{-3}), and the finer grid divides it by four at all three times. What is left is truncation error. The argument did not need this computation; the cell guards against a slip in the code on which all later experiments rely.

38.5.3 The Score Function

The smoothing term of the Fokker–Planck equation will turn out to be transport in disguise, and the velocity of that transport is built from one field, the gradient of $\log p_t$. The identity $\nabla p = p \, \nabla \log p$ puts it inside the Laplacian (Section 38.5.4). Every construction from there on needs this field, so we look at it first, with the mixture as the main example.

38.5.3.1 Definition and Geometry

The question is which direction, from a point $\mathbf{x}$, leads most quickly toward more probable points. The **score** of a density p_t answers it:

$$\mathbf{s}_t(\mathbf{x}) = \nabla_{\mathbf{x}} \log p_t(\mathbf{x}). \tag{38.55}$$

The score is a vector field over data space that points uphill on the log-density. The gradient is with respect to the input $\mathbf{x}$, not the parameters of a model; Section 37.5.1 introduced this data score and contrasted it with the parameter score of maximum likelihood.

The length of $\mathbf{s}_t$ is the rate at which $\log p_t$ increases, $\|\nabla p_t\|/p_t$. It is therefore large where the density is small but changing, in the tails and in the valleys between modes, and not where the density is large. The score vanishes at the critical points of p_t: modes, minima and saddles, which its derivative tells apart.

Take a Gaussian first. For $p(\mathbf{x}) = \mathcal{N}(\mathbf{x}; \mu, v\mathbf{I})$ the log-density is $-\|\mathbf{x}-\mu\|^2/(2v) + \text{const}$, so

$$\nabla \log p(\mathbf{x}) = -\frac{\mathbf{x}-\mu}{v}. \tag{38.56}$$

This is the linear spring of Section 37.5.1: it pulls toward the mean, and it is stiffer for smaller variance. With $\mu = 0$, $v = 0.25$ and $x = 1$ it is -4, four times the pull at unit variance. For a general covariance the spring is $-\Sigma^{-1}(\mathbf{x}-\mu)$. Figure 38.18 shows the score field of a two-mode density in two dimensions.

38.5.3.2 The Score of a Gaussian Mixture

Our data density is a mixture, so we need its score in a form we can evaluate and read. For $p(x) = \sum_k w_k\, N_k(x)$ with weights w_k and components $N_k(x) = \mathcal{N}(x; \mu_k, v_k)$, differentiate $\log p$ directly:

$$\nabla \log p(x) = \frac{\sum_k w_k \nabla N_k(x)}{\sum_j w_j N_j(x)} = \sum_k r_k(x) \left(-\frac{x-\mu_k}{v_k}\right), \qquad r_k(x) = \frac{w_k N_k(x)}{\sum_j w_j N_j(x)}. \tag{38.57}$$

The mixture's score is a weighted average of the component springs. The weight $r_k(x)$ is the *responsibility* of component k, the posterior probability that a point at x came from it, as in Section 37.5.1. For two components with equal weights, means $\pm\mu$ and a common variance v_0, the responsibilities reduce to a logistic function of x and the score has the closed form

$$s_0(x) = \frac{\mu \tanh(\mu x / v_0) - x}{v_0}. \tag{38.58}$$

Near each mode $\tanh \to \pm 1$ and the score is the local spring $-(x \mp \mu)/v_0$. For the running mixture, $\mu = 2$ and $v_0 = 0.25$, it is $s_0(x) = 8\tanh(8x) - 4x$. At $x = 0$ the score vanishes, the third critical point, and it grows from zero with slope

$$s_0'(0) = \frac{\mu^2 - v_0}{v_0^2} = 60.$$

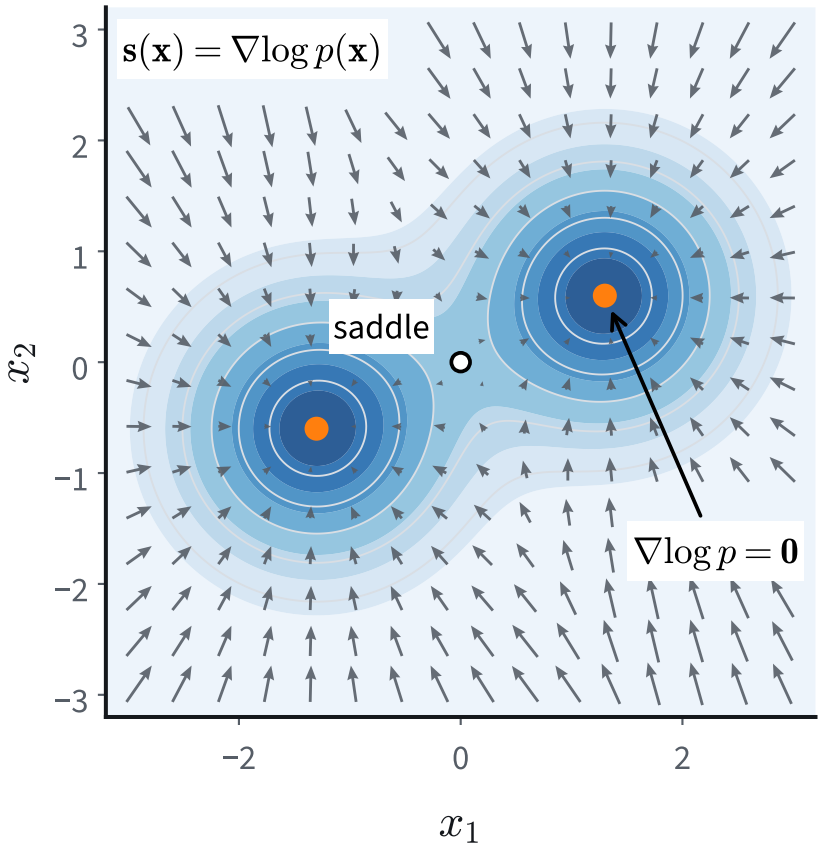


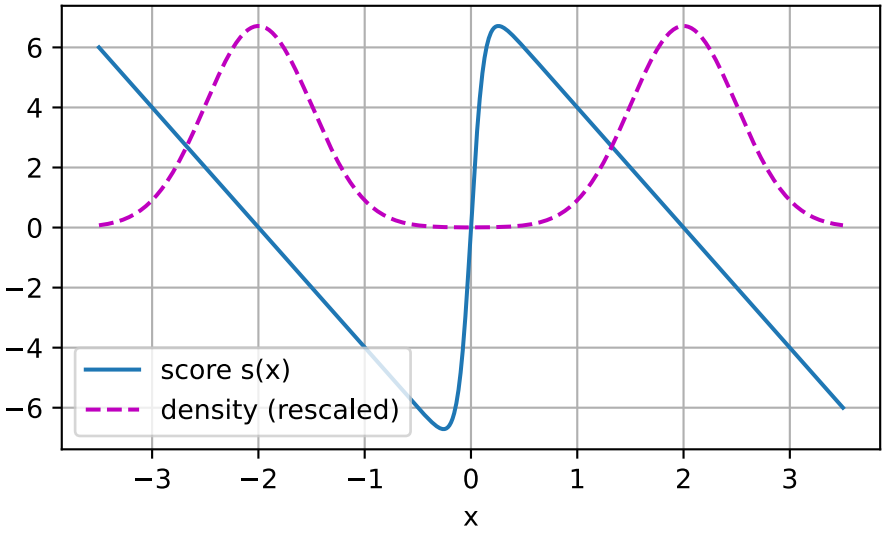
Figure 38.18: The score $\mathbf{s}(\mathbf{x}) = \nabla \log p(\mathbf{x})$ of a two-component Gaussian mixture, drawn over density contours. The arrows point uphill and vanish at critical points; the orange points mark the two modes, and the symmetry point between them is another zero, with different curvature. Arrows have the true length $\|\nabla p\|/p$, capped at 4 in the far tails, where it grows without bound; they are short near the modes and the saddle.

A positive slope means that $x = 0$ is a minimum of the density, a repeller of the uphill flow $\dot{x} = s_0(x)$. The mixture has two modes exactly when $\mu^2 > v_0$; for $\mu^2 \le v_0$ they merge into one (Exercise 4).

In code we use the general formula Equation 38.57, with a numerically stable softmax for the responsibilities, because the later experiments need the score of p_t at every time. The tanh form, the zeros and the slope 60 serve as checks on the implementation.

```
def score_t(x, t, ws=ws, mus=mus, v0s=v0s):
    """Exact score of p_t: responsibility-weighted component springs."""
    m, v = mixture_params(t, mus, v0s)
    logw = np.stack([np.log(w) - 0.5 * np.log(2 * np.pi * vk)
                     - (x - mk)**2 / (2 * vk) for w, mk, vk in zip(ws, m, v)])
    r = np.exp(logw - logw.max(axis=0))
    r /= r.sum(axis=0)                  # responsibilities, a stable softmax
    return sum(rk * -(x - mk) / vk for rk, mk, vk in zip(r, m, v))

gs = np.linspace(-3.5, 3.5, 1400)
mu, v0 = mus[1], v0s[1]
s = score_t(gs, 0.0)
s_tanh = (mu * np.tanh(mu * gs / v0) - gs) / v0
print(f'max|responsibility form - tanh form| = {np.abs(s - s_tanh).max():.1e}')
mid = (gs[:-1] + gs[1:]) / 2
print('score changes sign near', np.round(mid[s[:-1] * s[1:] < 0], 2) + 0.0)
delta = 1e-5                            # centered difference at x = 0
print(f'slope at 0: {(score_t(delta, 0.0) - score_t(-delta, 0.0)) / (2 * delta):.2f}, '
      f'formula (mu^2 - v0) / v0^2 = {(mu**2 - v0) / v0**2:.2f}')
dens = p_t(gs, 0.0)
d2l.plot(gs, [s, dens * np.abs(s).max() / dens.max()], 'x', '',
         legend=['score s(x)', 'density (rescaled)'], figsize=(5.5, 3))
```

```
max|responsibility form - tanh form| = 7.3e-15
score changes sign near [-2.  0.  2.]
slope at 0: 60.00, formula (mu^2 - v0) / v0^2 = 60.00
```



The checks pass: the two formulas agree to machine precision, and the zeros and the slope are where we computed them. What the plot adds is where the score is large: in the valley between the modes, where the density is nearly zero but its logarithm climbs steeply toward either mode, and in the tails.

38.5.4 The Continuity Equation and the Probability-Flow ODE

The Fokker–Planck equation describes particles that move at random. Which law governs particles that move deterministically? It turns out that the two laws coincide once the noise is written as a velocity.

38.5.4.1 Conservation of Probability

Forget noise for a moment. Suppose every particle follows a velocity field, $\dot{\mathbf{x}} = \mathbf{u}(\mathbf{x}, t)$. What PDE does their density q_t obey? Probability is neither created nor destroyed, so the mass inside a fixed region Ω can change only by flowing across its boundary. Particles cross a small patch of boundary at the rate density times velocity. We call this flux density the **probability current**, $\mathbf{j} = q\,\mathbf{u}$: at each point it gives the rate, per unit area, at which probability crosses a surface placed there. The mass balance of the region is

$$\frac{d}{dt}\int_{\Omega} q_t \, d\mathbf{x} = -\oint_{\partial\Omega} \mathbf{j}\cdot\mathbf{n}\, dS = -\int_{\Omega} \nabla\cdot(q_t\mathbf{u})\, d\mathbf{x}, \tag{38.59}$$

where the divergence theorem Equation 38.48 turned the boundary flux into a volume integral. The mass inside changes at minus the net outflow. Since Ω is arbitrary, the integrands agree pointwise, which gives the **continuity equation**:

$$\partial_t q_t + \nabla\cdot(q_t\,\mathbf{u}) = 0. \tag{38.60}$$

The density at a point falls at the rate at which the current diverges there. In one dimension the balance reads $\partial_t \int_a^b q\,dx = j(a) - j(b)$: what flows in at a minus what flows out at b.

Figure 38.19 draws it.

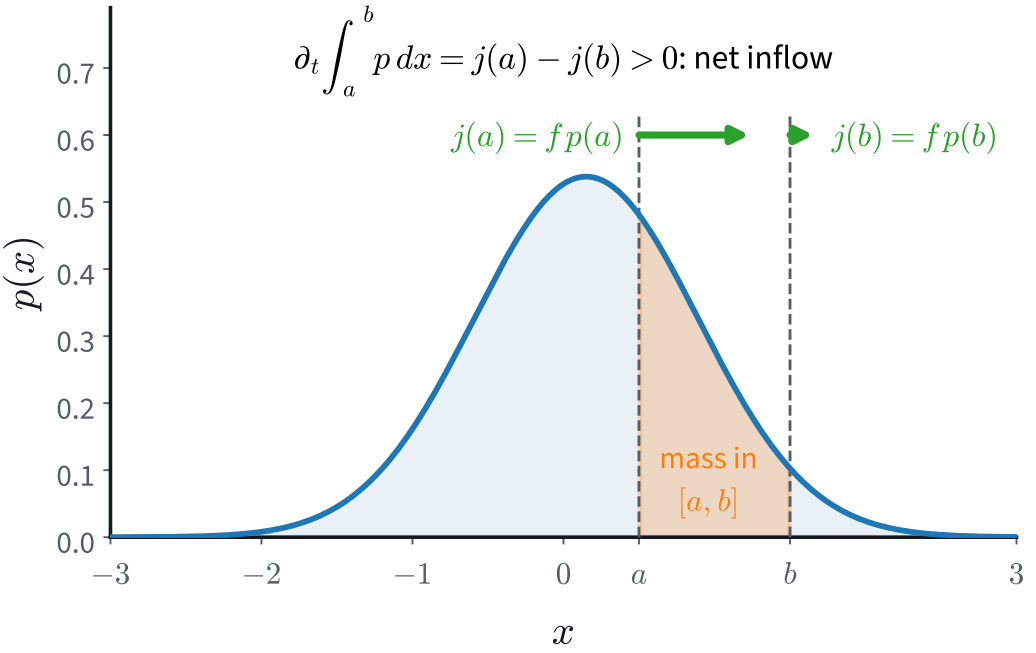


Figure 38.19: The continuity equation as a flux balance. The velocity is here a constant drift f and the density is written p: the density $p(x)$ (blue) is carried by f, giving the probability current $j(x) = f\,p(x)$. On the window from a to b on the right flank of the bump, the mass inside (orange) grows at the rate $j(a) - j(b)$, the net inflow: more current enters at a than leaves at b because the density is higher at a.

The balance also says what happens at the edges of the domain. On all of $\mathbb{R}^d$ we assume that the current decays at infinity. For the Fokker–Planck equation this means that $\mathbf{f}p_t$ and ∇p_t decay, which Gaussian tails ensure even though the OU drift grows linearly. Integrating Equation 38.60 over $\mathbb{R}^d$ then gives $\frac{d}{dt}\int q_t = 0$: total probability is conserved. On a bounded domain a *reflecting* boundary imposes $\mathbf{j} \cdot \mathbf{n} = 0$, so mass is conserved there too, while an *absorbing* boundary imposes $q = 0$, and mass can leave.

Two small cases show the equation at work in one dimension. For a translation, $u = c$, the equation is $\partial_t q + c\,\partial_x q = 0$, solved by $q_t(x) = q_0(x - ct)$: the profile slides without changing shape. For a contraction, $u = -x$, every particle moves as $x(t) = x(0)e^{-t}$, the cloud shrinks by the factor e^{-t}, and since the mass of each interval is preserved, the density carried along each trajectory rises by e^{t}.

The contraction suggests a general rule for the density seen by a moving particle. Expanding the divergence with the product rule,

$$\nabla \cdot (q\,\mathbf{u}) = \mathbf{u} \cdot \nabla q + q\,\nabla \cdot \mathbf{u},$$

the continuity equation becomes $\partial_t q + \mathbf{u} \cdot \nabla q = -q\,\nabla \cdot \mathbf{u}$. The left side is the rate of change of q along a trajectory $\mathbf{x}(t)$ of $\dot{\mathbf{x}} = \mathbf{u}$, by the chain rule. Dividing by q,

$$\frac{d}{dt}\log q_t(\mathbf{x}(t)) = -\nabla\cdot\mathbf{u}(\mathbf{x}(t),t). \tag{38.61}$$

Along a trajectory the log-density falls at the rate of the local expansion. For the contraction, $\nabla\cdot u=-1$ and the log-density rises at rate 1, as computed above. The trajectories of $\dot{\mathbf{x}}=\mathbf{u}$ are called the **characteristics** of the continuity equation. The rule is the instantaneous change of variables of Section 38.3.3, proved there along trajectories.

38.5.4.2 Diffusion as Transport

The Fokker–Planck equation Equation 38.52 is not yet a continuity equation: next to the transport term $-\nabla\cdot(\mathbf{f}p)$ it has a Laplacian. The chain rule turns the Laplacian into a transport term. It is a one-line calculation, and the most consequential one of the section. Wherever $p>0$,

$$\nabla p = p\,\nabla\log p, \tag{38.62}$$

which is the derivative of $\log p$ written backward, and therefore

$$\tfrac{1}{2}g^2\,\Delta p = \tfrac{1}{2}g^2\,\nabla\cdot(\nabla p) = \nabla\cdot\big(p\cdot\tfrac{1}{2}g^2\nabla\log p\big). \tag{38.63}$$

The diffusion term is the divergence of a current, the density times $+\frac{1}{2}g^2\nabla\log p$. Moved to the left side of a continuity equation it appears with a minus sign, as a velocity $-\frac{1}{2}g^2\nabla\log p$.

Proposition (Fokker–Planck is a continuity equation). *Wherever $p_t>0$, the Fokker–Planck equation Equation 38.52 is equivalent to*

$$\partial_t p_t + \nabla\cdot(p_t\,\mathbf{u}_t)=0, \qquad \mathbf{u}_t(\mathbf{x}) = \mathbf{f}(\mathbf{x},t) - \tfrac{1}{2}g(t)^2\,\nabla\log p_t(\mathbf{x}). \tag{38.64}$$

Proof. Substitute Equation 38.63 into Equation 38.52 and merge the two divergences:

$$\begin{aligned}\partial_t p &= -\nabla\cdot(\mathbf{f}p) + \nabla\cdot\big(p\cdot\tfrac{1}{2}g^2\nabla\log p\big)\\ &= -\nabla\cdot\big(p\,[\mathbf{f} - \tfrac{1}{2}g^2\nabla\log p]\big). \qquad \blacksquare\end{aligned}$$

At the level of densities, the SDE moves probability as if each particle followed the velocity $\mathbf{u}_t$: the drift plus a push down the log-density gradient. The corresponding current is

$$\mathbf{j} = \mathbf{f}\,p_t - \tfrac{1}{2}g^2\,\nabla p_t = p_t\,\mathbf{u}_t. \tag{38.65}$$

The heat equation shows what this velocity looks like. Take $\mathbf{f}=0$ and $p_t=\mathcal{N}(0,v_t)$ in one dimension with $v_t=v_0+g^2t$. The score is $-x/v_t$, so

$$u_t(x) = -\tfrac{1}{2}g^2\Big(-\frac{x}{v_t}\Big) = \frac{g^2}{2v_t}\,x.$$

The velocity points away from the center and is proportional to the distance from it, and it weakens as the density widens. The ODE $\dot{x} = g^2 x/(2v_t)$ separates: $\frac{d}{dt}\log x = \frac{g^2}{2v_t} = \frac{1}{2}\frac{d}{dt}\log v_t$, so

$$x(t) = x(0)\sqrt{v_t/v_0}. \tag{38.66}$$

Every particle is stretched by the same factor, and a Gaussian stretched by $\sqrt{v_t/v_0}$ has variance v_t, which is the variance that Brownian motion produces. With $v_0 = g = 1$ and $t = 3$ every particle doubles its distance from the center. A Brownian particle started at $x = 1$ instead lands at 1 plus a draw of standard deviation $\sqrt{3} \approx 1.73$. The two processes agree on the histogram at every time but move individual particles differently (Figure 38.20).

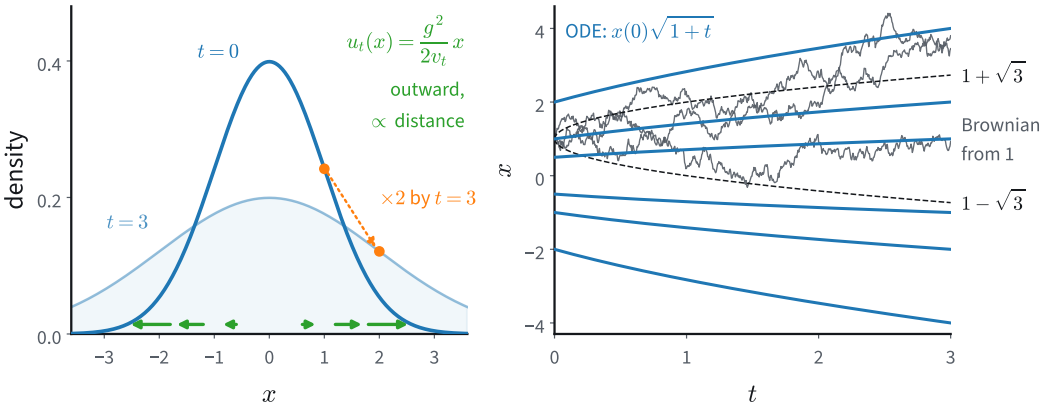


Figure 38.20: The heat equation $\partial_t p = \frac{1}{2}\partial_{xx} p$ from $p_0 = \mathcal{N}(0,1)$, solved as a flow. (a) The density at $t = 0$ and at $t = 3$, with the velocity $u_0(x) = x/2$ drawn as arrows on the axis: outward and proportional to distance. The particle at $x = 1$ moves to $x = 2$ by $t = 3$, and the density it carries halves. (b) Flow trajectories $x(0)\sqrt{1+t}$, smooth and never crossing, against three Brownian paths from $x = 1$ and their one-standard-deviation envelope $1 \pm \sqrt{t}$.

There is a limit to this construction. As $v_0 \to 0$ the velocity at small times is $g^2 x/(2g^2 t) = x/(2t)$, which is unbounded as $t \to 0$. A deterministic flow can spread a point mass into a Gaussian only through a field that blows up at the start, because a Lipschitz flow maps one point to one point.

The next cell runs this example. From one draw of 20,000 points of $\mathcal{N}(0,1)$ it forms a Brownian cloud at $t = 3$ (adding an independent $\mathcal{N}(0,3)$ increment) and a flow cloud by integrating $\dot{x} = x/(2(1+t))$ with Heun's method (Section 38.2). It prints the two standard deviations, and how far the flow cloud is from the starting sample doubled.

```
def heat_velocity(x, t, v0=1.0, g=1.0):  # u_t = -(1/2) g^2 score = g^2 x / (2 v_t)
    return g**2 * x / (2 * (v0 + g**2 * t))

n, T, steps = 20_000, 3.0, 600
dt = T / steps
z = rng.standard_normal(n)                # p_0 = N(0, 1)
```

```
x_bm = z + np.sqrt(T) * rng.standard_normal(n)   # Brownian motion to time T
x_ode = z.copy()
for k in range(steps):                  # Heun steps for the probability flow
    k1 = heat_velocity(x_ode, k * dt)
    k2 = heat_velocity(x_ode + dt * k1, (k + 1) * dt)
    x_ode += 0.5 * dt * (k1 + k2)
print(f'std at t={T}: Brownian {x_bm.std():.3f}, flow {x_ode.std():.3f}, '
      f'exact {np.sqrt(1 + T):.3f}')
print(f'max|x_ode - 2 z| = {np.abs(x_ode - 2 * z).max():.1e}')
```

```
std at t=3.0: Brownian 1.995, flow 1.994, exact 2.000
max|x_ode - 2 z| = 5.8e-06
```

Both clouds have standard deviation 2.00 up to a sampling error of order 0.01, and the flow map equals $x \mapsto 2x$ up to a Heun error below 10^{-5}. Random kicks and a deterministic stretch have produced the same histogram.

38.5.4.3 The Probability-Flow ODE

The heat equation was one example. The proposition above says that the same construction works for every SDE of our form: following the velocity $\mathbf{u}_t$ gives an ODE,

$$\frac{d\mathbf{x}}{dt} = \mathbf{u}_t(\mathbf{x}) = \mathbf{f}(\mathbf{x}, t) - \tfrac{1}{2} g(t)^2 \, \nabla \log p_t(\mathbf{x}), \tag{38.67}$$

the **probability-flow ODE** (Y. Song, Sohl-Dickstein, et al. 2021). The next proposition is the first of the two results that generative models take from this section.

Proposition (the deterministic twin). *Let p_t be the marginal density of $d\mathbf{X} = \mathbf{f}\,dt + g\,d\mathbf{W}$, $\mathbf{X}_0 \sim p_0$, assumed smooth and positive, so that it solves the Fokker–Planck equation. Suppose that $\mathbf{u}_t$ is continuous in t, continuously differentiable in $\mathbf{x}$, and Lipschitz in $\mathbf{x}$ with a constant that does not depend on t. Let $\mathbf{x}(t)$ solve Equation 38.67 with $\mathbf{x}(0) \sim p_0$. Then $\mathbf{x}(t) \sim p_t$ for every t: the ODE and the SDE share all time-marginals.*

Proof. By the existence and uniqueness theorem of Section 38.1.1.3, a field that is Lipschitz with a constant independent of t has exactly one trajectory through every point, forward and backward, so its time-t map is a bijection of $\mathbb{R}^d$. Let q_t be the density of the ODE's state. There is no circle here: the field $\mathbf{u}_t$ is built from p, not from q, so it is a known velocity and q_t obeys the continuity equation $\partial_t q + \nabla \cdot (q\mathbf{u}_t) = 0$. By the previous proposition p_t solves the same equation, and both start from $q_0 = p_0$. The derivation of the log-density rule Equation 38.61 used only this PDE, so along each characteristic $\log q_t$ and $\log p_t$ change at the same rate $-\nabla \cdot \mathbf{u}_t$, and they start equal. Since the characteristics reach every point, $q_t = p_t$ everywhere. ■

The Lipschitz hypothesis holds for the OU drift and for the score of a Gaussian or a Gaussian mixture with positive component variances, whose slope is bounded for t in $[0, T]$.

The marginals agree; the paths do not. An SDE path is jagged and crosses other paths. An ODE trajectory is smooth, and two trajectories never meet at the same point at the same time, by uniqueness (panel (b) of Figure 38.20).

The ODE also gives a likelihood. Integrating the log-density rule Equation 38.61 with $q_t = p_t$ along a trajectory from 0 to T gives

$$\log p_0(\mathbf{x}(0)) = \log p_T(\mathbf{x}(T)) + \int_0^T \nabla \cdot \mathbf{u}_t(\mathbf{x}(t))\, dt. \tag{38.68}$$

The log-density of a data point is the log-density of its image at time T, which is nearly Gaussian and easy to evaluate, plus the accumulated expansion along the way. With a learned score in place of the exact one this is, up to solver error, the log-density of the *model* defined by that field (Y. Song, Sohl-Dickstein, et al. 2021). It is one of the two numbers by which the likelihood of a diffusion model is reported; the other is a variational bound (Section 17.6).

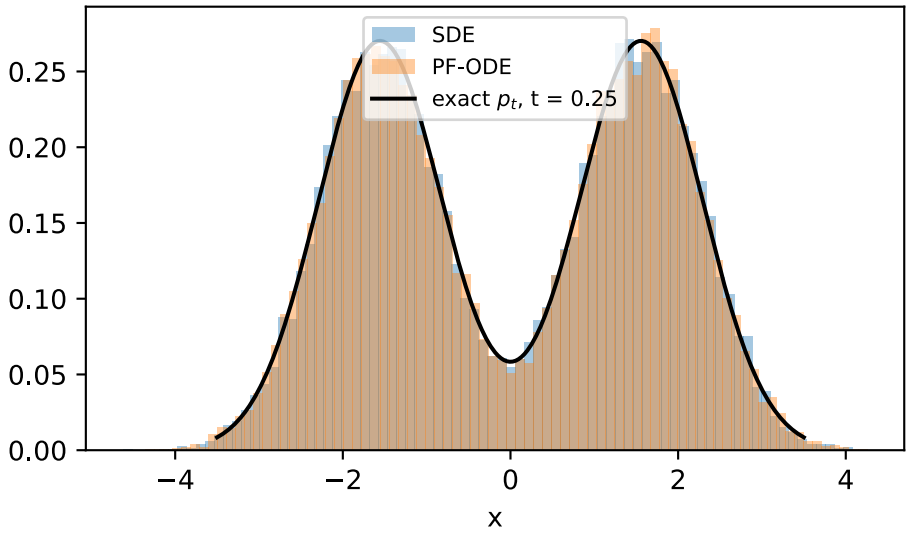
38.5.4.4 SDE and ODE Marginals Compared

The heat flow was a stretch of one Gaussian. On the mixture the proposition has more to do: the density changes shape, from two modes to one, and the jagged and the smooth process must still agree on it at every time. From one sample of p_0 we launch two clouds of $20{,}000$ particles: one follows the SDE with Euler–Maruyama steps, the other the probability-flow ODE with Heun steps and the exact score. As a yardstick we compute the one-sample Kolmogorov–Smirnov distance of each cloud to the exact CDF of p_t, the largest gap between the empirical and the exact CDF. For an exact sample of size n this distance exceeds $1.36/\sqrt{n}$ only 5% of the time. Before looking at the output, guess which cloud's distance changes more from one snapshot to the next.

```
def pf_velocity(x, t, score=score_t, theta=theta, g=g):
    return -theta * x - 0.5 * g**2 * score(x, t)   # u_t = f - (1/2) g^2 s_t

def ks_exact(sample, t, **mixture):
    """One-sample Kolmogorov--Smirnov distance to the exact p_t."""
    xs_sorted, n = np.sort(sample), len(sample)
    F = mixture_cdf(xs_sorted, t, **mixture)
    return max((np.arange(1, n + 1) / n - F).max(), (F - np.arange(n) / n).max())

n, T, steps = 20_000, 3.0, 600
dt = T / steps
x_sde = sample_p0(n, rng)
x_ode = x_sde.copy()                    # both clouds start from one sample
snaps = {}
for k in range(steps):
    t = k * dt
    x_sde += -theta * x_sde * dt + g * np.sqrt(dt) * rng.standard_normal(n)
    k1 = pf_velocity(x_ode, t)          # Heun steps for the ODE
    k2 = pf_velocity(x_ode + dt * k1, t + dt)
    x_ode += 0.5 * dt * (k1 + k2)
    if k + 1 in (50, 200, 600):         # t = 0.25, 1, 3
        snaps[(k + 1) * dt] = (x_sde.copy(), x_ode.copy())
print(f'95% band for n = {n}: {1.36 / np.sqrt(n):.4f}')
for t, (a, b) in snaps.items():
    print(f't={t:.2f}: KS(SDE) = {ks_exact(a, t):.4f}, KS(ODE) = {ks_exact(b, t):.4f}')
gx = np.linspace(-3.5, 3.5, 400)
d2l.set_figsize((5.5, 3))
d2l.plt.hist(snaps[0.25][0], bins=80, density=True, alpha=0.4, label='SDE')
d2l.plt.hist(snaps[0.25][1], bins=80, density=True, alpha=0.4, label='PF-ODE')
```

```
d2l.plt.plot(gx, p_t(gx, 0.25), 'k', label='exact $p_t$, t = 0.25')
d2l.plt.xlabel('x'); d2l.plt.legend(loc='upper center', fontsize=8);
```

```
95% band for n = 20000: 0.0096
t=0.25: KS(SDE) = 0.0051, KS(ODE) = 0.0074
t=1.00: KS(SDE) = 0.0088, KS(ODE) = 0.0074
t=3.00: KS(SDE) = 0.0044, KS(ODE) = 0.0074
```



In most runs both clouds stay within the 0.0096 band at all three times, with distances between about 0.003 and 0.01. A value just above the band is no alarm: the band is a 5% threshold, and an exact sample crosses it one time in twenty. At $t = 0.25$ the marginal is still bimodal, and both histograms reproduce its two modes; by $t = 1$ it has a single mode (the merging time is 0.78, Exercise 4).

Now the guess. The ODE's distance is the same at all three times, to every digit printed. A strictly increasing map that carries p_0 to p_t carries the sample's empirical CDF along with it, so the ODE cloud keeps the distance its starting sample had and adds only solver error. The SDE cloud is redrawn by the noise at every step, and its distance fluctuates.

38.5.5 Time Reversal

The forward SDE turns data into noise. Generation needs the process played backward, from noise to data. This may sound impossible: the heat equation cannot be solved backward from an arbitrary final density, because smoothing destroys detail that nothing can restore. We do not solve it backward. The forward family p_t is given, and we ask only for a process that visits it in reverse order (Figure 38.21); its drift will contain $\nabla \log p_t$, which carries the destroyed detail. The probability-flow ODE is one such process, integrated backward. For the SDE the answer is less obvious: the backward process is again a diffusion, and its drift carries twice the score correction of the ODE.

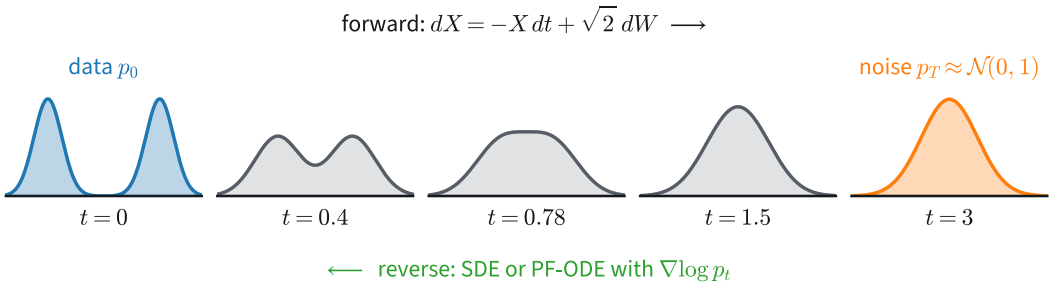


Figure 38.21: The forward marginals of the running mixture $p_0 = \frac{1}{2}\mathcal{N}(-2, 0.25) + \frac{1}{2}\mathcal{N}(2, 0.25)$ under OU with $\theta = 1$, $g = \sqrt{2}$, exact at every time. The reverse processes of this section run the same marginals from right to left. The flat top at $t \approx 0.78$ is where the two modes merge.

38.5.5.1 Bayes' Rule on an Infinitesimal Step

To reverse one small step we use Bayes' rule to ask where a particle came from. Over a short interval τ the forward step is the Euler–Maruyama Gaussian (Section 38.4.4.1):

$$p(\mathbf{x}_{t+\tau} \mid \mathbf{x}_t) \approx \mathcal{N}\big(\mathbf{x}_{t+\tau};\ \mathbf{x}_t + \mathbf{f}(\mathbf{x}_t, t)\,\tau,\ g^2\tau\,\mathbf{I}\big).$$

Now condition on the endpoint $\mathbf{x}_{t+\tau} = \mathbf{y}$. Bayes' rule gives the probable starting points as the forward kernel reweighted by the marginal at time t:

$$p(\mathbf{x}_t = \mathbf{x} \mid \mathbf{x}_{t+\tau} = \mathbf{y}) \propto p(\mathbf{y} \mid \mathbf{x})\ p_t(\mathbf{x}).$$

A starting point is likely if it could have reached $\mathbf{y}$ and if it was likely to begin with. Take logarithms and expand $\log p_t(\mathbf{x})$ around the endpoint: $\log p_t(\mathbf{x}) \approx \log p_t(\mathbf{y}) + (\mathbf{x} - \mathbf{y}) \cdot \nabla \log p_t(\mathbf{y})$. The step $\mathbf{x} - \mathbf{y}$ is of size $\sqrt{\tau}$, so the neglected quadratic term is of order τ and changes the posterior only at higher order. To the same order we may evaluate $\mathbf{f}$ at $\mathbf{y}$ instead of $\mathbf{x}$, since $\mathbf{f}(\mathbf{x}, t) - \mathbf{f}(\mathbf{y}, t)$ is of size $\sqrt{\tau}$ and is multiplied by τ. Now a linear term added to a Gaussian exponent shifts the mean by the variance times the slope and leaves the variance alone (complete the square). The exponent is $-\|\mathbf{x} - (\mathbf{y} - \mathbf{f}(\mathbf{y}, t)\,\tau)\|^2/(2g^2\tau)$, the variance is $g^2\tau$ and the slope is $\nabla \log p_t(\mathbf{y})$:

$$p(\mathbf{x} \mid \mathbf{y}) \approx \mathcal{N}\big(\mathbf{x};\ \mathbf{y} - [\mathbf{f}(\mathbf{y}, t) - g^2 \nabla \log p_t(\mathbf{y})]\,\tau,\ g^2\tau\,\mathbf{I}\big). \tag{38.69}$$

A backward step is a Gaussian kick of the same variance $g^2\tau$, whose drift is the forward drift corrected by the full g^2 times the score. The marginal p_t acts as a prior that bends the reversed step toward where the data density was.

38.5.5.2 The Reverse-Time SDE

The one-step calculation is a heuristic: it expands to first order and starts from an approximate kernel. Anderson (1982) proved that it gives the right answer for the whole process. We state his theorem without proof: the time reversal of the diffusion $d\mathbf{X} = \mathbf{f}\,dt + g\,d\mathbf{W}$ is the **reverse-time SDE**

$$d\mathbf{X} = \big[\mathbf{f}(\mathbf{X}, t) - g(t)^2\, \nabla_{\mathbf{x}} \log p_t(\mathbf{X})\big]\, dt + g(t)\, d\bar{\mathbf{W}}, \tag{38.70}$$

integrated from $t = T$ down to $t = 0$. Here $\bar{\mathbf{W}}$ is a Brownian motion that runs backward in time; in a simulation it means a fresh Gaussian increment at each backward step. Started from $\mathbf{X}_T \sim p_T$, the process has marginal p_t at every intermediate time, and at $t = 0$ it delivers the data density. The theorem identifies the reversed process as a whole; below we prove only the statement about marginals, which is the part that sampling needs.

Read as steps, the display is the calculation of the previous subsection. Going from $t + \tau$ down to t is a step with $dt = -\tau$, so the particle moves by $-[\mathbf{f} - g^2 \nabla \log p_t]\, \tau$ plus a kick of variance $g^2\tau$, and this is the Gaussian Equation 38.69. A clock that runs forward is easier to compute with. In the **reversed clock** $\bar{t} = T - t$ the same process is an ordinary SDE with drift $-\mathbf{f} + g^2\, \nabla \log p_t$ and noise g: the drift changes sign with the direction of time, and the noise does not care.

The smallest case can be checked by hand. Take OU at stationarity, $p_t = \mathcal{N}(0, 1)$ for all t, with $\theta = 1$, $g = \sqrt{2}$ and score $-x$. The probability-flow velocity is $-x - \frac{1}{2} \cdot 2 \cdot (-x) = 0$: the ODE particles do not move, so the cloud stays $\mathcal{N}(0, 1)$. In the reversed clock the reverse SDE has drift $-f + g^2 s_t = x - 2x = -x$ and noise $\sqrt{2}$, which is the same OU process. Stationary OU is its own reverse.

With $\frac{1}{2} g^2$ in place of g^2 the reverse drift would be $x - x = 0$, and the cloud would spread like Brownian motion instead of staying put. So the reverse SDE needs the full factor g^2, twice the correction of the ODE. A family of samplers explains the factor.

Proposition (one family, marginals fixed). *Let p_t be the marginal density of the forward SDE, under the hypotheses of the deterministic-twin proposition and with $\mathbf{f}$ Lipschitz in $\mathbf{x}$. For any $\lambda \geq 0$, the reverse-time SDE with noise amplitude λg and drift*

$$\mathbf{b}_\lambda = \mathbf{f} - \tfrac{1+\lambda^2}{2}\, g^2\, \nabla \log p_t \tag{38.71}$$

has marginals p_t when started from p_T. The choice $\lambda = 1$ is Anderson's reverse SDE Equation 38.70; the choice $\lambda = 0$ is the probability-flow ODE Equation 38.67.

The more noise a sampler injects, the larger its score correction.

Proof. Work in the reversed clock $\bar{t} = T - t$ and let $q_{\bar{t}} = p_{T - \bar{t}}$, the density evolution played in reverse, so that $\partial_{\bar{t}} q = -\partial_t p = \nabla \cdot (\mathbf{f} q) - \frac{1}{2} g^2 \Delta q$, with $\mathbf{f}$ and g evaluated at forward time $T - \bar{t}$. A process $d\mathbf{Y} = \mathbf{b}'\, d\bar{t} + \lambda g\, d\mathbf{W}$ in the reversed clock has its own Fokker–Planck equation $\partial_{\bar{t}} q = -\nabla \cdot (q\, \mathbf{b}') + \frac{\lambda^2}{2} g^2 \Delta q$. For the two to agree we need

$$-\nabla \cdot (q\, \mathbf{b}') = \nabla \cdot (\mathbf{f} q) - \tfrac{1+\lambda^2}{2} g^2 \Delta q = \nabla \cdot \Big(q \Big[\mathbf{f} - \tfrac{1+\lambda^2}{2} g^2 \nabla \log q \Big] \Big),$$

where *both* Laplacians were converted to transport form with Equation 38.63. The equation holds for $\mathbf{b}' = -\mathbf{f} + \frac{1+\lambda^2}{2} g^2 \nabla \log q$. A drift $\mathbf{b}'$ in the reversed clock is a drift $-\mathbf{b}'$ in forward time, which gives Equation 38.71. It remains to pass from equal evolution equations and the equal start $q_0 = p_T$ to equal marginals, and this is the step that needs uniqueness. For $\lambda = 0$ it is the argument along characteristics of the deterministic twin. For $\lambda > 0$ it is the uniqueness stated after Equation 38.52, which we did not prove; it applies because $\mathbf{b}'$ is Lipschitz when $\mathbf{f}$ and $\mathbf{u}_t$ are. ■

The proof also explains the factor of two. A reverse process with noise λg smooths its own density, so its drift must undo two smoothings. One $\frac{1}{2}g^2\nabla \log p_t$ cancels the forward smoothing; that is the whole correction of the probability-flow ODE, which injects no noise. A further $\frac{\lambda^2}{2}g^2\nabla \log p_t$ compensates the smoothing by the reverse process's own noise. Anderson's $\lambda = 1$ contains both halves. The marginals do not determine the drift, though: any other drift with the same marginals differs from $\mathbf{b}'$ by a field $\mathbf{w}$ with $\nabla \cdot (q\mathbf{w}) = 0$, and Exercise 7 builds such fields.

What is the freedom good for? Noise lets a sampler forget. An error in the distribution of the state, from a start that is not quite p_T or from an earlier step, is carried by the ODE all the way to $t = 0$: its map is a bijection and loses nothing. The noise of an SDE washes part of the error out (Exercise 10 makes both statements exact). In exchange the score term is larger, and a long noisy step adds error of its own. Diffusion-model samplers choose λ with this in mind (Section 17.5). Whatever the choice, the sampler needs the same field $\nabla \log p_t$, so one estimate of the score serves the whole family.

Figure 38.22 shows three values of λ on the running mixture. The deterministic paths split toward the two modes near $\bar{t} \approx 2.2$, that is, near forward time 0.78, where the marginal stops being bimodal.

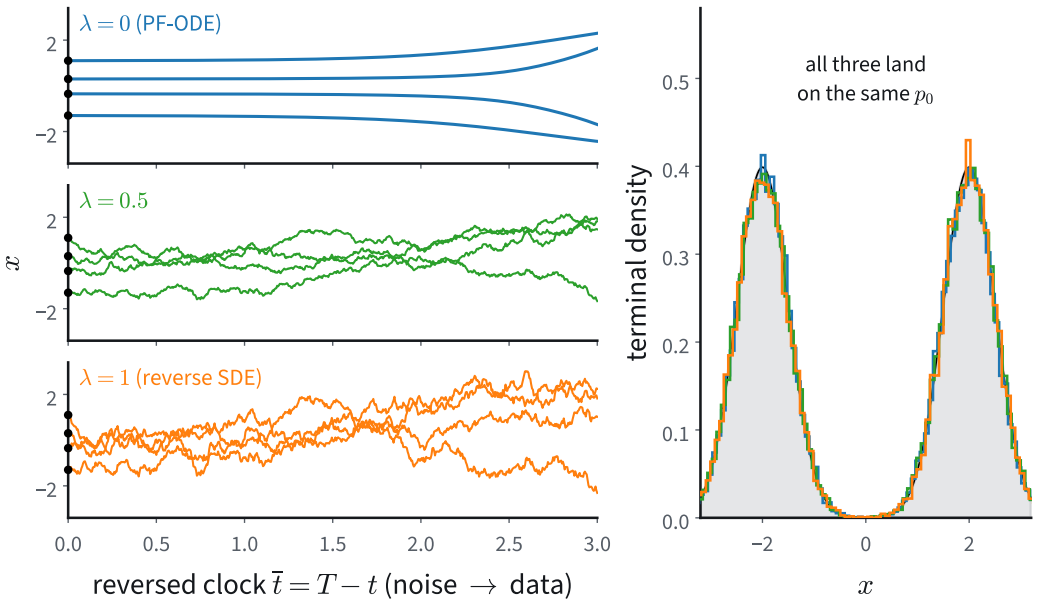


Figure 38.22: Reverse-time trajectories from the same four starting points at $t = T$, using the exact score of the running mixture with drift $\mathbf{f} - \frac{1+\lambda^2}{2}g^2\nabla \log p_t$ and noise λg. The probability-flow ODE ($\lambda = 0$) is deterministic; $\lambda = 0.5$ and Anderson's reverse SDE ($\lambda = 1$) add noise and a larger score correction. The three histograms at $t = 0$, of $20{,}000$ particles each (right), agree with the data density p_0.

38.5.5.3 Reverse-Time Sampling

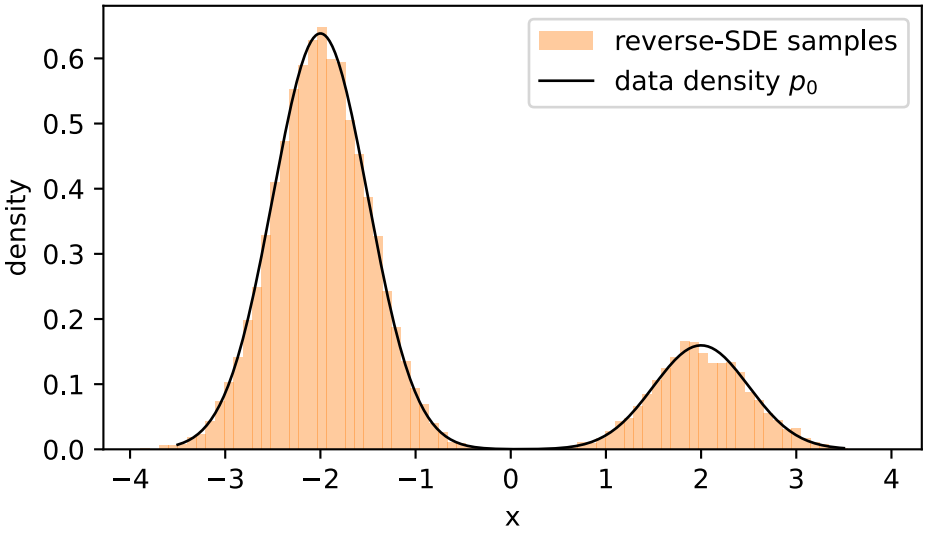
Let us generate data, with the exact score in place of a trained network. For this one experiment we make the data lopsided. The modes keep their places and widths, but four fifths

of the mass go to the left: $p_0 = 0.8\,\mathcal{N}(-2, 0.25) + 0.2\,\mathcal{N}(2, 0.25)$. A sampler that starts from symmetric noise must then get the proportions right as well as the modes, and that is a test it can fail.

We draw $20{,}000$ points from $\mathcal{N}(0, 1)$, which stands in for p_T at $T = 3$, and integrate Anderson's reverse SDE down to $t = 0$ with Euler–Maruyama steps in the reversed clock. The cell first measures how far p_T is from $\mathcal{N}(0, 1)$, then compares the sample with the data: the mass to the left of the origin, the means and widths of the modes, and the KS distance to p_0 beside that of an exact sample of the same size.

```
uneven = dict(ws=np.array([0.8, 0.2]))  # four fifths of the data on the left
pT = p_t(gs, 3.0, **uneven)
print(f'max|p_T - N(0,1)| at T=3: {np.abs(pT - gauss(gs, 0.0, 1.0)).max():.4f} '
      f'(peak {pT.max():.2f})')
x = rng.standard_normal(n)              # N(0, 1) stands in for p_T
for k in range(steps):                  # reversed clock: t_bar = k dt
    t = T - k * dt                      # forward time
    x += (theta * x + g**2 * score_t(x, t, **uneven)) * dt \
         + g * np.sqrt(dt) * rng.standard_normal(n)
left, right = x[x < 0], x[x >= 0]
print(f'mass on the left: {len(left) / n:.3f} (data 0.800)')
print(f'means {left.mean():+.3f}, {right.mean():+.3f} (data -2, +2)')
print(f'widths {left.std():.3f}, {right.std():.3f} (data 0.5)')
x_ref = sample_p0(n, rng, **uneven)     # an exact sample of the same size
print(f'KS to the exact p_0: sampler {ks_exact(x, 0.0, **uneven):.4f}, '
      f'exact sample {ks_exact(x_ref, 0.0, **uneven):.4f}')
d2l.set_figsize((5.5, 3))
d2l.plt.hist(x, bins=80, density=True, alpha=0.4, color='C1',
             label='reverse-SDE samples')
d2l.plt.plot(gs, p_t(gs, 0.0, **uneven), 'k', lw=1, label='data density $p_0$')
d2l.plt.xlabel('x'); d2l.plt.ylabel('density'); d2l.plt.legend();
```

```
max|p_T - N(0,1)| at T=3: 0.0145 (peak 0.40)
mass on the left: 0.800 (data 0.800)
means -1.993, +1.998 (data -2, +2)
widths 0.504, 0.509 (data 0.5)
KS to the exact p_0: sampler 0.0068, exact sample 0.0054
```



The sampler recovers the data. Its modes sit within about 0.02 of ± 2, their widths are close to 0.5, and four fifths of the particles end on the left, to within a hundredth. This last number is the one to look at. Half of the particles started to the right of the origin, and nothing in the sampler mentions the weights except the score. Where does the score keep them? Not near $t = 0$. There the responsibilities in Equation 38.57 are 0 or 1 wherever the density is not negligible, and the score is one of the two springs $-(x \pm 2)/0.25$, whatever the weights. This is the blindness of the score to the weights of distant modes that Section 37.5.1 found. Not at $t = 3$ either. There the two components are almost the same Gaussian, and the responsibility of the left one is close to its weight, 0.8, wherever a particle sits. But the two springs nearly coincide as well: they differ by $2m_t/v_t$, for component means $\mp m_t$ and variance v_t, which is 0.2 at $t = 3$ against 16 at $t = 0$. Averaging two almost equal springs with the weights 0.8 and 0.2 gives a small steady push to the left and decides no particle's mode.

The weights act in between, while the modes separate. The two components are equally responsible at the point $x_t^* = v_t \ln(0.8/0.2)/(2m_t)$. A particle to the left of this point more probably came from the left component, and under the reverse process it will more probably end there. At $t = 3$ the watershed lies at $x^* \approx 7$, far to the right of every particle. At $t = 1$ it has come in to 0.85, so a particle at $x = +0.5$ is still more likely to end on the left than on the right. At $t = 0$ it sits at 0.09, in the valley, and by then the valley is empty. At high noise the weights are everywhere and decide nothing; at low noise they would decide, and no particle is there.

The start was not quite right, and it hardly mattered. With unequal weights p_T has not forgotten the data altogether: its mean is $-1.2\, e^{-3} \approx -0.06$, and the first line shows a gap of up to 0.0145 to $\mathcal{N}(0, 1)$, against a peak of 0.40. The noise of the reverse SDE washes most of this mismatch out: it costs about 0.002 of the left mode's mass, less than the binomial noise 0.003 of the printed fraction. The probability-flow ODE, which forgets nothing, would lose several times as much (Exercise 10).

The sampler is not exact, and the KS line gives no more than a hint of it. Both distances are of the order of the 0.0096 band. The sampler's is usually the larger, and it lands above the band in about one run in six, where an exact sample does so in one run in twenty. A single run

of this size cannot tell such a sampler from an exact one. Besides the start there is a second source of error, the discretization. Euler–Maruyama has weak order one (Section 38.4.4.1), so its output carries a bias proportional to the step. The bias sits in the widths, but one run cannot measure it there: the Monte Carlo error of a width estimated from M samples is the width divided by $\sqrt{2M}$, about 0.003 for the left mode with its $16{,}000$ samples and 0.006 for the right mode with $4{,}000$.

The bias can be isolated without sampling. For Gaussian data $\mathcal{N}(2, 0.25)$, the right mode alone, the marginal is a Gaussian $\mathcal{N}(m_t, v_t)$ with score $-(x - m_t)/v_t$, so every reverse step is affine in x, and the mean m and the variance V of the sampler's state follow an exact recursion. With $\kappa_t = \theta - g^2/v_t$ the slope of the reverse drift, it reads $m \leftarrow m + \Delta t\,(\kappa_t m + g^2 m_t/v_t)$ and $V \leftarrow (1 + \kappa_t \Delta t)^2 V + g^2 \Delta t$. Started from the exact p_T, the recursion has no error but the discretization. The cell evaluates it on uniform grids of 300 to $2{,}400$ steps, and then spends $1{,}500$ steps in two ways: uniformly, and with $1{,}000$ of them on $t \le 0.5$.

```
def em_reverse_width(ts, mu=2.0, v0=0.25):
    """Exact output width of reverse Euler--Maruyama on data N(mu, v0),
    stepping down the time grid ts = (0, t_1, ..., T) from T to 0."""
    mu, v0 = np.array([mu]), np.array([v0])
    (m,), (V,) = mixture_params(ts[-1], mu, v0)     # start from the exact p_T
    for t, dt in zip(ts[:0:-1], np.diff(ts)[::-1]):
        (mt,), (vt,) = mixture_params(t, mu, v0)
        kappa = theta - g**2 / vt       # slope of the reverse drift in x
        m, V = (m + dt * (kappa * m + g**2 * mt / vt),
                (1 + kappa * dt)**2 * V + g**2 * dt)
    return np.sqrt(V)

grids = {f'{N} uniform steps': np.linspace(0, T, N + 1)
         for N in (300, 600, 1200, 2400, 1500)}
grids['1500 steps, 1000 on t <= 0.5'] = np.concatenate(
    [np.linspace(0, 0.5, 1001), np.linspace(0.5, T, 501)[1:]])
for name, ts in grids.items():
    print(f'{name:>28}: width excess {em_reverse_width(ts) - 0.5:.4f}')
```

```
         300 uniform steps: width excess 0.0096
         600 uniform steps: width excess 0.0047
        1200 uniform steps: width excess 0.0024
        2400 uniform steps: width excess 0.0012
        1500 uniform steps: width excess 0.0019
1500 steps, 1000 on t <= 0.5: width excess 0.0006
```

On the first four grids the width excess over 0.5 is 0.0096, 0.0047, 0.0024 and 0.0012: it halves when the step halves, as a first-order bias must. At 600 steps, the grid of the sampling run, it is 0.005. This is about the excess to expect in the printed widths, and it is the size of their Monte Carlo errors: one sampling run cannot resolve it, and the recursion can.

The last two lines ask where the steps should go. Spending $1{,}500$ steps uniformly removes six tenths of the 600-step bias. Making the steps ten times finer on $t \le 0.5$ only, also $1{,}500$ steps in all, removes nearly nine tenths. A tenfold refinement divides the bias from the refined part by ten, so the drop from 0.0047 to 0.0006 says that almost all of the output's bias comes from the steps on $t \le 0.5$, the last sixth of the reverse run.

Two things combine. The reverse drift is steepest there, with slope $\theta - g^2/v_0 = 1 - 8 = -7$

at $t = 0$ against -1 near $t = T$, so each step errs more. And an error made early in the reverse run is contracted by every later step, through the factor $(1 + \kappa_t \Delta t)^2 < 1$ of the recursion, while an error made in the last steps reaches the output as it is. For narrower data the slope $\theta - g^2/v_0$ grows without bound, and the step then meets the stability limit of Section 38.2.3; here we are far from it. Exercise 9 measures both effects and looks for the best placement.

All of this rested on a score we could write down. The data were a two-component mixture, and the only errors left were Euler–Maruyama's bias and the small gap between p_T and $\mathcal{N}(0, 1)$. Real data offer no such formula, and computing p_t from the Fokker–Planck equation is out of the question: a grid with 100 cells per axis has 100^d unknowns, and d counts pixels. The score has to be learned (Section 38.6). For images we have samples of p_0 and nothing else.

38.5.6 Summary

The density of an SDE $d\mathbf{X} = \mathbf{f}\,dt + g\,d\mathbf{W}$ obeys the Fokker–Planck equation $\partial_t p = -\nabla\cdot(\mathbf{f}p) + \frac{1}{2}g^2\Delta p$: the drift transports the density and the noise smooths it. Because $\nabla p = p\nabla \log p$, the smoothing is itself transport, with velocity $-\frac{1}{2}g^2\nabla \log p$, so the same density is moved by an ordinary differential equation, the probability-flow ODE $\dot{\mathbf{x}} = \mathbf{f} - \frac{1}{2}g^2\nabla \log p_t$. This ODE has the SDE's marginals at every time, smooth trajectories that never cross, and a likelihood. The same densities can be visited in reverse order, by a family of processes with noise λg and drift $\mathbf{f} - \frac{1+\lambda^2}{2}g^2\nabla \log p_t$. One half of the score correction undoes the forward smoothing and the rest compensates the sampler's own noise, which is why Anderson's reverse SDE ($\lambda = 1$) carries twice the correction of the ODE ($\lambda = 0$). We proved this for the marginals, given uniqueness for the Fokker–Planck equation; Anderson's theorem for the whole process is cited. Driven by the exact score, the reverse SDE turned symmetric noise into a mixture with weights 0.8 and 0.2, modes and proportions alike; the proportions are decided at intermediate noise, while the modes separate. In all of these samplers the only unknown is the score $\nabla \log p_t$.

38.5.7 Exercises

1. **Laplacian as averaging.** Averaging a smooth ψ over the sphere of radius r centered at $\mathbf{x}$ gives $\psi(\mathbf{x}) + \frac{r^2}{2d}\Delta\psi(\mathbf{x}) + O(r^4)$, as stated in Section 38.5.2.1.

    1. Let $\mathbf{e}$ be uniformly distributed on the unit sphere of $\mathbb{R}^d$. Show by symmetry that $\mathbb{E}[e_i] = 0$, $\mathbb{E}[e_i e_j] = \delta_{ij}/d$ and $\mathbb{E}[e_i e_j e_k] = 0$. Use these moments in a Taylor expansion of $\psi(\mathbf{x} + r\mathbf{e})$ to prove the averaging formula. Why does no r^3 term appear?
    2. In one dimension the sphere is the pair of points $x \pm \delta$. Show that one forward-Euler step of the heat equation $\partial_t p = D\,\partial_{xx} p$, with $\partial_{xx} p$ replaced by the centered difference $(p_{i+1} - 2p_i + p_{i-1})/\delta^2$, sets each new p_i to a weighted combination of p_{i-1}, p_i and p_{i+1} whose weights sum to one. Find the largest step Δt for which all three weights are nonnegative. Why can the iteration then never create a new maximum?

2. **Moments from the weak form.** The evolution law Equation 38.51 was derived for test functions that vanish outside a bounded set. Assume that it also holds for polynomial ϕ, which the Gaussian tails of this section's densities justify.

1. For the one-dimensional OU process, insert $\phi(x) = x$ and $\phi(x) = x^2$ and derive ODEs for the mean $m(t)$ and the variance $v(t)$ of X_t. Show that they are the moment ODEs Equation 38.54 for every initial law p_0, Gaussian or not. Check this against `mixture_params` by computing the mean and the variance of the two-mode mixture at time t.
2. For the linear SDE $d\mathbf{X} = \mathbf{A}\mathbf{X}\,dt + g\,d\mathbf{W}$ in $\mathbb{R}^d$ with a constant matrix $\mathbf{A}$, use $\phi(\mathbf{x}) = x_i$ and $\phi(\mathbf{x}) = x_i x_j$ to show that the mean and the covariance satisfy $\dot{\mathbf{m}} = \mathbf{A}\mathbf{m}$ and $\dot{\Sigma} = \mathbf{A}\Sigma + \Sigma\mathbf{A}^\top + g^2\mathbf{I}$. Which matrix equation must a stationary covariance solve?

3. **Stationary density from zero current.**
   1. Set $\partial_t p = 0$ in the OU Fokker–Planck equation, $0 = \theta\,\partial_x(xp) + \frac{1}{2}g^2\partial_{xx}p$. Integrate once in x and argue from decay at infinity that the constant of integration, which is minus the probability current Equation 38.65, vanishes. Solve the resulting first-order ODE and show that $p_\infty = \mathcal{N}(0, g^2/(2\theta))$ without assuming a Gaussian form.
   2. Repeat the argument for a general drift $f(x)$ with constant g, and express p_∞ through a potential U with $f = -U'$. Conversely, given a density $p \propto e^{-E(x)}$, find the drift that makes p stationary and write it through the score of p.
   3. Apply the result to the double-well drift $f(x) = x - x^3$ with $g = \sqrt{2}$ and find the modes of p_∞. What goes wrong for the unstable drift $f(x) = \theta x$ with $\theta > 0$?

4. **Mode merging.** Consider the symmetric two-component mixture with the score Equation 38.58.
   1. Derive Equation 38.58 from the responsibility formula Equation 38.57.
   2. Show that the score has three zeros when $\mu^2 > v_0$ and only the zero $x = 0$ when $\mu^2 \le v_0$. Classify each zero as a local maximum or a local minimum of the density. (Hint: the slope at 0 is $(\mu^2 - v_0)/v_0^2$.)
   3. Under the OU process the mixture stays symmetric, with component means $\pm\mu e^{-\theta t}$ and the component variance computed by `mixture_params`. Find the time t^* at which p_t stops being bimodal for the section's parameters, $\mu = 2$, $v_0 = 0.25$, $\theta = 1$ and $g = \sqrt{2}$.
   4. What does the change at t^* mean for a reverse-time sampler that starts from noise? (Biroli et al. (2024) analyze the high-dimensional counterpart of this symmetry breaking and call it speciation.)

5. code **State-dependent noise.** ● For an SDE $d\mathbf{X} = \mathbf{f}\,dt + \mathbf{G}(\mathbf{X}, t)\,d\mathbf{W}$ with a matrix noise, write $\mathbf{a} = \mathbf{G}\mathbf{G}^\top$. Itô's correction is then $\frac{1}{2}\sum_{ij} a_{ij}\,\partial_i\partial_j\phi$, and two integrations by parts, $\int p\,a_{ij}\,\partial_i\partial_j\phi = \int \phi\,\partial_i\partial_j(a_{ij}p)$, give $\partial_t p = -\nabla\cdot\mathbf{j}$ with the current

$$j_i = f_i\,p - \tfrac{1}{2}\sum_k \partial_k(a_{ik}\,p)$$

and, dividing by p, the probability-flow velocity

$$\mathbf{u} = \mathbf{f} - \tfrac{1}{2}\,\mathbf{a}\,\nabla\log p - \tfrac{1}{2}\,\nabla\cdot\mathbf{a}, \qquad (\nabla\cdot\mathbf{a})_i = \sum_k \partial_k a_{ik}.$$

A constant matrix only replaces g^2 by $\mathbf{a}$ in the score term; a state-dependent $\mathbf{a}$ adds the last term. Now let $dX = g(X)\,dW$ with no drift and a smooth positive g, on the circle $[0, 1)$ with periodic boundaries.

1. Specialize the matrix current and the general velocity above to this SDE. Show that a stationary density has constant current, that periodicity forces the constant to vanish, and find p_∞.
2. Replacing the constant g^2 of Equation 38.52 by $g(x)^2$ gives the PDE $\partial_t p = \frac{1}{2} g(x)^2\, \partial_{xx} p$. Which stationary density does this PDE predict?
3. Simulate $50{,}000$ particles started from the uniform distribution with Euler–Maruyama, $g(x) = 1 + \frac{1}{2}\sin(2\pi x)$ and $\Delta t = 10^{-3}$ up to $t = 2$, wrapping positions modulo 1. Compare the histogram with both predictions. Where do the particles accumulate, and why?

6. code **Fokker–Planck on a grid.** ● Divide $[-6, 6]$ into N cells of width $\delta = 12/N$ with centers x_i, and let $p_i(t)$ approximate the density in cell i. A conservative discretization of the OU equation assigns to each interior face $x_{i+1/2}$ the current

$$j_{i+1/2} = f(x_{i+1/2})\,\frac{p_i + p_{i+1}}{2} - \frac{g^2}{2}\,\frac{p_{i+1} - p_i}{\delta}, \tag{38.72}$$

sets the current through the two outer faces to zero, and evolves $\dot p_i = -(j_{i+1/2} - j_{i-1/2})/\delta$. This is a linear system $\dot{\mathbf{p}} = \mathbf{F}\mathbf{p}$.

1. Show that $\sum_i p_i\,\delta$ is conserved for every $\mathbf{p}$, the discrete counterpart of the divergence theorem Equation 38.48, and say which boundary condition of Section 38.5.4 the zero outer currents impose.
2. Build $\mathbf{F}$ for $\theta = 1$, $g = \sqrt{2}$ and $\delta = 0.05$. Start from the two-mode mixture `p_t(x, 0.0)` on the cell centers and integrate to $t = 0.5$ with forward Euler at step $\Delta t = 10^{-3}$ and with backward Euler at step $\Delta t = 10^{-2}$ (one `np.linalg.solve` per step). Compare both results with `p_t(x, 0.5)`.
3. Increase the forward-Euler step from 10^{-3} and find the largest step that stays stable on a run to $t = 5$, long enough for an unstable mode seeded by rounding error to grow. Relate this step to the most negative eigenvalue $-\nu$ of $\mathbf{F}$ through the bound $\Delta t < 2/\nu$ of Section 38.2.3. How does the limit change when δ is halved? Compare it with the condition of Exercise 1b.
4. A grid with 100 cells per axis in d dimensions has 100^d unknowns. Evaluate this count for $d = 2$ and for $d = 784$, the pixel count of a 28×28 image. What does it imply for how diffusion models use the Fokker–Planck equation?

Adapted from Särkkä and Solin (2019), Exercise 5.3.

7. **Divergence-free velocity fields.** ● Let $\mathbf{R}$ be a constant antisymmetric $d \times d$ matrix.

1. Show that $\nabla\cdot(p\,\mathbf{R}\nabla \log p) = 0$ for every smooth positive density p. Conclude that for every $\beta \in \mathbb{R}$ the ODE $\dot{\mathbf{x}} = \mathbf{u}_t(\mathbf{x}) + \beta\mathbf{R}\nabla \log p_t(\mathbf{x})$, with $\mathbf{u}_t$ from Equation 38.64, has the same marginals as the probability-flow ODE.
2. Take the two-dimensional OU process at stationarity, $p_t = \mathcal{N}(\mathbf{0}, \frac{g^2}{2\theta}\mathbf{I})$ for all t, and $\mathbf{R} = \left(\begin{smallmatrix} 0 & -1 \\ 1 & 0 \end{smallmatrix}\right)$. Show that the probability-flow velocity vanishes, and describe the trajectories of the modified ODE.
3. Now put the rotation into the SDE itself: $d\mathbf{X} = (-\theta\mathbf{I} + \beta\mathbf{R})\mathbf{X}\,dt + g\,d\mathbf{W}$ in $\mathbb{R}^2$. Use the covariance equation of Exercise 2b to show that $\frac{g^2}{2\theta}\mathbf{I}$ is a stationary covariance

for every β, so that the SDE has the same stationary density as in (b). Compute its stationary current, the current Equation 38.65 of this SDE. Why does the argument of Exercise 3a, which forces a stationary current to vanish, fail in two dimensions?

4. Suppose that $\mathbf{f} = -\nabla U$, so that $\mathbf{u}_t$ is a gradient field. Among all velocity fields $\mathbf{w}$ whose continuity equation Equation 38.60 is solved by p_t, show that $\mathbf{u}_t$ has the smallest kinetic energy $\mathbb{E}_{p_t}\|\mathbf{w}\|^2$. The Benamou–Brenier formulation (Benamou and Brenier 2000) minimizes the same energy over the path p_t as well.

8. **Exact Gaussian reverse step.** ● Let $p_t = \mathcal{N}(m_t, v_t)$ be a Gaussian OU marginal. Over a step τ, the exact forward transition is $X_{t+\tau} \mid X_t = x \sim \mathcal{N}(\alpha_\tau x, \sigma_\tau^2)$ with $\alpha_\tau = e^{-\theta\tau}$ and $\sigma_\tau^2 = \frac{g^2}{2\theta}(1 - e^{-2\theta\tau})$, by the kernel Equation 38.46.

   1. Use Gaussian conditioning to show that

$$X_t \mid X_{t+\tau} = y \;\sim\; \mathcal{N}\left(\frac{\sigma_\tau^2\, m_t + \alpha_\tau v_t\, y}{v_{t+\tau}},\; \frac{\sigma_\tau^2\, v_t}{v_{t+\tau}}\right). \tag{38.73}$$

   Here $v_{t+\tau} = \alpha_\tau^2 v_t + \sigma_\tau^2$ is the marginal variance at time $t + \tau$.

   2. Expand the mean and the variance to first order in τ, and recover the reverse step Equation 38.69, including the full factor g^2 in front of the score.

9. code **Placing the steps.** ● The function `em_reverse_width` returns the exact output width of reverse Euler–Maruyama on the data $\mathcal{N}(2, 0.25)$ for any time grid. The text compared two grids of $1{,}500$ steps.

   1. Start from the uniform grid of 600 steps and make the steps ten times finer on $t \le c$ only, for $c \in \{0.1, 0.25, 0.5, 1, 2\}$. Record the number of steps and the width excess. A tenfold refinement divides the bias from the refined part by ten. Use this to compute the share of the 600-step bias that comes from the steps on $t \le c$. How many extra steps on $t \le 0.1$ do the work of the 900 extra uniform steps of the text? The text names two reasons why these steps dominate. Measure each on the uniform grid: for the single step that leaves $t \in \{0.1, 0.5, 3\}$, compute the error in V that it commits when it starts from the exact v_t, and the factor by which the later steps multiply this error before $t = 0$.
   2. With a budget of $N = 1{,}500$ steps, try the grids $t_i = T\,(i/N)^\rho$ for $\rho \in \{1, 1.5, 2, 3, 4, 6\}$. Which ρ gives the smallest excess? Why does a large ρ do worse again?
   3. Is the bias on the squared grid, $\rho = 2$, still of first order in $1/N$? Check with $N = 750$, $1{,}500$ and $3{,}000$. What has the grid changed, the order or the constant?
   4. Repeat (b) for narrower data, $v_0 = 0.01$, where the slope of the reverse drift at $t = 0$ is $1 - 2/0.01 = -199$, and compare each excess with the target width 0.1. How does the best ρ move? What does this suggest for data whose variance is small in some directions? (Section 17.5 crowds the steps of image samplers toward low noise with a power-law grid in the noise level.)

10. [extended] **Prior mismatch, ODE versus SDE.** ●● Samplers start from the reference $q_T = \mathcal{N}(0, 1)$ instead of the exact p_T. This project compares how the reverse SDE and the reverse probability-flow ODE carry that mismatch to $t = 0$. Use the unequal weights $w = (0.8, 0.2)$ of the reverse sampler in Section 38.5.5, which make p_T differ

visibly from q_T for small T: pass `**uneven` to `p_t`, `score_t`, `mixture_cdf` and `sample_p0`, and `score=lambda x, t: score_t(x, t, **uneven)` to `pf_velocity`.

1. Run Anderson's reverse SDE as in the text and the reverse-time probability-flow ODE with Heun steps, both from q_T, for $T \in \{0.25, 0.5, 1, 2, 3\}$ with step $\Delta t = 0.005$. Record the fraction of samples with $x < 0$, whose target is 0.8. The SDE at $T = 3$ is the run of the text.
2. Anderson's theorem holds for the whole process: started at $X_T = y$, the reverse SDE produces $X_0 \sim p(x_0 \mid x_T = y)$. Use this to predict the SDE's left-mode mass as an expectation over $y \sim q_T$ of a responsibility of p_T (Equation 38.57). For the ODE, use that its map from x_T to x_0 is increasing and carries p_T to p_0, and predict the left-mode mass from the standard normal CDF and the CDFs of p_T and p_0. Evaluate both predictions by quadrature and compare them with the measurements.
3. Let q_0 be the law of a sampler's output. Use the data-processing inequality of Section 37.3.3 to prove that the ODE output satisfies $D_{\mathrm{KL}}(q_0 \| p_0) = D_{\mathrm{KL}}(q_T \| p_T)$, while the SDE output satisfies $D_{\mathrm{KL}}(q_0 \| p_0) \le D_{\mathrm{KL}}(q_T \| p_T)$.
4. Compute $D_{\mathrm{KL}}(q_T \| p_T)$ and the SDE's $D_{\mathrm{KL}}(q_0 \| p_0)$ by quadrature for each T. The posterior $p(x_0 \mid x_T = y)$ is a two-component Gaussian mixture whose components follow from Exercise 8a, so q_0 is a one-dimensional integral over y. By what factor does the SDE shrink the mismatch at $T = 1$?
5. Repeat the measurement of (a) at $T = 1$ for the samplers of the family Equation 38.71 with $\lambda \in \{0, 0.5, 1, 2\}$. Which sampler would you use when the forward horizon T must be short? What limits how large λ can be at a fixed step size?

38.6 Score Matching and Diffusion Models

We know how to turn Gaussian noise into data: integrate the reverse-time SDE or the probability-flow ODE. Both ask for one thing in return, the score $\nabla \log p_t$ of the noised data at every point they visit. For the two-mode mixture of the previous section a formula supplied it. Images come with samples and no formula. Here the samples are 4,096 draws from that mixture, standing in for a training set of images. We have to learn a field that we cannot evaluate at even a single point.

Fitting a density by maximum likelihood and differentiating it does not help, because a flexible density, the exponential of a network's output, needs a normalizing constant, an integral over all of space. Aapo Hyvärinen observed in 2005 that the score needs no normalizer, and that an integration by parts removes the unknown data score from the objective (Hyvärinen 2005). Pascal Vincent noticed something simpler still (Vincent 2011). The score of noised data is the average, over the clean points that could have produced a noisy point, of the direction back to the clean point. That direction is known for every training pair, so ordinary least squares learns the score from samples alone.

Diffusion models are trained this way: their standard loss, "predict the noise you added", is this regression with a particular weighting. Their two best-known samplers, DDPM and DDIM, bring no new equations either: each discretizes one we already have. A one-dimensional Gaussian turns each claim into a number we can check by hand before the mixture tests it.

No network is trained in this section. We learn the score at one noise level by least squares,

with a model small enough to be solved exactly. The samplers are then analyzed on the exact score of the mixture, so that each error we measure is the sampler's own: first at a single noise level, which turns out not to be enough, then at all levels at once. Once, near the end, a sampler runs on scores fitted by least squares at every level. Networks and their errors are the business of Chapter 17, which builds on three results from here: Vincent's theorem, the identification of DDPM with the variance-preserving SDE of Section 38.4, and that of DDIM with Euler's method. The proof of Hyvärinen's identity and the subsection on guidance can be skipped on a first reading.

38.6.1 Learning the Score

We want a vector field $\mathbf{s}_\theta$, in practice a network with d inputs and d outputs, that approximates the score of the data, and we can only draw samples. The first task is an objective that compares the model with the true field and that we can evaluate from samples.

38.6.1.1 A Normalizer-Free Objective

A neural network defines a density most flexibly through an energy: an *energy-based model* is $p_\theta(\mathbf{x}) = e^{-E_\theta(\mathbf{x})}/Z_\theta$ with a network E_θ and the normalizer $Z_\theta = \int e^{-E_\theta(\mathbf{x})}\, d\mathbf{x}$ (LeCun et al. 2006). Maximum likelihood needs $\nabla_\theta \log Z_\theta$, a d-dimensional integral that changes with every update of θ. The score in $\mathbf{x}$ does not see it: Z_θ is a constant in $\mathbf{x}$, so $\nabla_{\mathbf{x}} \log p_\theta = -\nabla_{\mathbf{x}} E_\theta$.

So we compare fields instead of densities. With $\nabla \log p$ the score of the data, the **explicit score-matching** objective is the mean squared distance between the two fields, weighted by where the data are:

$$J_{\text{ESM}}(\theta) = \tfrac{1}{2}\, \mathbb{E}_{\mathbf{x}\sim p}\left[\, \left\| \mathbf{s}_\theta(\mathbf{x}) - \nabla \log p(\mathbf{x}) \right\|^2 \right]. \tag{38.74}$$

It is zero exactly when the model field equals the data score wherever the data have mass. When $\mathbf{s}_\theta$ is the score of a density q, this is the Fisher divergence of Section 37.5.1, which also discusses what such a divergence can and cannot see. Here $\mathbf{s}_\theta$ may be any vector field, because a network with d outputs need not be the gradient of anything. The objective has removed the normalizer, but it still contains $\nabla \log p$, the one field we do not know. Hyvärinen's integration by parts removes it.

Proposition (Hyvärinen's identity). *Let p be a smooth positive density and $\mathbf{s}_\theta$ a smooth vector field, with $p(\mathbf{x})\, \mathbf{s}_\theta(\mathbf{x}) \to \mathbf{0}$ as $\|\mathbf{x}\| \to \infty$ and all the expectations below finite. Then*

$$J_{\text{ESM}}(\theta) = \mathbb{E}_{\mathbf{x}\sim p}\left[\, \tfrac{1}{2}\|\mathbf{s}_\theta(\mathbf{x})\|^2 + \nabla \cdot \mathbf{s}_\theta(\mathbf{x}) \right] + C, \tag{38.75}$$

where $\nabla \cdot \mathbf{s}_\theta = \sum_i \partial s_i/\partial x_i$ is the divergence and $C = \tfrac{1}{2}\mathbb{E}_p\|\nabla \log p\|^2$ does not depend on θ.

Proof. Expand the square in Equation 38.74. The term $\tfrac{1}{2}\mathbb{E}\|\nabla \log p\|^2$ is the constant C, the term $\tfrac{1}{2}\mathbb{E}\|\mathbf{s}_\theta\|^2$ appears unchanged in Equation 38.75, and it remains to transform the cross term. In one dimension,

$$-\mathbb{E}_{x\sim p}\left[s_\theta(x)\, (\log p)'(x) \right] = -\int s_\theta(x)\, \frac{p'(x)}{p(x)}\, p(x)\, dx = -\int s_\theta(x)\, p'(x)\, dx,$$

and integrating by parts with the boundary term $[s_\theta\, p]_{-\infty}^{\infty} = 0$ leaves $+\int s'_\theta(x)\, p(x)\, dx = \mathbb{E}_p[s'_\theta]$. This is Stein's identity $\mathbb{E}_p[s'_\theta + s_\theta\, (\log p)'] = 0$ of Section 37.5.2: the derivative has moved from the unknown density onto the model. In d dimensions, apply the same step to each coordinate i, with s_i in place of s_θ and $\partial_i p$ in place of p', integrating over x_i first; Fubini's theorem allows this under the stated integrability. Summed over i, the cross term becomes $\mathbb{E}_p[\sum_i \partial_i s_i] = \mathbb{E}_p[\nabla \cdot \mathbf{s}_\theta]$. ■

Every term on the right of Equation 38.75 is an expectation under p of something we can evaluate for a given model, so a sample average estimates it, with no Z_θ and no $\nabla \log p$. The right-hand side is called **implicit score matching**.

A Gaussian shows that the identity works. Let the data be $x \sim \mathcal{N}(0, v)$, whose score is $-x/v$, and take the one-parameter model $s_w(x) = -x/w$. Then $\frac{1}{2}s_w^2 + s'_w = x^2/(2w^2) - 1/w$, so the implicit objective is $J(w) = v/(2w^2) - 1/w$. Its derivative $-v/w^3 + 1/w^2$ vanishes at $w = v$: the objective never saw the data score and still returns the variance. The constant agrees as well. At the minimum $J(v) = -1/(2v)$, and $C = \frac{1}{2}\mathbb{E}[x^2/v^2] = 1/(2v)$, so the explicit objective is $J(v) + C = 0$ there, as it must be when the model matches the data.

In practice the expectation is a sample average. Let us replace v in J by the mean of x^2 over 10,000 draws from $\mathcal{N}(0, 2)$ and minimize over a grid of w. The minimizer is then that sample mean, whose standard error is $v\sqrt{2/10^4} \approx 0.03$, so we should see 2 give or take a few hundredths.

```
rng = np.random.default_rng()
v = 2.0
x = rng.normal(0.0, np.sqrt(v), 10000)        # data x ~ N(0, v)
w = np.linspace(0.5, 4.0, 701)                # model s_w(x) = -x / w
J_ism = (x**2).mean() / (2 * w**2) - 1 / w    # E[s_w^2 / 2 + s_w']
print(f'implicit score matching picks w = {w[J_ism.argmin()]:.2f}  (v = {v})')
```

```
implicit score matching picks w = 2.03  (v = 2.0)
```

We do. The two terms of Equation 38.75 pull against each other, and the Gaussian shows how. The divergence term $-1/w$ rewards a field that points inward at the samples, a sink where the data are, and rewards it more the steeper the sink. The squared norm $x^2/(2w^2)$ stops the field from growing without bound. The balance sits at the true variance.

Implicit score matching has two limits. The proof needs the boundary term to vanish. For data on a bounded interval it does not, and the implicit objective then picks the wrong field. The second limit is cost. The divergence is the trace of the Jacobian, $\nabla \cdot \mathbf{s}_\theta = \operatorname{tr}(\partial \mathbf{s}_\theta / \partial \mathbf{x})$. Computing it exactly takes one reverse-mode pass per row of the Jacobian, d passes in all, and d is in the millions for images. Hutchinson's estimator (Section 38.3.3) replaces the trace by one vector-Jacobian product per random direction, but it adds variance. Denoising score matching avoids the trace altogether.

38.6.1.2 Denoising Score Matching

We want an objective without the trace, and we want the score where a sampler will need it, which includes points near but not on the data. Matching the score of *noised* data meets both requirements. Perturb each sample with Gaussian noise of scale σ:

$$\tilde{\mathbf{x}} = \mathbf{x} + \sigma\epsilon, \qquad \epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I}),$$
$$p_\sigma(\tilde{\mathbf{x}} \mid \mathbf{x}) = \mathcal{N}(\tilde{\mathbf{x}};\, \mathbf{x},\, \sigma^2\mathbf{I}).$$

The noised marginal $p_\sigma(\tilde{\mathbf{x}}) = \int p_\sigma(\tilde{\mathbf{x}} \mid \mathbf{x})\, p(\mathbf{x})\, d\mathbf{x}$ is the data density blurred by a Gaussian, and it approaches p as σ shrinks. Its score is as unknown as the data score. The score of the *conditional*, however, follows from the logarithm of a Gaussian density:

$$\nabla_{\tilde{\mathbf{x}}} \log p_\sigma(\tilde{\mathbf{x}} \mid \mathbf{x}) = \frac{\mathbf{x} - \tilde{\mathbf{x}}}{\sigma^2} = -\frac{\epsilon}{\sigma}. \tag{38.76}$$

This vector points from the noisy observation back to the clean sample that produced it, and we know it for every pair we generate. **Denoising score matching** (DSM) uses it as a regression target:

$$J_{\text{DSM}}(\theta) = \tfrac{1}{2}\, \mathbb{E}_{\mathbf{x}\sim p,\, \tilde{\mathbf{x}}\sim p_\sigma(\cdot\mid\mathbf{x})} \Big[\big\| \mathbf{s}_\theta(\tilde{\mathbf{x}}) - \nabla_{\tilde{\mathbf{x}}} \log p_\sigma(\tilde{\mathbf{x}} \mid \mathbf{x}) \big\|^2 \Big]. \tag{38.77}$$

The loss asks the network to predict, from the noisy point alone, the direction back to a clean point that it cannot see. The target depends on the clean point, while the score we want depends only on the noisy one. Least squares connects the two, because it fits the target's conditional mean. The following lemma states that fact. It is elementary, and this section and the next both rest on it; the next applies it to a velocity.

Lemma (regression to the conditional mean). *Let (X, Y) be jointly distributed with $\mathbb{E}\|Y\|^2 < \infty$, and let $\mathbf{m}(X) = \mathbb{E}[Y \mid X]$. Then for every measurable $\mathbf{v}$ with $\mathbb{E}\|\mathbf{v}(X)\|^2 < \infty$,*

$$\mathbb{E}\,\|\mathbf{v}(X) - Y\|^2 = \mathbb{E}\,\|\mathbf{v}(X) - \mathbf{m}(X)\|^2 + \mathbb{E}\,\|Y - \mathbf{m}(X)\|^2. \tag{38.78}$$

The second term does not involve $\mathbf{v}$: minimizing a least-squares loss against the noisy target Y is the same problem as minimizing it against the conditional mean $\mathbf{m}(X)$, up to an additive constant.

Proof. Insert $\pm\mathbf{m}(X)$ and expand. The cross term is $2\,\mathbb{E}\left[(\mathbf{v}(X) - \mathbf{m}(X))^\top (\mathbf{m}(X) - Y)\right]$. By the tower rule $\mathbb{E}[Z] = \mathbb{E}[\mathbb{E}[Z \mid X]]$ (Section 36.2) we may first average over Y with X fixed, and $\mathbb{E}[\mathbf{m}(X) - Y \mid X] = \mathbf{0}$ makes the cross term vanish. ■

Proposition (Vincent's theorem). *Let the noising kernel $p_\sigma(\tilde{\mathbf{x}} \mid \mathbf{x})$ be differentiable in $\tilde{\mathbf{x}}$, with gradient and integral over $\mathbf{x}$ interchangeable (both hold for the Gaussian kernel), and let the DSM target and $\mathbf{s}_\theta(\tilde{\mathbf{x}})$ have finite second moments. Then*

$$J_{\text{DSM}}(\theta) = \tfrac{1}{2}\, \mathbb{E}_{\tilde{\mathbf{x}}\sim p_\sigma} \Big[\big\| \mathbf{s}_\theta(\tilde{\mathbf{x}}) - \nabla \log p_\sigma(\tilde{\mathbf{x}}) \big\|^2 \Big] + C,$$

with C independent of θ. The right-hand side is the explicit objective Equation 38.74 for p_σ, so denoising and explicit score matching on the noised data have the same minimizers and the same gradients.

Proof. Apply the lemma with $X = \tilde{\mathbf{x}}$ and $Y = \nabla_{\tilde{\mathbf{x}}} \log p_\sigma(\tilde{\mathbf{x}} \mid \mathbf{x})$. It remains to identify the conditional mean of Y, and the identification is the content of the theorem: *the marginal*

score is the posterior mean of the conditional scores. Writing the posterior $p(\mathbf{x} \mid \tilde{\mathbf{x}})$ by Bayes' rule and then exchanging gradient and integral,

$$\begin{aligned}\mathbb{E}\left[Y \mid \tilde{\mathbf{x}}\right] &= \int \frac{\nabla_{\tilde{\mathbf{x}}}\, p_\sigma(\tilde{\mathbf{x}} \mid \mathbf{x})}{p_\sigma(\tilde{\mathbf{x}} \mid \mathbf{x})}\, \frac{p_\sigma(\tilde{\mathbf{x}} \mid \mathbf{x})\, p(\mathbf{x})}{p_\sigma(\tilde{\mathbf{x}})}\, d\mathbf{x} \\ &= \frac{\nabla_{\tilde{\mathbf{x}}} \int p_\sigma(\tilde{\mathbf{x}} \mid \mathbf{x})\, p(\mathbf{x})\, d\mathbf{x}}{p_\sigma(\tilde{\mathbf{x}})} = \nabla \log p_\sigma(\tilde{\mathbf{x}}).\end{aligned}$$

■

The proof used no property of the Gaussian beyond differentiability, and none of the data distribution: for a finite training set the integral is a sum, and the theorem returns the score of the blurred training set. That has a sobering consequence. A network that reached the minimum at every noise level, and a sampler that followed its score without error, would return the training points and nothing else (Exercise 3 takes the limit $\sigma \to 0$). A model that produces new samples therefore owes them to a regression that stops short of its minimum, not to the objective; Section 17.10 lists what is known about when it does. Gaussian noise is chosen because its conditional score Equation 38.76 is the noise itself, scaled by $-1/\sigma$. Predicting the score and predicting the noise ϵ are therefore the same task, and a network ϵ_θ that predicts the noise defines the score model $\mathbf{s}_\theta = -\epsilon_\theta/\sigma$. This *noise-prediction* form is the one diffusion models use.

Example: the Gaussian. Let the data be $x \sim \mathcal{N}(0, v)$ again, noised at level σ. The noised marginal is $p_\sigma = \mathcal{N}(0, v + \sigma^2)$, with score $-\tilde{x}/(v + \sigma^2)$. Bayes' rule for two Gaussians gives the posterior of the clean point,

$$x \mid \tilde{x} \;\sim\; \mathcal{N}\Big(\frac{v}{v+\sigma^2}\,\tilde{x},\; \frac{v\sigma^2}{v+\sigma^2}\Big),$$

a shrinkage of the noisy point toward the data mean, with a spread that does not depend on $\tilde{x}$. Every claim of this subsection becomes a formula. The conditional mean of the target $Y = (x - \tilde{x})/\sigma^2$ is $(\frac{v}{v+\sigma^2} - 1)\tilde{x}/\sigma^2 = -\tilde{x}/(v + \sigma^2)$, the marginal score, as Vincent's theorem says. The best possible mean squared error, the second term of Equation 38.78, is the posterior variance of the target, $\mathrm{Var}(x \mid \tilde{x})/\sigma^4 = \frac{1}{\sigma^2} - \frac{1}{v+\sigma^2}$. With $v = 1$ and $\sigma = 0.5$ the floor is $4 - 0.8 = 3.2$, and the posterior mean of the clean point is $0.8\,\tilde{x}$. A Monte Carlo run on 100,000 pairs checks both numbers.

```
x, eps = rng.standard_normal((2, 100000))            # v = 1
xt, y = x + 0.5 * eps, -eps / 0.5                     # sigma = 0.5; DSM target y
print(f'mean squared error of the conditional mean -xt/1.25: {((y + xt /
↪  1.25)**2).mean():.3f}'
      '  (floor 3.2)')
print(f'least-squares slope of x on xt: {(xt * x).mean() / (xt**2).mean():.3f}  (0.8)')
```

```
mean squared error of the conditional mean -xt/1.25: 3.215  (floor 3.2)
least-squares slope of x on xt: 0.801  (0.8)
```

The best regression function has mean squared error 3.2 on the targets, not 0, because many clean points explain one noisy point and the regression can only return their average. Here

a loss of 3.2 is a perfect fit, and a loss of zero is impossible. The slope 0.80 is the shrinkage factor $v/(v+\sigma^2)$. Neither observation is special to the Gaussian.

The slope is an instance of *Tweedie's formula*, which Robbins published and credited to Tweedie (Robbins 1956; Efron 2011). Rearranging Vincent's identity $\mathbb{E}[(\mathbf{x}-\tilde{\mathbf{x}})/\sigma^2 \mid \tilde{\mathbf{x}}] = \nabla \log p_\sigma(\tilde{\mathbf{x}})$ gives

$$\mathbb{E}[\mathbf{x} \mid \tilde{\mathbf{x}}] = \tilde{\mathbf{x}} + \sigma^2 \nabla \log p_\sigma(\tilde{\mathbf{x}}). \tag{38.79}$$

The best denoiser under squared error moves the noisy point one step of size σ^2 up the score of the noised data (Figure 38.23). For the Gaussian it reads $\tilde{x} - \sigma^2 \tilde{x}/(v+\sigma^2) = 0.8\,\tilde{x}$, the slope just measured.

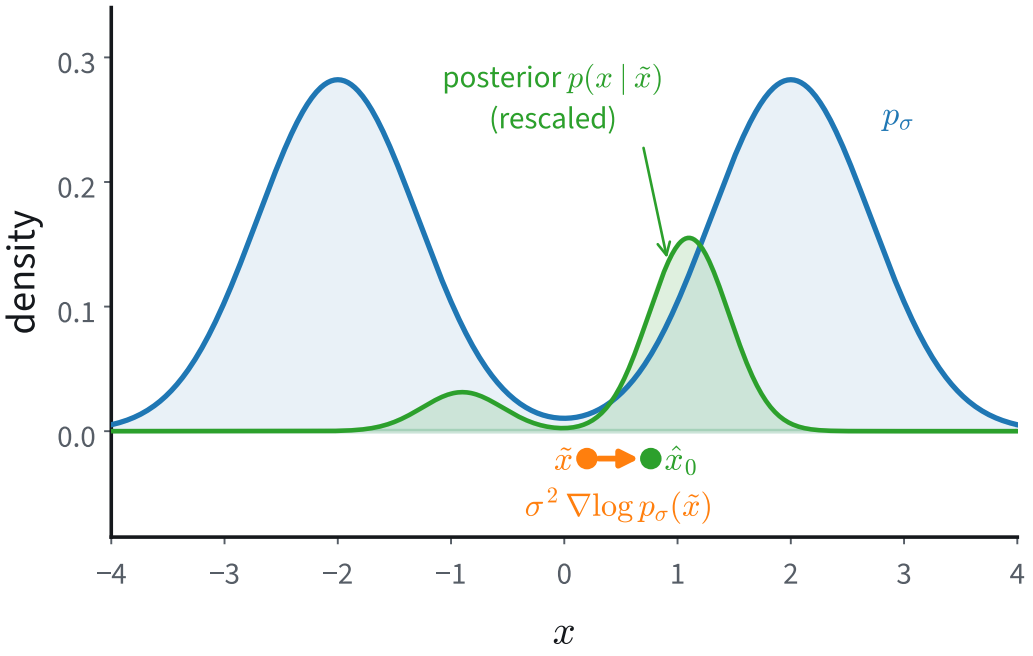


Figure 38.23: Tweedie's formula for the mixture $\frac{1}{2}\mathcal{N}(-2, 0.5^2) + \frac{1}{2}\mathcal{N}(2, 0.5^2)$ noised at $\sigma = 0.5$. A noisy observation $\tilde{x}$ lies in the low-density valley of the noised density p_σ. The posterior of the clean point is bimodal but asymmetric, and the step $\tilde{x} + \sigma^2 \nabla \log p_\sigma(\tilde{x})$ lands on its mean $\hat{x}_0 = \mathbb{E}[x \mid \tilde{x}]$, the optimal denoiser.

The floor is general as well. At the optimum the mean squared error $2J_{\text{DSM}}$ equals $\mathbb{E}\|Y - \mathbf{m}(X)\|^2$, the average posterior variance of the target and the *Bayes risk* of the regression. To compute it, apply the lemma Equation 38.78 to the field $\mathbf{v} = \mathbf{0}$, which gives $\mathbb{E}\|Y\|^2 = \mathbb{E}\|\mathbf{m}(X)\|^2 + \mathbb{E}\|Y - \mathbf{m}(X)\|^2$. The target $Y = -\epsilon/\sigma$ has $\mathbb{E}\|Y\|^2 = d/\sigma^2$ in d dimensions, and its conditional mean is the score, so

$$\mathbb{E}\left\|Y - \nabla \log p_\sigma(\tilde{\mathbf{x}})\right\|^2 = \frac{d}{\sigma^2} - \mathbb{E}_{\tilde{\mathbf{x}} \sim p_\sigma}\left\|\nabla \log p_\sigma(\tilde{\mathbf{x}})\right\|^2. \tag{38.80}$$

For the Gaussian this is $\frac{1}{\sigma^2} - \frac{1}{v+\sigma^2}$ again. An observed loss also contains the fit's own error, so it should be compared with this floor, not with zero.

The floor also says what goes wrong at a very small noise level. For data with a smooth density it grows like d/σ^2 as $\sigma \to 0$. The target $-\epsilon/\sigma$ becomes large, while the score it averages to keeps a finite mean square. A single small noise level therefore gives a poorly conditioned regression, whose targets are mostly noise.

Tweedie's formula says where the best denoiser sends a noisy point. Differentiating it once more says how fast that answer changes with the noisy point, and this turns out to limit how sharply a noised density can curve. We need the bound only once, in the last subsection, where a sampler is steered toward a class, but the proof takes a few lines here, while the posterior is in front of us.

Proposition (curvature of a noised density). *For any data distribution and any $\sigma > 0$,*

$$\nabla^2 \log p_\sigma(\tilde{\mathbf{x}}) = -\frac{\mathbf{I}}{\sigma^2} + \frac{1}{\sigma^4}\,\mathrm{Cov}\left[\mathbf{x} \mid \tilde{\mathbf{x}}\right], \tag{38.81}$$

so no eigenvalue of $-\nabla^2 \log p_\sigma$ exceeds $1/\sigma^2$.

Proof. By Tweedie's formula Equation 38.79, $\nabla \log p_\sigma(\tilde{\mathbf{x}}) = (\mathbb{E}[\mathbf{x} \mid \tilde{\mathbf{x}}] - \tilde{\mathbf{x}})/\sigma^2$, so it remains to differentiate the posterior mean $\mathbb{E}[\mathbf{x} \mid \tilde{\mathbf{x}}] = \int \mathbf{x}\, p(\mathbf{x} \mid \tilde{\mathbf{x}})\, d\mathbf{x}$. The posterior depends on $\tilde{\mathbf{x}}$ through Bayes' rule, which gives $\nabla_{\tilde{\mathbf{x}}} \log p(\mathbf{x} \mid \tilde{\mathbf{x}}) = \nabla_{\tilde{\mathbf{x}}} \log p_\sigma(\tilde{\mathbf{x}} \mid \mathbf{x}) - \nabla \log p_\sigma(\tilde{\mathbf{x}}) = (\mathbf{x} - \mathbb{E}[\mathbf{x} \mid \tilde{\mathbf{x}}])/\sigma^2$. Differentiating under the integral with $\nabla p = p\, \nabla \log p$, the Jacobian of the posterior mean is

$$\int \mathbf{x}\, \frac{(\mathbf{x} - \mathbb{E}[\mathbf{x} \mid \tilde{\mathbf{x}}])^\top}{\sigma^2}\, p(\mathbf{x} \mid \tilde{\mathbf{x}})\, d\mathbf{x} = \frac{1}{\sigma^2}\,\mathrm{Cov}\left[\mathbf{x} \mid \tilde{\mathbf{x}}\right].$$

Subtracting the Jacobian $\mathbf{I}$ of $\tilde{\mathbf{x}}$ and dividing by σ^2 gives Equation 38.81. A covariance is positive semidefinite, which gives the bound. ■

A noised density cannot curve more sharply than its own noise: none of its peaks is narrower than the Gaussian that blurred it. For Gaussian data the right-hand side of Equation 38.81 is $-\frac{1}{\sigma^2} + \frac{v}{\sigma^2(v+\sigma^2)} = -\frac{1}{v+\sigma^2}$, the curvature of $\log \mathcal{N}(0, v + \sigma^2)$, as it should be.

38.6.1.3 A Least-Squares Score in One Dimension

The Gaussian has a linear score, which a one-parameter model can represent. Does least squares on noisy pairs also recover a curved score? The data are the mixture of the previous section, $p = \frac{1}{2}\mathcal{N}(-2, 0.5^2) + \frac{1}{2}\mathcal{N}(2, 0.5^2)$, noised at $\sigma = 0.5$. Noising adds σ^2 to each component's variance, so p_σ is again a mixture, with component variance $0.25 + 0.25 = 0.5$. As derived in Section 38.5.3, a symmetric mixture with means $\pm\mu$ and common component variance v has the score $(\mu \tanh(\mu x/v) - x)/v$. With $\mu = 2$ and $v = 0.5$ the target curve is

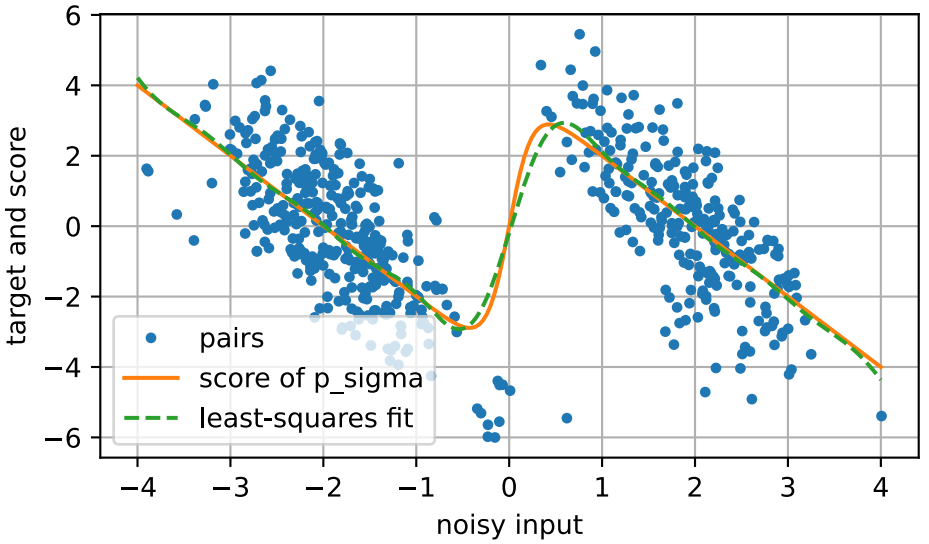
$$\nabla \log p_\sigma(\tilde{x}) = 4 \tanh(4\tilde{x}) - 2\tilde{x}.$$

Near each mode it is close to $-(\tilde{x} \mp 2)/0.5$, the linear score of a single component. Between the modes it rises from about -2.9 at $\tilde{x} = -0.45$ to $+2.9$ at $\tilde{x} = 0.45$, pushing points out of the valley.

The model is not a network. It is a linear combination of 17 fixed features, sixteen Gaussian bumps of width 0.6 centered on $[-4.5, 4.5]$ plus the identity x. We choose it because fitting the DSM loss Equation 38.77 is then ordinary least squares on the pairs $(\tilde{x}, -\epsilon/\sigma)$, which `np.linalg.lstsq` solves exactly: any gap between the fit and the score is sampling error, with no optimizer to blame. Noise costs nothing to draw, so each of the 4,096 clean points is noised four times. We fit the model on these 16,384 pairs, compare it with the analytic score, and evaluate both far outside the data, at $x = 8$. The printed mean squared errors are $2J_{\text{DSM}}$, the quantity the regression lemma splits.

```
n, sigma = 4096, 0.5
def mixture_score(q, var, means=(-2.0, 2.0), ws=(0.5, 0.5)):   # analytic score, any level
    logw = np.stack([np.log(w) - (q - mu)**2 / (2 * var) for mu, w in zip(means, ws)])
    r = np.exp(logw - logw.max(0))                              # responsibilities, stable
↪   softmax
    return (r * np.stack([(mu - q) / var for mu in means])).sum(0) / r.sum(0)
def noisy_pairs(n, rng):                                        # n points of p, four noisings
↪   each
    x = np.repeat(rng.normal(4.0 * rng.integers(0, 2, n) - 2.0, 0.5), 4)   # x ~ p, the
↪   mixture
    eps = rng.standard_normal(x.size)
    return x + sigma * eps, -eps / sigma                        # the (input, target) pairs
centers = np.linspace(-4.5, 4.5, 16)
def features(q):                                               # identity + 16 bumps, width 0.6
    return np.column_stack([q, np.exp(-(q[:, None] - centers)**2 / 0.72)])
def fit(xt, y):                                                # least squares = the regression
    return np.linalg.lstsq(features(xt), y, rcond=None)[0]
xt, y = noisy_pairs(n, rng)
w_ls = fit(xt, y)
s_hat = lambda q: features(q) @ w_ls
g = np.linspace(-8, 8, 16001); dg = g[1] - g[0]
p_sigma = 0.5 * (np.exp(-(g + 2)**2) + np.exp(-(g - 2)**2)) / np.sqrt(np.pi)  # var 0.5 each
floor = 1 / sigma**2 - (p_sigma * mixture_score(g, 0.5)**2).sum() * dg
excess = (p_sigma * (s_hat(g) - mixture_score(g, 0.5))**2).sum() * dg
print(f'mean squared error: fit {((s_hat(xt) - y)**2).mean():.2f}, exact score '
      f'{((mixture_score(xt, 0.5) - y)**2).mean():.2f}, population floor {floor:.2f}')
print(f'excess risk {excess:.3f}; fit at x = 8: {s_hat(np.array([8.0]))[0]:+.1f}, '
      f'exact score {4 * np.tanh(32) - 16:.0f} (no pairs there)')
grid = np.linspace(-4, 4, 201)
d2l.plot([xt[:600], grid, grid], [y[:600], 4 * np.tanh(4 * grid) - 2 * grid, s_hat(grid)],
         'noisy input', 'target and score', fmts=('.', '-', '--'),
         legend=['pairs', 'score of p_sigma', 'least-squares fit'], figsize=(5.5, 3))
```

```
mean squared error: fit 2.08, exact score 2.07, population floor 2.11
excess risk 0.009; fit at x = 8: -4.7, exact score -12 (no pairs there)
```



The plot shows all that the regression sees: a cloud of targets $-\epsilon/\sigma$ that spreads over several units at every $\tilde{x}$. Through the cloud runs its conditional mean, the score of p_σ, and the least-squares fit lies on it. The printout follows the regression lemma. The population floor Equation 38.80, here $\frac{1}{\sigma^2} - \mathbb{E}_{p_\sigma}(\nabla \log p_\sigma)^2$ computed by quadrature, is 2.11. It is the mean squared vertical spread of the cloud around the curve. The fit and the exact score both print a mean squared error within a tenth of this floor, and they agree with each other to about a hundredth. Neither need land on 2.11, because each is an average over one sample of pairs, and such an average moves from one sample to the next. What separates the fit from the score is the *excess risk*, the first term of the lemma. It is around 0.01, under one percent of the floor.

The last number shows the limit of the method. No noisy pair lands beyond $|\tilde{x}| \approx 5$, so at $x = 8$ the fit is whatever the basis extrapolates to. The printed value may fall near the exact -12 or far from it, and one fit cannot tell us which to expect. Let us therefore repeat the experiment twenty times, each time with fresh data and fresh noise. We record the exact score's mean squared error, to see how far one sample moves it, and the fit at two points: at $x = 1$, on the flank of the right mode, where pairs are plentiful, and at $x = 8$, where there are none.

```
runs = [noisy_pairs(n, rng) for _ in range(20)]                # 20 fresh draws of data and
↪ noise
errs = np.array([((mixture_score(xt_r, 0.5) - y_r)**2).mean() for xt_r, y_r in runs])
W = np.array([fit(xt_r, y_r) for xt_r, y_r in runs])           # (20, 17) fitted weights
print(f'exact score over 20 draws: mean squared error {errs.mean():.2f}, sd
↪ {errs.std():.3f}')
for q in (1.0, 8.0):
    s_q = W @ features(np.array([q]))[0]
    print(f'fit at x = {q:.0f} over 20 draws: from {s_q.min():+.1f} to {s_q.max():+.1f} '
          f'(exact score {4 * np.tanh(4 * q) - 2 * q:+.1f})')
```

```
exact score over 20 draws: mean squared error 2.11, sd 0.037
fit at x = 1 over 20 draws: from +1.8 to +2.1 (exact score +2.0)
```

```
fit at x = 8 over 20 draws: from -15.6 to +17.1 (exact score -12.0)
```

Over the twenty draws the exact score's mean squared error has a standard deviation of 0.03 to 0.04. One sample of pairs therefore gives the floor to a few hundredths and no better (Exercise 3 obtains the same figure from a single sample). At $x = 1$ the twenty fits stay within about two tenths of the exact score $+2.0$. At $x = 8$ they are spread over tens of units and take both signs: in some draws the fitted score is positive and points away from the data. The regression has no opinion about a place where it saw no pairs, and it returns a number there all the same. Least squares constrains the fit only where pairs fall, so a sampler that starts far from the data meets an arbitrary field there.

38.6.2 Sampling with a Score

The samplers of the previous section need the score at every noise level of a forward process. With the score at one level only, or the exact score of p itself, we can still draw samples, by a method that needs no forward process at all. It carries the name of Paul Langevin, who in 1908 described a particle in a fluid by a force plus random kicks.

38.6.2.1 Langevin Dynamics

Picture a particle that drifts uphill on the log-density and is jostled by noise. The drift pulls it toward high density, and the noise keeps it from collapsing onto a mode. **Langevin dynamics** is the SDE

$$d\mathbf{X} = \tfrac{1}{2}\nabla \log p(\mathbf{X})\, dt + d\mathbf{W}, \tag{38.82}$$

whose drift is half the score and whose noise is a standard Brownian motion $\mathbf{W}$. The claim is that p is its stationary law.

For the Gaussian $p = \mathcal{N}(0, v)$, with score $-x/v$, we already know this. Then Equation 38.82 reads $dX = -\frac{X}{2v}\, dt + dW$, an Ornstein–Uhlenbeck process. An OU process $dX = -\theta X\, dt + g\, dW$ has the stationary law $\mathcal{N}(0, g^2/(2\theta))$ (Section 38.4.3), and here $\theta = 1/(2v)$ and $g = 1$ give the variance v. The proposition extends this to smooth targets whose score grows at most linearly.

Proposition (stationarity). *Let p be a smooth positive density whose score $\nabla \log p$ is locally Lipschitz and grows at most linearly, so that Equation 38.82 has a unique solution that does not explode and its Fokker–Planck equation has a unique solution for each initial density. If $\mathbf{X}_0 \sim p$, then $\mathbf{X}_t \sim p$ for all $t \geq 0$.*

Proof. For an SDE $d\mathbf{X} = \mathbf{b}\, dt + d\mathbf{W}$ with unit diffusion, the density q_t of $\mathbf{X}_t$ solves the Fokker–Planck equation $\partial_t q = -\nabla \cdot (q\, \mathbf{b}) + \frac{1}{2}\Delta q$ (Section 38.5.2). Substitute $\mathbf{b} = \frac{1}{2}\nabla \log p$ and $q = p$, and use $p\, \nabla \log p = \nabla p$:

$$-\nabla \cdot \big(\tfrac{1}{2}\, p\, \nabla \log p\big) + \tfrac{1}{2}\Delta p = -\tfrac{1}{2}\nabla \cdot (\nabla p) + \tfrac{1}{2}\Delta p = 0.$$

So $q_t \equiv p$ solves the equation with the initial density p. Uniqueness turns "p solves the equation" into "the law of $\mathbf{X}_t$ is p". ■

The drift transports mass up the log-density at exactly the rate at which the noise spreads it out. Stationarity says nothing yet about a chain started elsewhere, and a sampler always

starts elsewhere. For targets like those of the proposition, smooth and positive with a score that grows at most linearly, the law of $\mathbf{X}_t$ does converge to p from every starting point (Roberts and Tweedie 1996). We do not prove it. How long the convergence takes is another matter, and the experiment below measures it.

To sample, we discretize. The Euler–Maruyama step of Equation 38.82 with step h is the **Langevin sampler**

$$\mathbf{x} \leftarrow \mathbf{x} + \tfrac{h}{2}\,\mathbf{s}(\mathbf{x}) + \sqrt{h}\,\xi, \qquad \xi \sim \mathcal{N}(\mathbf{0}, \mathbf{I}),$$

which needs only a score $\mathbf{s}$, exact or learned. The discretized chain does not inherit the stationarity of the SDE. For the Gaussian the step is the linear recursion $x \leftarrow (1 - \frac{h}{2v})\,x + \sqrt{h}\,\xi$, whose stationary variance V solves $V = (1-\frac{h}{2v})^2 V + h$. Hence $V = v/(1-h/(4v))$, and the sampler overstates the variance by about $h/4$ (Exercise 1 measures this).

A Gaussian has one mode, and one mode hides the difficulty. Let us give the sampler the lopsided mixture of Section 38.5.5, $p = 0.8\,\mathcal{N}(-2, 0.5^2) + 0.2\,\mathcal{N}(2, 0.5^2)$, and let us give it the exact score of p, not a fit, so that whatever goes wrong is the sampler's doing. A fifth of the mass of p lies right of zero, and $\mathbb{E}X^2 = 4 + 0.25 = 4.25$ whatever the weights. We run 10,000 chains for 2,000 steps of size $h = 0.01$, three times: from a broad start $\mathcal{N}(0, 3^2)$, from the left mean, and from the right mean. Before reading the output, guess what fraction of the broad start ends right of zero, and how many chains started in one mode end in the other.

```
def langevin(q, h, steps, var=0.25, ws=(0.8, 0.2)):   # var: component variance of p
    for _ in range(steps):
        xi = rng.standard_normal(q.shape)
        q = q + 0.5 * h * mixture_score(q, var, ws=ws) + np.sqrt(h) * xi
    return q
warm = langevin(rng.normal(0.0, 3.0, 10000), 0.01, 2000)
print(f'broad start: P(X > 0) = {(warm > 0).mean():.3f}, '
      f'E[X^2] = {(warm**2).mean():.2f}  (truth 0.2, 4.25)')
for start, w in ((-2.0, 0.8), (2.0, 0.2)):              # every chain in one mode
    q = langevin(np.full(10000, start), 0.01, 2000)
    print(f'all chains at {start:+.0f} (weight {w}): '
          f'{(np.sign(q) != np.sign(start)).mean():.3f} end on the other side')
```

```
broad start: P(X > 0) = 0.484, E[X^2] = 4.20  (truth 0.2, 4.25)
all chains at -2 (weight 0.8): 0.006 end on the other side
all chains at +2 (weight 0.2): 0.025 end on the other side
```

Of the 10,000 chains started in the heavy mode, about one in two hundred has ended on the other side. Of those started in the light mode, about one in forty has. In the valley the density is tiny and the drift points back to the mode the chain came from. Crossing needs a rare run of noise in one direction, and the waiting time grows exponentially with the height of the barrier in $-\log p$.

The two fractions differ, as they should. In equilibrium as much mass flows from left to right as from right to left, so the light mode, with a quarter of the heavy mode's mass, must give up its chains four times as readily. The sampler has its rates in about the right ratio. They are far too small. A system with two states forgets its initial proportions at the sum of its two rates, and the two printed fractions add up to about 0.03 per run. An error in the

proportions therefore shrinks by a factor e only every $2{,}000/0.03 \approx 67{,}000$ steps, more than thirty times the length of our run.

This changes how the first line must be read. The second moment matches 4.25 to within a few hundredths, the sampling error of 10,000 chains (the step-size bias $h/4 \approx 0.0025$ is far below it). Each chain has equilibrated inside the mode it fell into. But almost half of the chains end right of zero, where p has a fifth of its mass. The start is symmetric, each chain slides into the mode on its side of the valley, and hardly any leaves. The proportions are those of the start, not of p (Section 17.1 measures the same blindness with a learned score on three modes).

The time a chain needs to forget where it started is its *mixing time*. Slow mixing is a property of the SDE itself, so a smaller step does not cure it. A smaller step removes only the discretization bias.

38.6.2.2 Limits of a Single Noise Level

A score at one noise level has now failed in three ways. Where no noisy pair landed, the learned score is arbitrary. With the correct score, Langevin dynamics hardly ever crosses the valley between the modes. And its samples have the proportions of its start: almost half of the mass in a mode that holds a fifth. The third failure is not the sampler's alone. Where a mixture of well-separated modes has mass, its score is almost that of the nearest mode, whatever the weights (Section 37.5.1): at low noise the score hardly records the proportions.

The reverse sampler of Section 38.5.5 met the same lopsided mixture from an equally symmetric start and returned four fifths of the mass to the left. It had what Langevin dynamics lacks, the score at every noise level. Its proportions were settled at intermediate noise, where the two components still overlap and their pulls already differ, so that the weights decide where a particle goes.

A decreasing sequence of noise levels addresses all three limits. At a large σ the noised density has no valley, and the noisy pairs cover the whole region a sampler visits. For the symmetric mixture, p_σ has component variance $0.25 + \sigma^2$ and is unimodal once its component standard deviation reaches the distance 2 from the center to each mean (Exercise 4 of the previous section), that is, for $\sigma \geq 1.94$. Unequal weights close the valley sooner. As σ shrinks the detail returns, and a sampler that has already spread its mass across both modes only needs to refine it.

Annealed Langevin dynamics (Song and Ermon 2019) runs the sampler down such a sequence, each level with its own score. Exercise 2 frees chains from a single mode this way, and Section 17.2 shows that such a sequence recovers unequal weights. Levels chosen one by one leave two questions open: how to space them, and how to move from one to the next. A forward process settles both. It supplies a continuum of noise levels, and its reverse is the sampler.

38.6.3 Scores Across Noise Levels

From here on the model is one field $\mathbf{s}_\theta(\mathbf{x}, t)$ that takes the noise level as a second input and is trained at every level at once, by the loss of the next subsection. The samplers are the two we started from, the reverse-time SDE of Section 38.5.5 and the probability-flow ODE of Section 38.5.4.3.

38.6.3.1 Forward Processes and the Noise-Conditional Loss

DSM needs the conditional law of the noised point given the clean one, and it needs that law to be Gaussian so that the target is explicit. Every SDE whose drift is linear in $\mathbf{x}$ and whose noise amplitude does not depend on $\mathbf{x}$ has Gaussian transition kernels, as Section 38.4.3 showed for the OU process. The two standard forward processes of Section 38.4 are of this kind, and their kernels can be written once for both as

$$p_t(\mathbf{x}_t \mid \mathbf{x}_0) = \mathcal{N}(\alpha_t \mathbf{x}_0,\, \sigma_t^2 \mathbf{I}), \qquad \text{i.e.} \qquad \mathbf{x}_t = \alpha_t \mathbf{x}_0 + \sigma_t \epsilon.$$

The coefficient α_t shrinks the signal and σ_t scales the noise. The two processes differ only in these two numbers.

- The **variance-exploding (VE)** SDE $d\mathbf{X} = \sqrt{\frac{d}{dt}\sigma^2(t)}\; d\mathbf{W}$, with $\sigma^2(t)$ increasing and $\sigma(0) = 0$, adds noise without shrinking the data: $\alpha_t = 1$ and $\sigma_t = \sigma(t)$. It is the continuous version of the sequence of noise levels that annealed Langevin dynamics descends.
- The **variance-preserving (VP)** SDE is an Ornstein–Uhlenbeck process with a time-dependent rate,

  $$d\mathbf{X} = -\tfrac{1}{2}\beta(t)\,\mathbf{X}\,dt + \sqrt{\beta(t)}\; d\mathbf{W},$$

  with $\alpha_t = e^{-\frac{1}{2}\int_0^t \beta(r)\,dr}$ and $\sigma_t^2 = 1 - \alpha_t^2$, so that unit-variance data keep unit variance at every t (Section 38.4).

The rest of the section works with the VP process. The VE process returns once, as the coordinates in which DDIM turns out to be simple.

With a Gaussian kernel at every t, the conditional score is $-\epsilon/\sigma_t$, and DSM applies at each level. Averaging over levels with a weight $w(t) > 0$ gives the **noise-conditional DSM loss**

$$\mathcal{L}(\theta) = \tfrac{1}{2}\,\mathbb{E}_t\, w(t)\, \mathbb{E}_{\mathbf{x}_0,\,\epsilon}\left[\left\| \mathbf{s}_\theta(\alpha_t \mathbf{x}_0 + \sigma_t \epsilon, t) + \frac{\epsilon}{\sigma_t} \right\|^2\right], \tag{38.83}$$

with t drawn uniformly from the training range. Each draw of $(t, \mathbf{x}_0, \epsilon)$ is one regression pair. This loss is what the rest of the section uses. The choice of the weight, to which we turn now, is a detail on a first reading.

The weight decides which levels the network serves best, not what it learns. For any positive w the population minimizer is $\mathbf{s}(\cdot, t) = \nabla \log p_t$ at every t, by Vincent's theorem at each level, provided the network can represent it. What the weight changes is the size of each level's contribution. By Equation 38.80 the floor at level t is $d/\sigma_t^2 - \mathbb{E}\|\nabla \log p_t\|^2$, which grows like d/σ_t^2 at small σ_t. An unweighted loss is therefore dominated by its least noisy levels.

Two weights are standard. The discrete-time model of the next subsection uses $w(t) = \sigma_t^2$, which cancels this growth, so that every level contributes a loss of the same order.

The *likelihood weighting* $w = g^2$, the squared diffusion coefficient of the forward SDE, is the weight under which the score error controls a likelihood. The squared error of the score

under this weight, integrated over the levels, bounds the KL divergence from the data to the model that the reverse-time SDE defines, apart from a term for the mismatch between p_T and the Gaussian the sampler starts from (Exercise 7). The loss we can compute contains the floor on top of this error, and under the likelihood weighting the floor is troublesome. For the VP process $g^2 = \beta$, and the weighted floor grows like d/t as $t \to 0$, so its integral down to $t = 0$ is infinite. Training therefore starts at a small time $\varepsilon > 0$ (Exercise 4).

Generation needs nothing new. The two samplers run with the learned field in place of the exact score. For a forward SDE $d\mathbf{X} = \mathbf{f}(\mathbf{X}, t)\, dt + g(t)\, d\mathbf{W}$ on $[0, T]$, we start from the Gaussian that approximates p_T and integrate either the reverse-time SDE (Section 38.5.5) or the probability-flow ODE (Section 38.5.4.3) from $t = T$ down to 0:

$$d\mathbf{X} = \big[\mathbf{f} - g^2\, \mathbf{s}_\theta(\mathbf{X}, t)\big]\, dt + g\, d\bar{\mathbf{W}}, \qquad \frac{d\mathbf{x}}{dt} = \mathbf{f} - \tfrac{1}{2} g^2\, \mathbf{s}_\theta(\mathbf{x}, t), \tag{38.84}$$

where $\bar{\mathbf{W}}$ is a Brownian motion running backward in time. For VP, $\mathbf{f} = -\frac{1}{2}\beta\mathbf{x}$ and $g^2 = \beta$.

A generative model is thus a triple: a forward process, a learned score, and a sampler. Figure 38.24 shows the three at work in two dimensions, the companion of the forward and reverse densities of Figure 38.21.

38.6.3.2 DDPM and the Variance-Preserving SDE

The Denoising Diffusion Probabilistic Model (DDPM) (Ho et al. 2020) is written in discrete time and looks at first like a separate construction. It has three parts, a noising chain, a loss and a sampler, and each turns out to be something we already have. The chain is a Markov chain of $N = 1000$ noising steps with a schedule $\beta_1, \dots, \beta_N \in (0, 1)$,

$$\mathbf{x}_t = \sqrt{1 - \beta_t}\, \mathbf{x}_{t-1} + \sqrt{\beta_t}\, \epsilon_t, \qquad \epsilon_t \sim \mathcal{N}(\mathbf{0}, \mathbf{I}) \text{ i.i.d.}, \tag{38.85}$$

which shrinks the state slightly and adds a little noise at each step. In this subsection and the next, t counts levels, $t = 0, 1, \dots, N$, and level t stands for the time $t\,\Delta$ of the SDE, with $\Delta = T/N$.

How is the chain related to the VP-SDE? With $\beta_t = \beta((t-1)\Delta)\,\Delta$ the step agrees to first order in β_t with the Euler–Maruyama step $(1 - \frac{1}{2}\beta_t)\,\mathbf{x}_{t-1} + \sqrt{\beta_t}\,\epsilon_t$ of the VP-SDE. More is true, and Section 38.4.4.1 showed it for a constant rate: the step is an *exact* transition of the VP process, over an interval of almost the same length. Between any two times the VP kernel has the form we wrote from time 0, with the integral of β taken between those times. Take an interval on which $\int \beta(r)\, dr = -\log(1 - \beta_t)$. The mean coefficient of its kernel is $e^{-\frac{1}{2}\int \beta} = \sqrt{1 - \beta_t}$ and the variance is $1 - (1 - \beta_t) = \beta_t$, the law of Equation 38.85 given $\mathbf{x}_{t-1}$. For the largest step of the standard schedule, $\beta_t = 0.02$, such an interval has $\int \beta = 0.0202$ instead of 0.02. The DDPM chain is therefore the VP process observed at discrete times, and the law of $\mathbf{x}_t$ given $\mathbf{x}_0$ is a VP kernel from time 0. This is the result of the subsection to remember; the DDPM training loss follows from it in two lines. In DDPM's own notation the marginal can also be verified directly on the chain, without the SDE.

Proposition (the $\bar{\alpha}$-marginal). *Let $\bar{\alpha}_t = \prod_{s=1}^{t}(1 - \beta_s)$. Then conditionally on $\mathbf{x}_0$,*

$$\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\, \mathbf{x}_0 + \sqrt{1 - \bar{\alpha}_t}\, \epsilon, \qquad \epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I}). \tag{38.86}$$

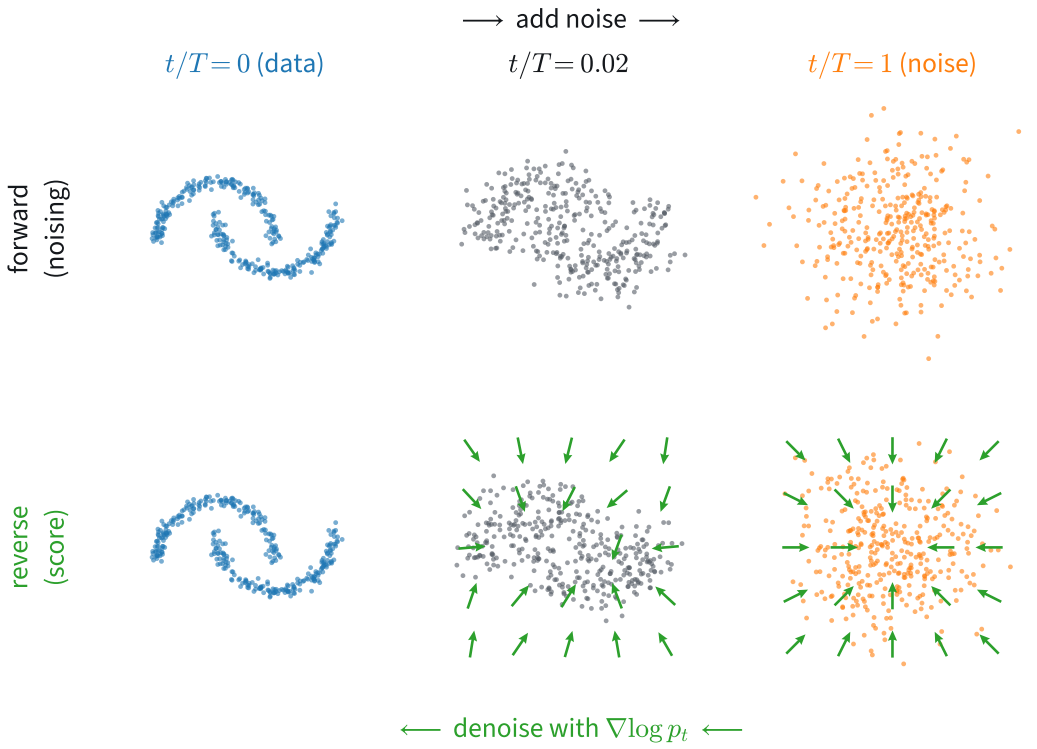


Figure 38.24: A two-moons distribution under the VP forward process at $t/T = 0, 0.02$, and 1 (top row, left to right), approaching an isotropic Gaussian. The bottom row traverses the same marginals in reverse, right to left; short arrows show the direction of the exact score $\nabla \log p_t$ of the noised data. At high noise the score is the pull toward the origin; at low noise it points toward the data.

Proof. Induction on t; the case $t = 0$ holds with $\bar{\alpha}_0 = 1$. Assume Equation 38.86 at $t-1$ and substitute it into Equation 38.85:

$$\mathbf{x}_t = \sqrt{(1-\beta_t)\bar{\alpha}_{t-1}}\,\mathbf{x}_0 + \sqrt{(1-\beta_t)(1-\bar{\alpha}_{t-1})}\,\epsilon' + \sqrt{\beta_t}\,\epsilon_t,$$

with ϵ' and ϵ_t independent standard Gaussians. A sum of independent Gaussians is Gaussian with the sum of the variances, $(1-\beta_t)(1-\bar{\alpha}_{t-1}) + \beta_t = 1 - (1-\beta_t)\bar{\alpha}_{t-1} = 1 - \bar{\alpha}_t$, and $(1-\beta_t)\bar{\alpha}_{t-1} = \bar{\alpha}_t$. ■

In the kernel notation of the previous subsection, DDPM has $\alpha_t = \sqrt{\bar{\alpha}_t}$ and $\sigma_t = \sqrt{1-\bar{\alpha}_t}$. Training therefore never simulates the chain. It jumps to any level in one step by Equation 38.86, and because the kernel is Gaussian, DSM applies directly: the conditional score is $-\epsilon/\sigma_t$. Parameterize the score model through a noise-prediction network, $\mathbf{s}_\theta(\mathbf{x}_t, t) = -\epsilon_\theta(\mathbf{x}_t, t)/\sigma_t$. Then

$$\left\| \mathbf{s}_\theta(\mathbf{x}_t, t) + \frac{\epsilon}{\sigma_t} \right\|^2 = \frac{1}{\sigma_t^2} \left\| \epsilon - \epsilon_\theta(\mathbf{x}_t, t) \right\|^2,$$

so the DDPM *simple loss* $\frac{1}{2}\mathbb{E}_{t,\mathbf{x}_0,\epsilon}\|\epsilon - \epsilon_\theta(\mathbf{x}_t, t)\|^2$ is the noise-conditional DSM loss Equation 38.83 with weight $w(t) = \sigma_t^2 = 1 - \bar{\alpha}_t$. "Predict the noise you added" is denoising score matching with a weight that cancels the $1/\sigma_t^2$ growth of the floor: every level contributes a loss of order d. (A warning for readers of (Ho et al. 2020): the paper writes α_t for $1-\beta_t$, where our α_t is the signal coefficient $\sqrt{\bar{\alpha}_t}$, and its simple loss lacks our factor $\frac{1}{2}$.)

The third part is the sampler. *Ancestral sampling* steps through the levels from N down to 1,

$$\mathbf{x}_{t-1} = \frac{\mathbf{x}_t + \beta_t\,\mathbf{s}_\theta(\mathbf{x}_t, t)}{\sqrt{1-\beta_t}} + \sqrt{\beta_t}\,\xi, \qquad \xi \sim \mathcal{N}(\mathbf{0}, \mathbf{I}),$$

without the noise in the last step. To first order in β_t this is $\mathbf{x}_t + \frac{1}{2}\beta_t\mathbf{x}_t + \beta_t\mathbf{s}_\theta + \sqrt{\beta_t}\,\xi$, the Euler–Maruyama step of the reverse SDE in Equation 38.84 for the VP process. The model was first derived by a different route (Sohl-Dickstein et al. 2015; Ho et al. 2020), as a bound on the log-likelihood of a latent-variable model, the route that Section 17.6 follows; it yields the same regression with a different weight on each level.

The samplers below use the standard schedule, linear in β_t from 10^{-4} to 0.02 over $N = 1000$ levels. The VP process is built for data of unit variance, so from here on the mixture is *standardized*: divided by $\sqrt{4.25}$, with modes at $\pm 2/\sqrt{4.25} = \pm 0.970$ and component standard deviation 0.243. Before any sampling, let us look at the schedule itself. We print $\bar{\alpha}_t$ at three levels and, at the first and the last level, the ratio $\sqrt{(1-\bar{\alpha}_t)/\bar{\alpha}_t}$ of noise to signal, which the next subsection uses as its time variable.

```
N = 1000
beta = np.linspace(1e-4, 0.02, N)                       # the DDPM schedule
alpha_bar = np.cumprod(1.0 - beta)
sigma_bar = np.sqrt((1 - alpha_bar) / alpha_bar)        # noise scale of x_t / sqrt(alpha_bar)
print(f'alpha_bar at t = 10, 100, 1000: {alpha_bar[9]:.3f}, {alpha_bar[99]:.3f}, '
      f'{alpha_bar[-1]:.0e}; sigma_bar from {sigma_bar[0]:.2f} to {sigma_bar[-1]:.0f}')
```

```
alpha_bar at t = 10, 100, 1000: 0.998, 0.897, 4e-05; sigma_bar from 0.01 to 157
```

The signal fades slowly at first and completely in the end. At $t = 100$, $\bar{\alpha} = 0.897$ and the noised law is still bimodal, with modes at $\pm\sqrt{0.897} \cdot 0.970 = \pm 0.92$ and component standard deviation $\sqrt{0.897 \cdot 0.243^2 + 0.103} = 0.39$. At $t = N$, $\bar{\alpha}_N \approx 4 \times 10^{-5}$, and the sample is a standard Gaussian whatever the data were. From the first level to the last the ratio of noise to signal grows from 0.01 to 157, by four orders of magnitude.

38.6.4 Deterministic Sampling and Guidance

Ancestral sampling calls the network once per level, $N = 1000$ times per sample. Can the same network produce samples with far fewer calls? DDIM does it, and the reason is a fact about ODE solvers. We prove it on exact scores, then run DDIM once on fitted ones, and finally ask what a sampler follows when it is steered toward a class.

38.6.4.1 Denoising Diffusion Implicit Models

The probability-flow ODE is deterministic and smooth, so a coarse grid of levels should suffice to integrate it. Denoising diffusion implicit models (DDIM) (J. Song et al. 2021) realize this with an update that reuses the trained noise-prediction network and skips levels. At level t, the network's noise estimate gives a guess of the clean sample by inverting the marginal Equation 38.86,

$$\hat{\mathbf{x}}_0 = \frac{\mathbf{x}_t - \sqrt{1-\bar{\alpha}_t}\,\epsilon_\theta(\mathbf{x}_t, t)}{\sqrt{\bar{\alpha}_t}}.$$

To move to any earlier level $t' < t$, DDIM rebuilds the sample from this guess and the *same* noise estimate, with the coefficients of level t':

$$\mathbf{x}_{t'} = \sqrt{\bar{\alpha}_{t'}}\,\hat{\mathbf{x}}_0 + \sqrt{1-\bar{\alpha}_{t'}}\,\epsilon_\theta(\mathbf{x}_t, t). \tag{38.87}$$

The update is deterministic: no fresh noise enters. Read literally, it moves a sample along the forward curve $\sqrt{\bar{\alpha}_t}\,\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\,\epsilon$ of the pair $(\hat{\mathbf{x}}_0, \epsilon_\theta)$. But a network trained by least squares returns the conditional mean $\mathbb{E}[\epsilon \mid \mathbf{x}_t]$, not the noise that produced this sample, and the estimate changes from level to level. What, then, does DDIM integrate, and how large an error does a long stride make?

A change of coordinates answers both questions. Divide by $\sqrt{\bar{\alpha}_t}$. The rescaled point $\bar{\mathbf{x}}_t = \mathbf{x}_t/\sqrt{\bar{\alpha}_t}$ equals $\mathbf{x}_0 + \bar{\sigma}_t\epsilon$ with $\bar{\sigma}_t = \sqrt{(1-\bar{\alpha}_t)/\bar{\alpha}_t}$, the ratio of noise to signal that we printed with the schedule. In these coordinates the VP process is a VE process, clean data plus noise of scale $\bar{\sigma}$, which grows from 0.01 at the first level to 157 at the last. The schedule only decides which values of $\bar{\sigma}$ the grid visits. On $\bar{\mathbf{x}}_t$ and $\bar{\sigma}_t$ the bar marks the rescaled point and its noise level; the bar on $\bar{\alpha}_t$ is DDPM's and means something else.

Proposition (DDIM is Euler's method). *In the coordinates $(\bar{\sigma}, \bar{\mathbf{x}})$ the DDIM update Equation 38.87 reads*

$$\bar{\mathbf{x}}_{t'} = \bar{\mathbf{x}}_t + (\bar{\sigma}_{t'} - \bar{\sigma}_t)\,\epsilon_\theta(\mathbf{x}_t, t), \tag{38.88}$$

one Euler step of the ODE $d\bar{\mathbf{x}}/d\bar{\sigma} = \epsilon_\theta$. If ϵ_θ is the exact noise prediction $\hat{\epsilon}(\mathbf{x}_t, t) = \mathbb{E}[\epsilon \mid \mathbf{x}_t]$, this ODE is the probability-flow ODE of the VP process, written in the coordinates $(\bar{\sigma}, \bar{\mathbf{x}})$.

Proof. Step (i) is algebra; the substance is in (ii) and (iii), which show that the probability-flow ODE takes this form in the new coordinates. In (ii), t is the continuous time of the SDE.

(i) Divide Equation 38.87 by $\sqrt{\bar{\alpha}_{t'}}$ and substitute $\hat{\mathbf{x}}_0 = \bar{\mathbf{x}}_t - \bar{\sigma}_t \epsilon_\theta$: $\bar{\mathbf{x}}_{t'} = \hat{\mathbf{x}}_0 + \bar{\sigma}_{t'} \epsilon_\theta = \bar{\mathbf{x}}_t + (\bar{\sigma}_{t'} - \bar{\sigma}_t)\epsilon_\theta$.

(ii) Change variables in the VP probability-flow ODE $d\mathbf{x}/dt = -\frac{1}{2}\beta\mathbf{x} - \frac{1}{2}\beta\nabla_{\mathbf{x}} \log p_t$ of Equation 38.84, with $\alpha_t = \sqrt{\bar{\alpha}(t)}$ and $\bar{\alpha}(t) = e^{-\int_0^t \beta}$. Let $\bar{p}_t$ be the density of $\bar{\mathbf{x}}_t$; then $\nabla_{\mathbf{x}} \log p_t = \nabla_{\bar{\mathbf{x}}} \log \bar{p}_t / \sqrt{\bar{\alpha}}$. Since $d\sqrt{\bar{\alpha}}/dt = -\frac{1}{2}\beta\sqrt{\bar{\alpha}}$, the shrinkage terms cancel and $d\bar{\mathbf{x}}/dt = -\frac{\beta}{2\bar{\alpha}} \nabla \log \bar{p}_t$. Since $\bar{\sigma}^2 = 1/\bar{\alpha} - 1$, we have $d\bar{\sigma}/dt = \beta/(2\bar{\alpha}\bar{\sigma})$, and dividing gives $d\bar{\mathbf{x}}/d\bar{\sigma} = -\bar{\sigma}\, \nabla \log \bar{p}(\bar{\mathbf{x}})$.

(iii) Tweedie's formula for the kernel $\mathcal{N}(\mathbf{x}_0, \bar{\sigma}^2 \mathbf{I})$ reads $\mathbb{E}[\mathbf{x}_0 \mid \bar{\mathbf{x}}] = \bar{\mathbf{x}} + \bar{\sigma}^2 \nabla \log \bar{p}$, so $\mathbb{E}[\epsilon \mid \bar{\mathbf{x}}] = (\bar{\mathbf{x}} - \mathbb{E}[\mathbf{x}_0 \mid \bar{\mathbf{x}}])/\bar{\sigma} = -\bar{\sigma} \nabla \log \bar{p}$, the right-hand side of (ii). Conditioning on $\bar{\mathbf{x}}_t$ is the same as conditioning on $\mathbf{x}_t$. ■

DDIM is therefore a first-order solver for the probability-flow ODE, in a clock in which the noise level is the time. Its error is Euler's error: a stride holds ϵ_θ fixed while the exact $\hat{\epsilon}$ changes along the trajectory. Two corollaries mark the extremes of this error.

Corollary (a point mass is integrated without error). *If the data are a single point $\mathbf{x}_0$, every DDIM stride with the exact $\hat{\epsilon}$ lands on the probability-flow trajectory, whatever its length.* Here $\hat{\epsilon} = (\bar{\mathbf{x}} - \mathbf{x}_0)/\bar{\sigma}$, and along a solution of $d\bar{\mathbf{x}}/d\bar{\sigma} = (\bar{\mathbf{x}} - \mathbf{x}_0)/\bar{\sigma}$ the ratio $(\bar{\mathbf{x}} - \mathbf{x}_0)/\bar{\sigma}$ is constant. An Euler step for a constant right-hand side is exact.

Corollary (Gaussian data shrink). *For data $\mathbf{x}_0 \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ the exact flow is the identity, but one DDIM stride with the exact noise prediction multiplies every sample by*

$$\sqrt{\bar{\alpha}_t \bar{\alpha}_{t'}} + \sqrt{(1 - \bar{\alpha}_t)(1 - \bar{\alpha}_{t'})} = \cos(\phi_t - \phi_{t'}), \qquad \bar{\alpha}_t = \cos^2 \phi_t,$$

which is below 1 for every stride of positive length. The VP process leaves such data as they are: the marginal is $\mathcal{N}(\mathbf{0}, \mathbf{I})$ at every level, the score is $-\mathbf{x}$, and the probability-flow drift $-\frac{1}{2}\beta\mathbf{x} - \frac{1}{2}\beta(-\mathbf{x})$ vanishes, so no sample of the exact flow moves. By Tweedie, $\hat{\epsilon}(\mathbf{x}_t, t) = \sqrt{1 - \bar{\alpha}_t}\, \mathbf{x}_t$, and substituting into Equation 38.87 gives the factor. Writing $\bar{\alpha}_t = \cos^2 \phi_t$ turns it into the cosine of the angle advanced. In the bar coordinates the exact solution is $\bar{\mathbf{x}} \propto \sqrt{1 + \bar{\sigma}^2}$, so $\hat{\epsilon} = \bar{\sigma}\bar{\mathbf{x}}/(1 + \bar{\sigma}^2)$ changes along it. Holding it fixed across a stride removes too much signal, by the factor above.

The corollary gives the factor of one stride. Over a sampling run the factors multiply, and we compute the product for 10, 50 and 1,000 equally spaced strides of the DDPM schedule. How much of this error survives on data that are not Gaussian? To see, DDIM runs on the standardized mixture with its exact noise prediction $\hat{\epsilon} = -\sqrt{1 - \bar{\alpha}_t}\, \nabla \log p_t$, where the noised law at level t has modes $\pm\sqrt{\bar{\alpha}_t}\, m$ and component variance $\bar{\alpha}_t v_0 + 1 - \bar{\alpha}_t$, with $m = 0.970$ and $v_0 = 0.243^2$ the clean values. We compare 10 and 50 strides with 1,000 from the same initial draws in two ways: by the mean distance between paired endpoints, and by the Kolmogorov–Smirnov (KS) statistic of the two samples, the largest gap between their empirical distribution functions.

```
m, v0 = 2 / np.sqrt(4.25), 0.25 / 4.25                # standardized: modes +-m, variance v0
def ks(a, b):                                        # two-sample Kolmogorov-Smirnov statistic
    q = np.sort(np.concatenate([a, b]))
    return np.abs(np.searchsorted(np.sort(a), q, 'right') / len(a)
                  - np.searchsorted(np.sort(b), q, 'right') / len(b)).max()
def strides(K):                                       # K equal strides from level N to level 1
    return np.linspace(N - 1, 0, K + 1).round().astype(int)
for K in (10, 50, 1000):                                 # the Gaussian corollary
    a = alpha_bar[strides(K)]
    print(f'N(0,1) data, {K:4d} strides: every sample times '
          f'{np.prod(np.sqrt(a[:-1] * a[1:]) + np.sqrt((1 - a[:-1]) * (1 - a[1:]))):.3f}')
def eps_hat(xt, a, ws=(0.5, 0.5)):                    # exact noise prediction of the mixture
    means = (-np.sqrt(a) * m, np.sqrt(a) * m)
    return -np.sqrt(1 - a) * mixture_score(xt, a * v0 + 1 - a, means, ws)
def ddim(z, K, eps_fn):                                  # K DDIM strides from noise z
    xt, idx = z.copy(), strides(K)
    for t1, t2 in zip(idx[:-1], idx[1:]):
        a1, a2 = alpha_bar[t1], alpha_bar[t2]
        e = eps_fn(xt, a1)
        xt = np.sqrt(a2) * (xt - np.sqrt(1 - a1) * e) / np.sqrt(a1) + np.sqrt(1 - a2) * e
    return xt
z = rng.standard_normal(8000)                         # one draw, shared by every stride count
x_ref = ddim(z, 1000, eps_hat)
for K in (10, 50):
    x_K = ddim(z, K, eps_hat)
    print(f'mixture, {K:3d} strides vs 1000: mean |gap| {np.abs(x_K - x_ref).mean():.3f}, '
          f'KS {ks(x_K, x_ref):.3f}')
```

```
N(0,1) data,   10 strides: every sample times 0.832
N(0,1) data,   50 strides: every sample times 0.964
N(0,1) data, 1000 strides: every sample times 0.998
mixture,  10 strides vs 1000: mean |gap| 0.083, KS 0.077
mixture,  50 strides vs 1000: mean |gap| 0.015, KS 0.016
```

Ten strides shrink unit-variance Gaussian data by the factor 0.832, fifty by 0.964, and a thousand by 0.998. These factors are deterministic and measure the Euler error alone, since the noise prediction is exact. Five times as many strides cut the shortfall from 0.17 to 0.04, as a first-order method should. The mixture behaves the same way. Ten strides land on average 0.08 from the thousand-stride result, on a scale where the modes sit at ± 0.97, and the KS statistic is 0.08 as well. Fifty strides reduce both numbers to about 0.02. Figure 38.25 draws both experiments.

Panel (a) answers the literal reading of the update. The sample travels along the probability-flow trajectory through its noise end, not along the forward curve that produced it. DDIM is the deterministic end of the family of reverse processes (Section 38.5.5), whose members reinject a chosen amount of noise and share the single-time marginals of the forward process, so one trained network serves them all. Section 17.5 chooses among them.

38.6.4.2 DDIM on Fitted Scores

Every sampler so far was handed the exact score. We now learn one and sample with it, on the data that defeated Langevin dynamics: the two standardized modes at $-m$ and $+m$, reweighted so that the left one holds 0.8 of the mass and the right one 0.2.

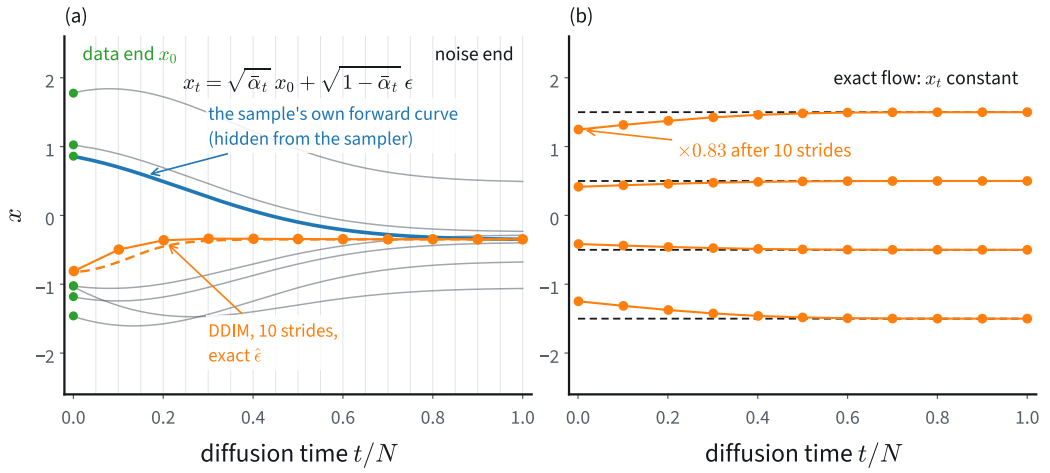


Figure 38.25: (a) Each forward pair (x_0, ϵ) of the standardized mixture owns one curve; the blue pair crossed the valley. From the same noise end, DDIM with the exact noise prediction follows the probability-flow ODE (10 strides, dots; 1,000 strides, dashed) into the other mode: it never sees the sample's own curve. (b) For unit-variance Gaussian data the exact flow is the identity (dashed); ten DDIM strides with the exact noise prediction shrink every sample by 0.83, the Euler error of the Gaussian corollary.

The model is the one that worked at a single level, the 17 fixed features, now fitted separately at each of the 50 levels that a 50-stride DDIM run visits. At each level we noise 4,096 clean draws four times and regress the noise ϵ on the features of the noisy point, evaluated at $\sqrt{4.25}\, x_t$ so that the bumps cover the standardized data as they covered the original. A small weight decay (Section 2.7) handles the bumps that no pair reaches at a given level: least squares would leave their weights arbitrary, as it left the fit at $x = 8$, and the penalty sets them to zero. DDIM then runs twice from the noise of the previous cell, on the exact noise prediction of the lopsided mixture and on the fitted one. If regression at every level learns what a single level could not, both rows will show a light mode of mass 0.2.

```
ws = (0.8, 0.2)                                    # the right mode is the light one
def draw(size):                                    # standardized modes +-m, weights ws
    return rng.normal(np.where(rng.random(size) < ws[0], -m, m), np.sqrt(v0))
x_pairs = np.repeat(draw(4096), 4)                 # four noisy copies per clean point
def fit_eps(a, x_pairs):                           # ridge least squares for eps at one
↪ level
    e = rng.standard_normal(x_pairs.size)
    F = features(np.sqrt(4.25) * (np.sqrt(a) * x_pairs + np.sqrt(1 - a) * e))
    return np.linalg.solve(F.T @ F + 1e-6 * e.size * np.eye(17), F.T @ e)
coef = {a: fit_eps(a, x_pairs) for a in alpha_bar[strides(50)[:-1]]}   # the levels DDIM
↪ visits
x_exact = ddim(z, 50, lambda q, a: eps_hat(q, a, ws))
x_fit = ddim(z, 50, lambda q, a: features(np.sqrt(4.25) * q) @ coef[a])
x_true = draw(8000)
for name, xs in (('exact', x_exact), ('fitted', x_fit)):
    L, R = xs[xs < 0], xs[xs > 0]
```

```
    print(f'{name:6s} scores: light mode {(xs > 0).mean():.3f}, means {L.mean():+.2f} '
          f'{R.mean():+.2f}, widths {L.std():.3f} {R.std():.3f}, KS to p0 {ks(xs,
          ↪ x_true):.2f}')
print(f'truth: light mode 0.2, means {-m:+.2f} {m:+.2f}, widths {np.sqrt(v0):.3f}; '
      f'KS between the clouds {ks(x_exact, x_fit):.2f}')
```

```
exact  scores: light mode 0.195, means -0.96 +0.94, widths 0.228 0.244, KS to p0 0.03
fitted scores: light mode 0.202, means -0.97 +0.94, widths 0.226 0.274, KS to p0 0.02
truth: light mode 0.2, means -0.97 +0.97, widths 0.243; KS between the clouds 0.02
```

They do. Both rows put a fifth of the samples in the light mode, to within a hundredth or two, where Langevin dynamics with the exact score of one level left almost half. The fitted sampler never saw a density, a score or the number 0.2. It saw noisy points and the noise that made them.

The two rows start from the same noise, so what they share is the sampler's error and what separates them is the fit's. They share a light mode a few hundredths short of 0.97 and a heavy mode that is slightly too narrow, about 0.23 against 0.243. With the exact noise prediction the only error is Euler's, so this is the error of fifty strides. It has the size the Gaussian corollary led us to expect, a shrinkage of a few percent, where fifty strides multiplied Gaussian data by 0.964. The fit adds one visible error of its own, a light mode of width 0.26 to 0.28 where the exact row has about 0.24. The two modes differ in nothing but their share of the data, so the excess comes from there. Least squares spends its 17 weights where the pairs are, and only a fifth of the pairs fall in the light mode. The KS statistics agree. Either row is a few hundredths from fresh draws of the data, and the two rows are no further from each other.

A diffusion model does not fit each level separately. Its network takes the level as an input and shares its weights across levels (Exercise 8). The regression, the sampler and the two kinds of error stay the same.

38.6.4.3 Guidance with Bayes' Rule

A sampler for p returns samples of every kind that p contains. Often we want one kind only: images of one class, or images that fit a caption. *Guidance* steers a sampler toward a class y by changing the score it follows. Section 17.4 builds guided samplers and measures them on images. Here we keep the two facts that the mixture lets us check by hand: guidance is Bayes' rule on scores, and with a scale above one the sampler follows densities that no noised data could have.

On the standardized mixture with equal weights, let y be "the right mode", so that we want samples from the class $\mathcal{N}(m, v_0)$ alone, with $m = 0.970$ and $v_0 = 0.243^2$. Bayes' rule at noise level t, $p_t(\mathbf{x} \mid y) \propto p_t(\mathbf{x})\, p_t(y \mid \mathbf{x})$, becomes additive for scores, because the evidence $p(y)$ does not depend on $\mathbf{x}$:

$$\nabla_{\mathbf{x}} \log p_t(\mathbf{x} \mid y) = \nabla_{\mathbf{x}} \log p_t(\mathbf{x}) + \nabla_{\mathbf{x}} \log p_t(y \mid \mathbf{x}). \tag{38.89}$$

The conditional score is the unconditional score plus the gradient of a classifier's log-probability, and every sampler of this section runs unchanged with it. **Classifier guidance** (Dhariwal and Nichol 2021) takes the second term from a classifier $p_\phi(y \mid \mathbf{x}, t)$ of noisy inputs and scales it by a **guidance scale** γ:

$$\tilde{\mathbf{s}}(\mathbf{x}, t) = \nabla \log p_t(\mathbf{x}) + \gamma \nabla \log p_\phi(y \mid \mathbf{x}, t),$$
$$\tilde{\mathbf{s}} = \nabla \log \tilde{p}_t, \qquad \tilde{p}_t(\mathbf{x}) \propto p_t(\mathbf{x})\, p_\phi(y \mid \mathbf{x}, t)^\gamma.$$

With $\gamma = 1$ this is Bayes' rule. With $\gamma > 1$ the guided field is the score of a *tilted* density that raises the classifier's probability to a power. **Classifier-free guidance (CFG)** (Ho and Salimans 2022) reads Equation 38.89 from right to left, $\nabla \log p_t(y \mid \mathbf{x}) = \nabla \log p_t(\mathbf{x} \mid y) - \nabla \log p_t(\mathbf{x})$, and so needs no classifier, only a conditional and an unconditional score model $\mathbf{s}_\theta(\mathbf{x}, t, y)$ and $\mathbf{s}_\theta(\mathbf{x}, t, \emptyset)$, which one network can supply (Section 17.4.4.2):

$$\tilde{\mathbf{s}}(\mathbf{x}, t) = \mathbf{s}_\theta(\mathbf{x}, t, \emptyset) + \gamma \left[\mathbf{s}_\theta(\mathbf{x}, t, y) - \mathbf{s}_\theta(\mathbf{x}, t, \emptyset)\right], \tag{38.90}$$

equivalently $(1 - \gamma)\, \mathbf{s}_\emptyset + \gamma\, \mathbf{s}_y$, which for $\gamma > 1$ extrapolates past the conditional score.

One fact limits the interpretation of $\gamma > 1$. The tilted density $\tilde{p}_t$ is in general not the noised marginal of *any* clean distribution. The marginal p_t is the law of $\sqrt{\bar{\alpha}_t}\, \mathbf{x}_0$ noised at scale σ_t, so by the curvature bound Equation 38.81 no eigenvalue of $-\nabla^2 \log p_t$ exceeds $1/\sigma_t^2$, whatever the data. Raising the classifier to the power γ adds curvature, most of it midway between the classes. For the mixture the bound fails there once $\gamma > 1 + v_0 v_t / (\sigma_t^2 m^2)$, where $v_t = \bar{\alpha}_t v_0 + \sigma_t^2$ is the variance of a noised class (Exercise 6). This threshold is 1.0625 at the noisiest levels and 1.09 at $t = 100$. At $\gamma = 3$ the bound holds only at about ten of the thousand levels, the least noisy ones, and at $\gamma = 10$ at fewer still.

Why should we care? If the tilted densities were the noised marginals of one clean distribution, a reverse chain run on their scores would end on that distribution, as it did for p. They are not, so nothing tells us in advance where a guided chain ends. Let us look. We run the ancestral chain of Section 38.6.3.2 on exact scores, the unconditional score of the mixture and the class-conditional score, that of the single Gaussian $\mathcal{N}(\sqrt{\bar{\alpha}_t}\, m,\ v_t)$, combined by Equation 38.90 at $\gamma \in \{1, 3, 10\}$.

```
def cfg_sample(gamma, n=8000):                          # ancestral chain with the guided score
    xt = rng.standard_normal(n)
    for t in range(N - 1, -1, -1):
        a, b = alpha_bar[t], beta[t]
        v = a * v0 + 1 - a                               # noised component variance
        s_uncond = mixture_score(xt, v, (-np.sqrt(a) * m, np.sqrt(a) * m))
        s_cond = (np.sqrt(a) * m - xt) / v               # class y = the right mode
        xt = (xt + b * ((1 - gamma) * s_uncond + gamma * s_cond)) / np.sqrt(1 - b)
        if t > 0:
            xt = xt + np.sqrt(b) * rng.standard_normal(n)
    return xt
for gamma in (1.0, 3.0, 10.0):
    xs = cfg_sample(gamma)
    print(f'gamma = {gamma:4.1f}: mass right of 0 {(xs > 0).mean():.3f}, '
          f'mean {xs.mean():.3f}, std {xs.std():.3f}')
print(f'exact conditional: mean {m:.3f}, std {np.sqrt(v0):.3f}')
```

```
gamma =  1.0: mass right of 0 1.000, mean 0.969, std 0.247
gamma =  3.0: mass right of 0 1.000, mean 1.031, std 0.231
gamma = 10.0: mass right of 0 1.000, mean 1.077, std 0.233
exact conditional: mean 0.970, std 0.243
```

At $\gamma = 1$ the chain reproduces the class: all the mass lies right of zero, where the unconditional mixture puts only half of it, and the mean and standard deviation are within about 0.01 of the exact 0.970 and 0.243. At $\gamma = 3$ and 10 there is no mass left to move between modes, so the guidance moves the mode itself, to about 1.03 and then to between 1.07 and 1.08, away from the class boundary. The width hardly changes and stays within about 5% of 0.24.

This is where the broken bound shows. At the clean level the classes are eight standard deviations apart and the classifier is certain, so the tilted density there is the class itself, with mean 0.970 for every γ. A chain that followed the marginals of one forward process would end on it. At $\gamma = 3$ and 10 ours ends about 0.06 and 0.1 to its right, because the densities whose scores it follows do not belong to one process. The shift is produced at intermediate levels, where the noised classes overlap (Exercise 5). Guidance with $\gamma > 1$ thus moves samples away from the other class, beyond where the class itself sits. A class with a single Gaussian mode has little diversity to lose; on images the loss is measurable (Section 17.4.4.2).

Guided or not, every sampler of this section evaluates the score many times, because its paths curve. The Gaussian corollary showed that even the simplest data bend them in DDIM's clock. We took the forward process for its Gaussian kernels and never asked whether its paths are easy to follow. Which paths from noise to data are straight?

38.6.5 Summary

A score needs no normalizing constant, and for noised data it is the conditional mean of a target we know for every training pair, $-\epsilon/\sigma$. Least squares on noisy pairs therefore learns it from samples alone, and its loss ends at the posterior-variance floor, not at zero. One noise level is not enough: far from the data the fit is arbitrary, and Langevin dynamics keeps the proportions of its start, with almost half of its samples in a mode that holds a fifth. With scores at every level of a forward process, the two best-known samplers discretize equations we already had. DDPM is the variance-preserving SDE observed at discrete times, with denoising score matching weighted by σ_t^2 as its loss. DDIM is Euler's method for the probability-flow ODE in the coordinates $(\bar{\sigma}, \bar{\mathbf{x}})$, exact for a point mass and too contractive for Gaussian data. Run on scores fitted by least squares at every level, DDIM returned the weights 0.8 and 0.2 that a single level could not. Guidance is Bayes' rule on scores, and for $\gamma > 1$ it follows densities that curve more sharply than any noised density can.

38.6.6 Exercises

1. code **Langevin discretization bias.** For the Gaussian target $p = \mathcal{N}(0, v)$, the Langevin sampler of Equation 38.82 with step h is the linear recursion $x \leftarrow (1 - \frac{h}{2v})\, x + \sqrt{h}\, \xi$.
    1. Find the step sizes for which the recursion has a stationary distribution, compute its variance, and expand the bias to first order in h.
    2. Treat each mode of the mixture in the cell that defines `langevin` as a separate Gaussian with $v = 0.25$, and predict the stationary $\mathbb{E}[X^2]$ at $h \in \{0.01, 0.1, 0.25, 0.5\}$. Compare each predicted bias with the Monte Carlo standard error of an estimate from 10,000 chains. Then run `langevin` for 2,000 steps at each of these step sizes from the initial points `rng.normal(0.0, 3.0, 10000)` of `warm`. From which step size on does the experiment resolve the bias?

3. The Metropolis-adjusted Langevin algorithm (Roberts and Tweedie 1996) removes the bias with an accept–reject test that needs the density ratio $p(\mathbf{x}')/p(\mathbf{x})$. Why can a sampler that has only a trained score network $\mathbf{s}_\theta$ not apply this test directly?

2. code **Annealing out of one mode.** In the cell that defines `langevin`, chains started in one mode of $0.8\,\mathcal{N}(-2, 0.5^2) + 0.2\,\mathcal{N}(2, 0.5^2)$ stay there, and the broad start leaves almost half of the mass on the right. Annealed Langevin dynamics first runs the sampler on smoothed densities p_σ, here the mixture with component variance $0.25 + \sigma^2$, for the decreasing sequence of noise levels $\sigma = 2, 1, 0.5, 0.25, 0.1, 0$.

   1. With equal weights, p_σ has no valley for $\sigma \geq 1.94$, as the text found. With the weights 0.8 and 0.2 the valley closes sooner. Count the sign changes of `mixture_score` with `ws=(0.8, 0.2)` on a fine grid to find the smallest σ without a valley. Which levels of the sequence have no valley?
   2. Run 300 steps of `langevin` at each level of the sequence, passing the component variance $0.25 + \sigma^2$ as `var`, for 10,000 chains started at -2 and again for 10,000 chains started at $+2$. Do this once with the fixed step $h = 0.01$ and once with $h = 0.04\,(0.25 + \sigma^2)$, which equals 0.01 at $\sigma = 0$. Report the fraction of chains with $X > 0$ at the end of each of the four runs. Which step rule forgets the start? Its fraction comes close to the target 0.2 without reaching it. Compute the mass of p_σ to the right of zero at each level, and say at which level the proportions were fixed.
   3. Song and Ermon (2019) scale the step size of each level with σ^2. Use the contraction factor of the linear recursion in Exercise 1 to explain why the fixed step fails at the large noise levels.

3. code **The Bayes-risk floor.** ● The cell that fits `w_ls` prints the mean squared error of the fit and of the exact score, both averages over the same 16,384 noisy pairs, and the population floor by quadrature.

   1. The text obtains the floor Equation 38.80, the Bayes risk of the regression, from the lemma with $\mathbf{v} = \mathbf{0}$. Use Tweedie's formula Equation 38.79 to show that it also equals $\mathbb{E}\,\operatorname{tr}\operatorname{Cov}[\mathbf{x} \mid \tilde{\mathbf{x}}]/\sigma^4$, and check that the two expressions agree for $\mathbf{x} \sim \mathcal{N}(\mathbf{0}, v\mathbf{I})$. What does the floor do as $\sigma \to 0$ for data with a smooth density, and what for a training set of n distinct points? What does the optimal denoiser of the training set return in that limit?
   2. The printed exact-score average is not the floor. Compute its standard error twice: from the 16,384 per-pair squared errors as if they were independent, and from the 4,096 averages over the four noisy copies of each clean point. Why do the two differ, and which one agrees with `errs.std()`, the standard deviation over twenty fresh draws that the text prints? Is the printed value consistent with the population floor? Then explain why the fit and the exact score, evaluated on the same pairs, differ from each other by far less than this standard error.

4. **Weightings across noise levels.** ● Consider the VP-SDE with the linear schedule $\beta(t) = \beta_{\min} + t\,(\beta_{\max} - \beta_{\min})$ on $[0, 1]$, with $\beta_{\min} = 0.1$ and $\beta_{\max} = 20$, so that $\bar{\alpha}(t) = \exp\big(-\beta_{\min} t - \tfrac{1}{2}(\beta_{\max} - \beta_{\min})t^2\big)$ and $\sigma_t^2 = 1 - \bar{\alpha}(t)$. Let the data in $\mathbb{R}^d$ have a smooth density, and assume that $\mathbb{E}\|\nabla \log p_t\|^2$ stays bounded as $t \to 0$. Write $R(t)$ for the Bayes risk of the noise-conditional DSM loss Equation 38.83 at level t, the floor Equation 38.80 with σ_t in place of σ.

   1. Show that $R(t) \approx d/(\beta_{\min} t)$ as $t \to 0$.

2. Find the behavior of $w(t)\,R(t)$ as $t \to 0$ for $w = 1$, for the DDPM weighting $w = \sigma_t^2$, and for the likelihood weighting $w = g^2 = \beta(t)$. For which weightings is $\int_0^1 w(t)\,R(t)\,dt$ finite, and which noise levels does each weighting emphasize?
3. Training draws t uniformly from $[\varepsilon, 1]$. For the likelihood weighting, show that the second moment of $w(t)\,R(t)$ under this draw grows like $1/\varepsilon$. Y. Song, Durkan, et al. (2021) (Section 5.1) instead draw t from the density proportional to $\beta(t)/\sigma_t^2$ on $[\varepsilon, 1]$ and multiply the DSM term at the drawn level by $c\,\sigma_t^2$, with the constant $c = \frac{1}{1-\varepsilon}\int_\varepsilon^1 \beta(r)/\sigma_r^2\,dr$. Show that this estimate is unbiased for the likelihood-weighted loss and that its value at the optimum, $c\,\sigma_t^2\,R(t)$, stays bounded as $t \to 0$.

5. code **Guidance by noise level.** ● In the cell that defines `cfg_sample`, raising γ from 1 to 10 moves the mean of the guided samples from about 0.97 to between 1.07 and 1.08.
   1. The text claims that at the clean level the tilted density is the class itself. Check it. At the last level of the chain, $t = 0$ in `cfg_sample`, the class posterior of the right mode is a logistic function of x. Write it, and compute by quadrature the mean of the tilted density proportional to $p_t(x)\,p_t(y \mid x)^\gamma$ at that level for $\gamma \in \{1, 3, 10\}$. How far is each mean from m, and how far are the printed means?
   2. Modify `cfg_sample` so that it uses $\gamma = 10$ only on the steps $t \geq 500$, only on $200 \leq t < 500$, or only on $t < 200$, and $\gamma = 1$ on the others. Which band of noise levels produces the shift? Relate the answer to the separation $\sqrt{\bar\alpha_t}\,m/\sqrt{v_t}$ of the two noised classes, where m and v_0 are the variables `m` and `v0`, and $v_t = \bar\alpha_t\,v_0 + 1 - \bar\alpha_t$ is the noised component variance, the variable `v` of `cfg_sample`.
   3. Kynkäänniemi et al. (2024) find on image models that guidance is harmful at high noise levels, largely unnecessary at low noise levels, and beneficial in between. Which parts of this pattern does the one-dimensional example reproduce, and which can it not exhibit?

6. **Guided densities and noised marginals.** ● The guided field Equation 38.90 with $\gamma > 1$ is the score of the tilted density $\tilde p_t(\mathbf{x}) \propto p_t(\mathbf{x})\,p_t(y \mid \mathbf{x})^\gamma$. Call a density a noised marginal at level t if it is the law of $\sqrt{\bar\alpha_t}\,\mathbf{x}_0 + \sigma_t\epsilon$, with $\sigma_t^2 = 1 - \bar\alpha_t$, for some clean distribution of $\mathbf{x}_0$.
   1. Apply Equation 38.81 to the clean variable $\sqrt{\bar\alpha_t}\,\mathbf{x}_0$ to show that a noised marginal p_t has

$$\nabla^2 \log p_t(\mathbf{x}) = -\frac{\mathbf{I}}{\sigma_t^2} + \frac{\bar\alpha_t}{\sigma_t^4}\mathrm{Cov}\left[\mathbf{x}_0 \mid \mathbf{x}_t = \mathbf{x}\right].$$

      Conclude that no eigenvalue of $-\nabla^2 \log p_t$ exceeds $1/\sigma_t^2$, whatever the clean distribution.

   2. In `cfg_sample`, the two noised classes are $\mathcal{N}(\pm\sqrt{\bar\alpha_t}\,m, v_t)$, where m and v_0 are the variables `m` and `v0`, and $v_t = \bar\alpha_t\,v_0 + \sigma_t^2$ is the noised component variance. The label y names the right class. Show that $-(\log \tilde p_t)''(x) = 1/v_t + (\gamma - 1)\,a_t^2\,\varsigma(a_t x)\,(1 - \varsigma(a_t x))$, where $a_t = 2\sqrt{\bar\alpha_t}\,m/v_t$ and $\varsigma(u) = 1/(1 + e^{-u})$ is the logistic function.
   3. Find the smallest γ for which $\tilde p_t$ violates the bound of

(a) at some x. Evaluate it at the levels $t = 0$, 100, and 500 of `alpha_bar`, and find its limit as $t \to N$. What do the values say about the densities that the chain of `cfg_sample` follows at $\gamma = 3$?

7. **Score error and likelihood.** ●● With the likelihood weighting $w = g^2$, the population score loss bounds the Kullback–Leibler divergence from the data to the model, by Theorem 1 of Y. Song, Durkan, et al. (2021) . The paper proves the bound with path measures; this problem derives it from the Fokker–Planck equation of Section 38.5.2, following C. Lu, Zheng, et al. (2022), who take this route for the SDE (their Proposition C.1) and for the probability-flow ODE (their Theorem 3.1). Let the forward SDE $d\mathbf{x} = \mathbf{f}(\mathbf{x}, t)\, dt + g(t)\, d\mathbf{W}$ have marginals p_t on $[0, T]$. In reverse time $\bar{t} = T - t$, the exact reversal and the model are

$$\begin{aligned} d\mathbf{x} &= \big[-\mathbf{f} + g^2\, \nabla \log p_t\big]\, d\bar{t} + g\, d\mathbf{B}_{\bar{t}}, \\ d\mathbf{x} &= \big[-\mathbf{f} + g^2\, \mathbf{s}_\theta(\mathbf{x}, t)\big]\, d\bar{t} + g\, d\mathbf{B}_{\bar{t}}, \end{aligned} \tag{38.91}$$

evaluated at $t = T - \bar{t}$. The first has marginals p_t; the second, started from a prior π at $\bar{t} = 0$, has marginals q_t. Assume that all densities are smooth and positive and decay fast enough for every integration by parts.

1. Write both Fokker–Planck equations in $\bar{t}$ in the form $\partial_{\bar{t}} \rho = -\nabla \cdot (\rho\, \mathbf{u})$, and identify the velocities $\mathbf{u}_p$ and $\mathbf{u}_q$.
2. Show that $\frac{d}{d\bar{t}} D_{\mathrm{KL}}(p_t \,\|\, q_t) = \mathbb{E}_{p_t}\big[(\mathbf{u}_p - \mathbf{u}_q)^{\top} \nabla \log(p_t / q_t)\big]$.
3. Bound the right-hand side by $\frac{1}{2} g^2\, \mathbb{E}_{p_t} \|\mathbf{s}_\theta - \nabla \log p_t\|^2$, integrate over $\bar{t}$, and obtain

$$\begin{aligned} D_{\mathrm{KL}}(p_0 \,\|\, q_0) \le\ & D_{\mathrm{KL}}(p_T \,\|\, \pi) \\ & + \frac{1}{2} \int_0^T g(t)^2\, \mathbb{E}_{p_t} \big\|\mathbf{s}_\theta(\mathbf{x}, t) - \nabla \log p_t(\mathbf{x})\big\|^2\, dt. \end{aligned} \tag{38.92}$$

   Under which condition on q_t is the bound of the derivative tight?

4. Repeat (a) and (b) for the probability-flow ODE with $\mathbf{s}_\theta$ in place of the score. Which term that made (c) possible is missing, and why does the score loss alone no longer bound $D_{\mathrm{KL}}(p_0 \,\|\, q_0)$?

8. [extended] **Learned noise prediction.** ●● The cell that defines `fit_eps` learns the noise prediction with 17 fixed features and a separate fit at each of 50 levels. A diffusion model has one network for all levels. Build it for the same data, the draws `draw(size)` of the lopsided standardized mixture, under the DDPM schedule `beta` and `alpha_bar`.

1. In the framework of your tab, train a noise-prediction network $\epsilon_\theta(x_t, t)$ with input $(x_t, t/1000)$ and two hidden layers of 64 tanh units, with Adam at learning rate 3×10^{-3} for 4,000 steps on batches of 256 fresh draws, on the simple loss with t uniform over the 1,000 levels. Compute its root-mean-square error against the exact noise prediction `eps_hat(xt, a, ws)` on samples of p_t at several levels t, and do the same for the fitted features of the text at the levels they were fitted for. Where is each error largest?

2. Sample with `ddim` from the noise `z` at $K = 10$ and $K = 50$, and with the ancestral step of Section 38.6.3.2, the loop of `cfg_sample` with the network's score in place of the guided one. Compare each set of samples with `x_exact`, `x_fit` and `x_true` by the mass of the light mode, the widths of the two modes, and `ks`. Which sampler is most sensitive to the error of (a), and at which step of that sampler does the error enter?
3. Retrain with the per-level weights that correspond to $w = 1$ and to the likelihood weighting $w = \beta$, that is, with the noise loss at level t multiplied by $w(t)/(1-\bar{\alpha}_t)$, and repeat (a) and (b). Exercise 4 predicts which levels each weighting serves: do the errors agree?

The deliverable is a table of the statistics of (b) for each weighting and sampler, together with a plot of the error of (a) against t.

38.7 Flow Matching and Optimal Transport

The samplers of the previous section call the score once per step, a thousand times for the reverse chain and a few dozen for its deterministic strides. They need so many steps because the paths from noise to data curve, and a solver can follow a curved path only in short pieces. Nobody designed those paths. They came with the noising process. Could we choose the path from noise to data ourselves, and choose one along which a solver can take long steps?

Picture two data points at -2 and $+2$ and a cloud of Gaussian noise that must end on them. If every particle moved in a straight line at constant speed toward its own point, one Euler step of length one would carry it there exactly. But a velocity field is one function of position and time. Particles bound for different points pass through the same place, and the field there can point only one way.

Flow matching (Lipman et al. 2023) and rectified flow (Liu et al. 2023) do choose: they prescribe the path and learn only the velocity that moves mass along it. Which paths are cheapest is a much older question, the problem of *optimal transport*. Gaspard Monge asked it in 1781 about moving a pile of earth into a hole, and under squared distance the cheapest motion turns out to be the one we just wished for, every particle on a straight line at constant speed (McCann 1997; Benamou and Brenier 2000).

One regression trains the velocity of any path assembled from simple pieces, because that velocity is an average over particles and least squares learns averages (Section 38.7.1). Straight pieces still give a curved flow. Re-pairing noise with data straightens it (Section 38.7.2), and optimal transport says what the re-pairing aims at (Section 38.7.3). Two parts can wait for a second reading: the conversions between score, noise and velocity that follow the definition of the Gaussian path in Section 38.7.1.3, and the solver comparison of Section 38.7.4.

> **!** One clock
>
> Time runs as in the previous section, from the data at $t = 0$ to the noise, now at $t = 1$. This holds for scores and velocities alike: training noises the data forward in time, and every sampler integrates from $t = 1$ down to $t = 0$.
> The flow-matching papers cited in this section, and their code, run time the other way, from noise at 0 to data at 1 (Figure 38.26). To compare a formula with theirs, substitute

$1 - t$ for their t, read their noise $\mathbf{x}_0$ as our ϵ and their data $\mathbf{x}_1$ as our $\mathbf{x}_0$, and flip the sign of every velocity.

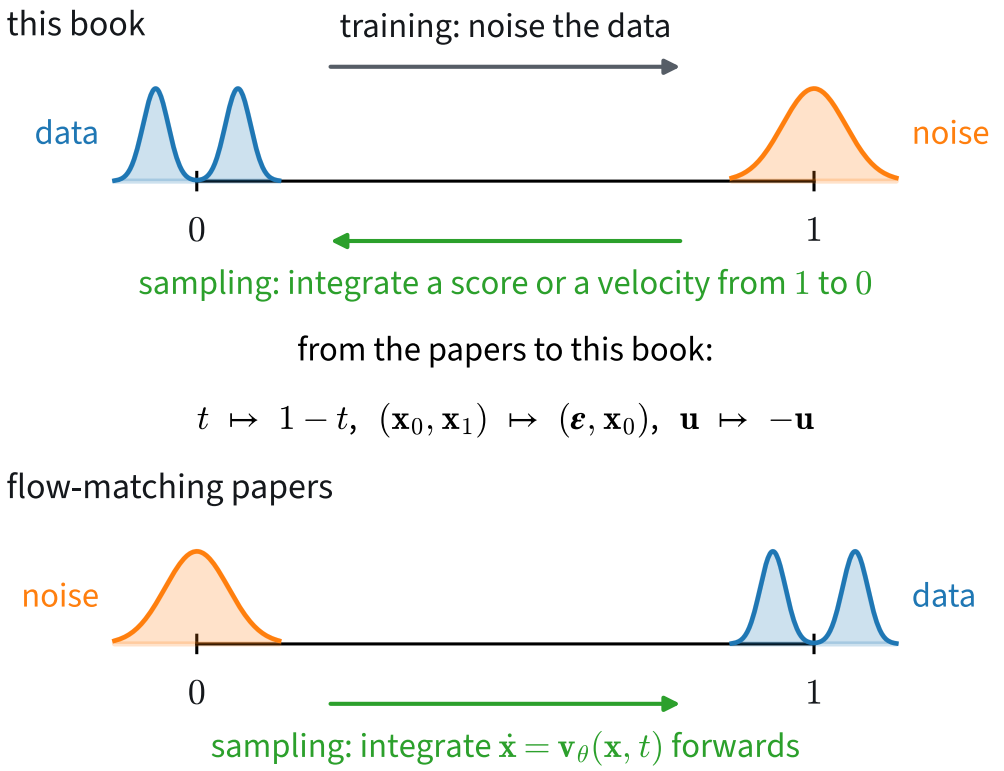


Figure 38.26: One clock and the papers' clock. This book (top) puts the data at $t = 0$ and the noise at $t = 1$, noises the data forward in time, and samples by integrating a score or a velocity from 1 to 0. The flow-matching papers (bottom) put the noise at 0 and the data at 1 and sample forward. The endpoint densities are the same; the substitution $t \mapsto 1 - t$ converts one clock into the other and flips the sign of every velocity.

The data are $\mathbf{x}_0 \sim p_0$, and p_t is a path of densities from p_0 to the noise $p_1 = \mathcal{N}(\mathbf{0}, \mathbf{I})$. A sample from the noise end has two names: ϵ when it is Gaussian noise, as in the previous section, and $\mathbf{x}_1$ in statements about pairings, which hold for any two endpoint laws.

38.7.1 Flow Matching

38.7.1.1 Probability Paths and Velocity Fields

Diffusion got its path from a noising process and then had to learn the score to run that process backward. Here the path is ours to choose. A **probability path** is a family of densities $(p_t)_{t\in[0,1]}$ with p_0 the data and p_1 the noise. A velocity field $\mathbf{u}_t$ *realizes* the path if the pair satisfies the continuity equation of Section 38.5.4, which says that a density moved by a velocity field changes by the net inflow of mass,

$$\partial_t p_t + \nabla \cdot (p_t\, \mathbf{u}_t) = 0.$$

Then, for a Lipschitz field, integrating $\dot{\mathbf{x}} = \mathbf{u}_t(\mathbf{x})$ from a sample of p_1 at $t = 1$ down to $t = 0$ yields a sample of p_t at every time, and of the data at $t = 0$. Note the direction. The field points the way the noising goes, from data toward noise, and a sampler steps against it. If a network $\mathbf{u}_\theta(\mathbf{x}, t)$ approximates $\mathbf{u}_t$, generation is an ODE solve from noise. Such a model is a continuous normalizing flow (Section 38.3.3) run backward in time: its flow map from $t = 1$ to $t = 0$ is the generator. Flow matching trains it without solving the ODE even once. The natural objective is the **flow-matching loss**

$$\mathcal{L}_{\mathrm{FM}}(\theta) = \mathbb{E}_{t\sim\mathcal{U}[0,1],\ \mathbf{x}\sim p_t}\left[\,\left\|\mathbf{u}_\theta(\mathbf{x}, t) - \mathbf{u}_t(\mathbf{x})\right\|^2\right]. \tag{38.93}$$

It is the mean squared error of the network against the true velocity, averaged over time and over the cloud. As it stands it is of no use, because it regresses on a field we cannot evaluate: $\mathbf{u}_t$ depends on the whole data distribution.

We therefore build the path out of pieces we can write down. Let a conditioning variable $\mathbf{z}$, drawn from a law q, be a data point, or a pair of a noise point and a data point. For each value of $\mathbf{z}$, pick a small moving blob $p_t(\mathbf{x} \mid \mathbf{z})$ that sits on the data point at $t = 0$ and ends in the noise at $t = 1$: spread over all of it if $\mathbf{z}$ is a data point, on its own noise point if $\mathbf{z}$ is a pair. Each blob comes with a velocity $\mathbf{u}_t(\mathbf{x} \mid \mathbf{z})$ that we know in closed form. The path is the mixture of the blobs, $p_t(\mathbf{x}) = \int p_t(\mathbf{x} \mid \mathbf{z})\, q(\mathbf{z})\, d\mathbf{z}$, and we write $\mathbf{x}_t$ for a particle of it: draw $\mathbf{z}$ from q, then a point of its blob at time t. Which velocity moves the mixture? Everything in this section rests on the answer.

Proposition (the marginal velocity is a posterior mean). *Suppose that for each* $\mathbf{z}$ *the density* $p_t(\cdot \mid \mathbf{z})$ *is positive and continuously differentiable in* $(\mathbf{x}, t)$*, that it satisfies the continuity equation* $\partial_t p_t(\mathbf{x} \mid \mathbf{z}) = -\nabla \cdot (p_t(\mathbf{x} \mid \mathbf{z})\, \mathbf{u}_t(\mathbf{x} \mid \mathbf{z}))$ *with a continuously differentiable velocity, and that the integrals below may be differentiated under the integral sign. Define on* $\{p_t > 0\}$ *the* **marginal velocity**

$$\mathbf{u}_t(\mathbf{x}) = \mathbb{E}\left[\mathbf{u}_t(\mathbf{x} \mid \mathbf{z}) \mid \mathbf{x}_t = \mathbf{x}\right] = \int \mathbf{u}_t(\mathbf{x} \mid \mathbf{z})\, \frac{p_t(\mathbf{x} \mid \mathbf{z})\, q(\mathbf{z})}{p_t(\mathbf{x})}\, d\mathbf{z}. \tag{38.94}$$

Then $(p_t, \mathbf{u}_t)$ *satisfies the continuity equation.*

The velocity of the cloud at $\mathbf{x}$ is the average of the velocities of the blobs that could have put a particle at $\mathbf{x}$, each weighted by how likely it is to have done so.

Proof. Differentiate the mixture under the integral sign, substitute the conditional continuity equation, and pull the divergence out of the integral:

$$\begin{aligned}\partial_t p_t(\mathbf{x}) &= \int \partial_t p_t(\mathbf{x} \mid \mathbf{z})\, q(\mathbf{z})\, d\mathbf{z} \\ &= -\nabla \cdot \int p_t(\mathbf{x} \mid \mathbf{z})\, \mathbf{u}_t(\mathbf{x} \mid \mathbf{z})\, q(\mathbf{z})\, d\mathbf{z} = -\nabla \cdot (p_t(\mathbf{x})\, \mathbf{u}_t(\mathbf{x}))\,.\end{aligned}$$

The last step multiplies and divides by $p_t(\mathbf{x})$ and recognizes Equation 38.94. ■

Where only one blob passes, the marginal velocity is that blob's velocity. Where several pass, it is their average, and this averaging is what will bend the flow.

38.7.1.2 The Conditional Flow Matching Theorem

The target of Equation 38.93 is the posterior mean Equation 38.94 of a quantity we can compute for every sample. By the regression lemma (Section 38.6.1.2), least squares against a per-sample target fits its conditional mean. We may therefore regress on the per-sample quantity, which is what the **conditional flow matching** (CFM) loss does:

$$\mathcal{L}_{\text{CFM}}(\theta) = \mathbb{E}_{t,\, \mathbf{z}\sim q,\, \mathbf{x}\sim p_t(\cdot|\mathbf{z})} \left[\left\| \mathbf{u}_\theta(\mathbf{x}, t) - \mathbf{u}_t(\mathbf{x} \mid \mathbf{z}) \right\|^2 \right]. \tag{38.95}$$

It needs only a sample of $(t, \mathbf{z}, \mathbf{x})$ and the closed-form conditional velocity.

Theorem (CFM trains the marginal field). *Let $\mathbf{u}_t$ be the marginal velocity Equation 38.94 and suppose that $\mathbb{E}\|\mathbf{u}_t(\mathbf{x} \mid \mathbf{z})\|^2 < \infty$. Then*

$$\mathcal{L}_{\text{CFM}}(\theta) = \mathcal{L}_{\text{FM}}(\theta) + C$$

with C independent of θ. In particular the two objectives have the same gradients and the same minimizers (Lipman et al. 2023; Tong et al. 2024).

Proof. Apply the regression lemma with $X = (\mathbf{x}, t)$, where $\mathbf{x} \sim p_t(\cdot \mid \mathbf{z})$, and $Y = \mathbf{u}_t(\mathbf{x} \mid \mathbf{z})$, which is square-integrable by hypothesis. The law of X is $t \sim \mathcal{U}[0,1]$ and $\mathbf{x} \sim p_t$, and the conditional mean of Y given X is the marginal velocity, by its definition Equation 38.94. The lemma splits $\mathcal{L}_{\text{CFM}}$ into $\mathbb{E}\|\mathbf{u}_\theta(X) - \mathbf{u}_t(\mathbf{x})\|^2 = \mathcal{L}_{\text{FM}}$ and $C = \mathbb{E}\|Y - \mathbf{u}_t(\mathbf{x})\|^2$, which does not involve θ. ■

The proof is one line because the lemma does the work, and the theorem needs no hypothesis on the path beyond a square-integrable target. It says only that the regression has the right minimizer. That this minimizer carries p_1 to p_0, when integrated from $t = 1$ to $t = 0$, is what the proposition before it guarantees. The constant C is the conditional variance of the target. It measures how much the per-sample velocities through a point disagree, so the CFM loss of a perfect field equals C, not zero. We call C the *floor* of the loss. Denoising score matching has the same structure, with the score of the noising kernel as the per-sample quantity and the marginal score as its posterior mean.

No solver appears in this loss, and no trajectory is integrated during training. The model of Section 38.3.1 that was trained through ten Euler steps hit its targets with ten steps and missed them when it was integrated more finely: it had fitted a program of ten steps. What this regression fits is a field, and the number of steps is chosen afterwards.

38.7.1.3 Score, Noise, and Velocity

The marginal-velocity proposition asks for blobs with known velocities, and so far we have named none. The first family is one we already own. The Gaussian paths of the previous section are probability paths of this kind, and on them the velocity is not a new object. Suppose a network has been trained to predict the noise, as diffusion models are. Do we need a second training run to obtain a velocity? We do not. The score, the noise, the clean point and the velocity are affine functions of one posterior mean, and a network that has learned the noise has learned the other three.

Condition on a data point, $\mathbf{z} = \mathbf{x}_0$, and take the path

$$\mathbf{x}_t = \alpha_t\, \mathbf{x}_0 + \sigma_t\, \epsilon, \qquad \epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I}), \tag{38.96}$$

with smooth coefficients (α_t, σ_t), $(\alpha_0, \sigma_0) = (1, 0)$ and $(\alpha_1, \sigma_1) = (0, 1)$. The blob around $\alpha_t \mathbf{x}_0$ has width σ_t: at $t = 0$ it is the data point, at $t = 1$ the noise. This is the noising kernel of the previous section, with the noise end placed at $t = 1$.

Let us start with the smallest case, Gaussian data $x_0 \sim \mathcal{N}(0, s_0^2)$ in one dimension on the cosine path $\alpha_t = \cos \frac{\pi t}{2}$, $\sigma_t = \sin \frac{\pi t}{2}$. The marginal is $\mathcal{N}(0, s_t^2)$ with $s_t^2 = \alpha_t^2 s_0^2 + \sigma_t^2$, the score is $-x/s_t^2$, and the velocity is $u_t(x) = (\dot{s}_t/s_t)\, x$: a Gaussian cloud only stretches or shrinks (Exercise 3). With $s_0 = 2$ at $t = \frac{1}{2}$ we get $s_t^2 = 2.5$, and a particle at x has velocity $-0.94\, x$. The field points inward because noising shrinks this cloud, from width 2 at $t = 0$ to width 1 at $t = 1$. A sampler runs against it and stretches the cloud back to width 2. With $s_0 = 1$ every p_t is the standard Gaussian and the cloud never moves: $u \equiv 0$.

Proposition (score–velocity identity). *For the Gaussian path Equation 38.96, at any t with $\alpha_t, \sigma_t > 0$ and on $\{p_t > 0\}$, the marginal velocity is an affine function of the marginal score:*

$$\mathbf{u}_t(\mathbf{x}) = \frac{\dot{\alpha}_t}{\alpha_t}\, \mathbf{x} - \left(\sigma_t \dot{\sigma}_t - \sigma_t^2 \frac{\dot{\alpha}_t}{\alpha_t} \right) \nabla \log p_t(\mathbf{x}). \tag{38.97}$$

The velocity is a multiple of the position plus a multiple of the score, and the schedule alone fixes both multiples. The data enter only through the score. The bracket in front of the score equals $\sigma_t^2 \frac{d}{dt} \log(\sigma_t/\alpha_t)$, which is positive whenever the ratio of noise to signal is growing, as it is on every path we use. The identity can then be solved for the score, and each field determines the other.

Proof. We show that both sides are affine in one posterior mean, $\hat{\mathbf{x}}_0 = \mathbb{E}[\mathbf{x}_0 \mid \mathbf{x}_t = \mathbf{x}]$. Differentiating the path gives the conditional velocity $\dot{\mathbf{x}}_t = \dot{\alpha}_t \mathbf{x}_0 + \dot{\sigma}_t \epsilon$, and $\epsilon = (\mathbf{x} - \alpha_t \mathbf{x}_0)/\sigma_t$. Taking the posterior mean as in Equation 38.94,

$$\mathbf{u}_t(\mathbf{x}) = \frac{\dot{\sigma}_t}{\sigma_t}\, \mathbf{x} + \left(\dot{\alpha}_t - \alpha_t \frac{\dot{\sigma}_t}{\sigma_t} \right) \hat{\mathbf{x}}_0.$$

The marginal score is the posterior mean of the scores of the kernels, which is Tweedie's formula of the previous section. The kernel $\mathcal{N}(\alpha_t \mathbf{x}_0, \sigma_t^2 \mathbf{I})$ has score $(\alpha_t \mathbf{x}_0 - \mathbf{x})/\sigma_t^2$, so

$$\nabla \log p_t(\mathbf{x}) = \mathbb{E}\left[\frac{\alpha_t \mathbf{x}_0 - \mathbf{x}}{\sigma_t^2} \,\middle|\, \mathbf{x}_t = \mathbf{x} \right] = \frac{\alpha_t \hat{\mathbf{x}}_0 - \mathbf{x}}{\sigma_t^2},$$

$$\hat{\mathbf{x}}_0 = \frac{\mathbf{x} + \sigma_t^2 \nabla \log p_t(\mathbf{x})}{\alpha_t}.$$

Substituting this $\hat{\mathbf{x}}_0$ into the velocity and collecting the $\mathbf{x}$ and $\nabla \log p_t$ terms gives Equation 38.97. ■

Two special cases are worth writing out. On a variance-preserving path, $\alpha_t^2 + \sigma_t^2 = 1$, so $\sigma_t \dot{\sigma}_t = -\alpha_t \dot{\alpha}_t$ and the coefficient of the score collapses to $\dot{\alpha}_t/\alpha_t$: $\mathbf{u}_t(\mathbf{x}) = \frac{\dot{\alpha}_t}{\alpha_t}(\mathbf{x} + \nabla \log p_t(\mathbf{x}))$. For the variance-preserving SDE of the previous section, $\alpha_t = \exp\big(-\frac{1}{2}\int_0^t \beta(s)\, ds\big)$, so $\dot{\alpha}_t/\alpha_t = -\frac{1}{2}\beta(t)$ and the velocity is $-\frac{1}{2}\beta(t)(\mathbf{x} + \nabla \log p_t(\mathbf{x}))$. This is the probability-flow velocity $\mathbf{f} - \frac{1}{2} g^2 \nabla \log p_t$ (Section 38.5.4) with $\mathbf{f} = -\frac{1}{2}\beta\, \mathbf{x}$ and $g^2 = \beta$. On this path the flow-matching velocity and the probability-flow ODE are the same field on

the same clock. The cosine path of the Gaussian example is variance-preserving too, so we can check the form on it: $\nabla \log p_t = -x/s_t^2$ gives $\frac{\dot\alpha_t}{\alpha_t}(1 - 1/s_t^2)\, x = \alpha_t\dot\alpha_t(s_0^2 - 1)\, x/s_t^2$, the stretching rate $\dot s_t/s_t$ found above.

On the straight path $\alpha_t = 1-t$, $\sigma_t = t$, which the rest of the section uses, the identity reads $\mathbf{u}_t(\mathbf{x}) = -(\mathbf{x} + t\,\nabla \log p_t(\mathbf{x}))/(1-t)$.

The same posterior mean underlies the targets that networks are trained on in practice (Table 38.2).

Table 38.2: Network targets on a Gaussian path. Each row gives the quantity a network predicts, the posterior mean it estimates, and its conversion from the score. The name **v**-prediction comes from the literature; on the cosine path its target $\alpha_t\epsilon - \sigma_t\mathbf{x}_0$ is $2/\pi$ times the velocity target.

Network predicts	Posterior mean	From the score $\mathbf{s} = \nabla \log p_t$
score $\mathbf{s}_\theta$	$\nabla \log p_t(\mathbf{x})$	$\mathbf{s}$
noise $\hat\epsilon$	$\mathbb{E}[\epsilon \mid \mathbf{x}_t = \mathbf{x}]$	$-\sigma_t\,\mathbf{s}$
clean point $\hat{\mathbf{x}}_0$	$\mathbb{E}[\mathbf{x}_0 \mid \mathbf{x}_t = \mathbf{x}]$	$(\mathbf{x} + \sigma_t^2\,\mathbf{s})/\alpha_t$
velocity $\mathbf{u}_\theta$	$\mathbb{E}[\dot{\mathbf{x}}_t \mid \mathbf{x}_t = \mathbf{x}]$	the identity Equation 38.97
v-prediction $\hat{\mathbf{v}}$	$\mathbb{E}[\alpha_t\epsilon - \sigma_t\mathbf{x}_0 \mid \mathbf{x}_t = \mathbf{x}]$	$-\frac{\sigma_t}{\alpha_t}\,\mathbf{x} - \sigma_t\,\frac{\alpha_t^2+\sigma_t^2}{\alpha_t}\,\mathbf{s}$

Each row is a t-dependent affine function of the same posterior mean, so a network trained on any row defines all of them. Which row is easiest to learn is a separate question, about the variance of the training targets at the two ends of the path (Exercise 6). A noise predictor is therefore also a velocity field: the samplers of this section apply to it after the conversion of Table 38.2.

38.7.1.4 Example: Two Points in One Dimension

Before choosing among paths we work one case to the end. For the two-point data set of the opening every object so far has a closed form, and the closed form shows the failure that Section 38.7.2 repairs. The data are $x_0 = \pm a$ with probability $\frac{1}{2}$ each, with $a = 2$: the two modes of the chapter's running mixture with their width removed. The noise is $\epsilon \sim \mathcal{N}(0,1)$, drawn independently of x_0. We take the Gaussian path with $\alpha_t = 1 - t$ and $\sigma_t = t$, which joins each data point to its noise point by a straight line at constant speed: $x_t = (1-t)\,x_0 + t\,\epsilon$, with velocity $\dot x_t = \epsilon - x_0$.

Take the data point as the conditioning variable, $\mathbf{z} = x_0$. Given x_0, the position x_t is $\mathcal{N}((1-t)\,x_0, t^2)$, a positive density for $t > 0$, so the marginal-velocity proposition applies. Eliminating $\epsilon = (x - (1-t)\,x_0)/t$ expresses the velocity as a function of the current position, $u_t(x \mid x_0) = (x - x_0)/t$. Run backward in time, the particle heads for its data point at the speed that arrives exactly at $t = 0$.

To average this velocity we need the posterior of x_0 at a point x. The posterior odds of $+a$ against $-a$ at x are the ratio of the two Gaussian densities. In the exponent the terms x^2 and $(1-t)^2a^2$ cancel and only the cross term survives, so the odds are $\exp\left(2(1-t)\,ax/t^2\right)$ and the posterior mean is $\mathbb{E}[x_0 \mid x_t = x] = a\tanh\left((1-t)\,ax/t^2\right)$. The marginal velocity is therefore

$$u_t(x) = \frac{x - a\tanh\left(\frac{(1-t)\,ax}{t^2}\right)}{t}. \tag{38.98}$$

Run backward in time, the cloud at x heads for the average of the two targets, weighted by how plausible each is from x, at the speed that arrives on time. At $(t, x) = (0.5, 0.5)$ the velocity is -2.86; at $(0.7, -1)$ it is 0.97. Three properties can be read off Equation 38.98, and Figure 38.27 draws the first two.

- **At $t = 1$ the field is $u_1(x) = x$.** The noise sample carries no information about its target, so the posterior mean is the data mean 0. One Euler step from $t = 1$ to $t = 0$, $x \mapsto x - u_1(x)$, sends *every* sample to 0. This holds for any data under independent pairing: $\mathbf{u}_1(\mathbf{x}) = \mathbb{E}[\epsilon - \mathbf{x}_0 \mid \epsilon = \mathbf{x}] = \mathbf{x} - \mathbb{E}[\mathbf{x}_0]$, so one step of the exact field lands on the data mean.
- **The origin first attracts, then repels.** As t falls, a sampler moves against u_t, so a positive slope at the origin pulls particles toward it and a negative slope pushes them away. The slope is $\partial_x u_t(0) = \left(1 - (1-t)\,a^2/t^2\right)/t$: $+0.56$ at $t = 0.9$, zero at $t^* = 2\sqrt{2} - 2 \approx 0.83$ (the root of $(1-t)\,a^2 = t^2$), -0.31 at $t = 0.8$ and -14 at $t = 0.5$. Particles near 0 drift toward 0 until t falls to t^* and leave for a mode only later, so the flow chooses a mode late. Far from the origin the tanh saturates and $u_t \approx (x \mp a)/t$, the conditional velocity itself.
- **The floor is collected near the noise.** The target $\epsilon - x_0$ has mean square $a^2 + 1 = 5$. Of this, $C = \mathbb{E}_t \mathbb{E}_x \mathrm{Var}(x_0 \mid x_t)/t^2 \approx 2.0$ is floor, with $\mathrm{Var}(x_0 \mid x_t = x) = a^2\,\mathrm{sech}^2\left((1-t)\,ax/t^2\right)$, and the remaining 3.0 is the mean square of the marginal velocity. Nearly all of C comes from $t > 0.5$, where the two blobs overlap and a position does not tell which point its particle is bound for. Below $t = 0.4$ it does, and only about a thousandth of the floor is collected there, although the conditional velocity $(x - x_0)/t$ grows without bound (Exercise 2).

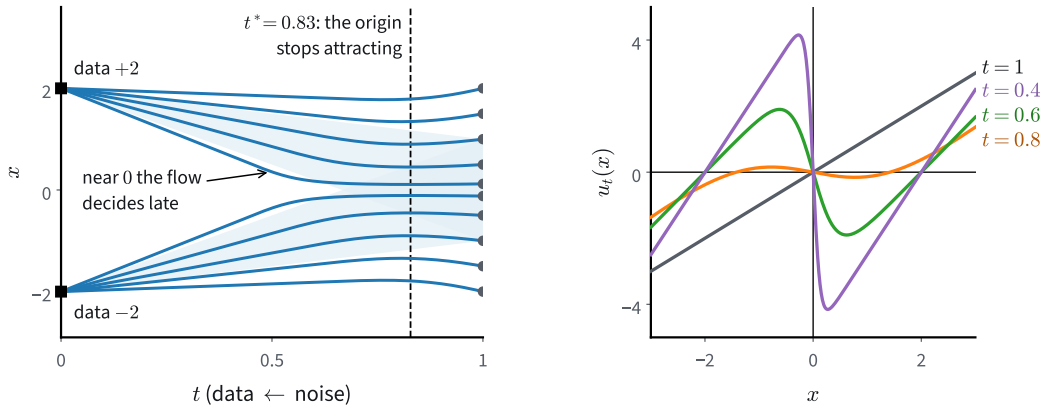


Figure 38.27: Two data points at ± 2 and standard Gaussian noise with independent pairing; sampling runs from the noise at $t = 1$ (right) to the data at $t = 0$ (left). (a) Trajectories of the exact marginal flow over the one-standard-deviation bands of the two conditional blobs: until t falls to $t^* = 0.83$ every trajectory moves toward 0, and those that start near 0 leave for a mode only below $t \approx 0.6$. (b) The velocity $u_t(x)$ is x at $t = 1$, its slope at the origin turns negative and steep as t falls, and it tends to $(x \mp 2)/t$ away from the origin.

The cell takes the slope at the origin by a finite difference at the three times of the second property. It then puts the first property to the test, and asks what more steps would do. It integrates the exact field from $t = 1$ to $t = 0$ with 1, 2, 4 and 8 equal Euler steps, starting from the whole noise law, and prints how much of the mass lands within 0.1 of a data point, together with the mean of $|x|$ at the end. The noise law is a fine grid of starting points weighted by the Gaussian density, so nothing in the output is random.

```
a = 2.0                                     # data at -a and +a, noise N(0, 1)
def u_two_point(x, t):                      # closed form of the marginal velocity
    return (x - a * np.tanh((1 - t) * a * x / t**2)) / t

ts = np.array([0.9, 0.8, 0.5])
print('slope at the origin at t = 0.9, 0.8, 0.5:',
      ((u_two_point(1e-6, ts) - u_two_point(-1e-6, ts)) / 2e-6).round(2),
      '   t* =', round(2 * np.sqrt(2) - 2, 4))
x1 = np.linspace(-6, 6, 4001)               # the noise law N(0, 1), as a weighted grid
w = np.exp(-x1**2 / 2) / np.exp(-x1**2 / 2).sum()
for K in (1, 2, 4, 8):                      # K equal Euler steps from t = 1 to t = 0
    x, ts = x1, np.linspace(1, 0, K + 1)
    for t, t_next in zip(ts[:-1], ts[1:]):
        x = x - (t - t_next) * u_two_point(x, t)
    print(f'{K} Euler step(s): mass within 0.1 of +-a {w @ (abs(abs(x) - a) < 0.1):.3f}, '
          f'mean |x| {w @ abs(x):.2f}')
```

```
slope at the origin at t = 0.9, 0.8, 0.5: [  0.56  -0.31 -14.  ]    t* = 0.8284
1 Euler step(s): mass within 0.1 of +-a 0.000, mean |x| 0.00
2 Euler step(s): mass within 0.1 of +-a 0.359, mean |x| 1.48
4 Euler step(s): mass within 0.1 of +-a 0.987, mean |x| 1.99
8 Euler step(s): mass within 0.1 of +-a 0.999, mean |x| 2.00
```

The slopes are those of the formula. After one Euler step no mass at all lies near a data point, and the mean of $|x|$ is 0: every sample sits on the data mean. So the wish of the opening fails at its first test. Every conditional path is straight, yet the marginal flow is curved, and the one long step we hoped to take collapses every sample onto the mean.

More steps repair the damage quickly. Two bring a little over a third of the mass to a data point, four all but about one percent of it, and eight practically all. So few suffice because the last step of any grid lands on the posterior mean $a \tanh(\cdot)$, which is $\pm a$ wherever the earlier steps have moved a particle far enough from the origin (Exercise 2). But steps are what we set out to save. How to pair noise with data so that the flow comes out straight, and one step is enough, is the subject of Section 38.7.2.

38.7.2 Rectified Flow

38.7.2.1 Straight Paths and Their Crossings

The two-point example used straight conditional paths. In d dimensions the same construction pairs a data sample $\mathbf{x}_0$ with a noise sample ϵ, drawn independently, and joins them by a straight line traversed at constant speed:

$$\mathbf{x}_t = (1 - t)\,\mathbf{x}_0 + t\,\epsilon, \qquad \mathbf{u}_t(\mathbf{x}_t \mid \mathbf{z}) = \dot{\mathbf{x}}_t = \epsilon - \mathbf{x}_0, \qquad \mathbf{z} = (\mathbf{x}_0, \epsilon). \tag{38.99}$$

The conditional velocity does not depend on t, and the CFM loss Equation 38.95 becomes the **rectified flow** objective (Liu et al. 2023; Lipman et al. 2023)

$$\mathcal{L}_{\mathrm{RF}}(\theta) = \mathbb{E}_{t,\, \mathbf{x}_0 \sim p_0,\, \epsilon \sim p_1} \left[\left\| \mathbf{u}_\theta((1-t)\mathbf{x}_0 + t\epsilon,\, t) - (\epsilon - \mathbf{x}_0) \right\|^2 \right]. \tag{38.100}$$

Training draws data and noise, interpolates between them, and regresses on the difference. That is the whole algorithm. Its name looks ahead: *rectified* means straightened, and straight is what the flow becomes once the pairing has been repaired. The flow of independent pairs, as we just saw, is not.

One technical point has to be settled. With Gaussian noise we could condition on $\mathbf{x}_0$ alone, as in the two-point example, and the marginal-velocity proposition would apply. Conditioning on the pair is what lets us change the pairing later, but given the pair, a point sliding along a line has no density, so the proposition does not apply as stated. Its conclusion still holds, by the same computation applied to the expectation $\mathbb{E}\,\varphi(\mathbf{x}_t)$ of a smooth test function φ (Exercise 7).

Why does the flow bend? Two segments that cross at $(\mathbf{x}, t)$ feed the posterior mean Equation 38.94 two different directions (Figure 38.28). A function has one value at each point, so the field averages them. An ODE whose field is Lipschitz in $\mathbf{x}$ has exactly one trajectory through each point (Section 38.1.1.3), so two trajectories of the learned flow can never meet. Where the segments cross, the trajectories bend away from each other instead.

The goal is a field whose trajectories are straight lines traversed at constant speed. Such a field is constant along each trajectory, so one Euler step of any length is exact. The independent pairing is far from this limit. The loss is minimized by the right field and the pieces are as straight as pieces can be, so the one thing left to change is which noise point is joined to which data point.

38.7.2.2 Example: Two Moons

The two points had a formula, and the formula said that one step fails. How many steps does a trained network need, on data that have no formula? We take the two moons, two interleaved crescents in the plane, train a small network on them, and count. To count we need a score for samples, and we use the **squared energy distance** (Székely and Rizzo 2013) between laws P and Q with finite first moments,

$$\mathcal{E}^2(P, Q) = 2\,\mathbb{E}\|X - Y\| - \mathbb{E}\|X - X'\| - \mathbb{E}\|Y - Y'\|,$$

where $X, X' \sim P$ and $Y, Y' \sim Q$ are independent. It compares the mean distance between a point of P and a point of Q with the mean distances within each law. Székely and Rizzo call this expression itself the energy distance; we call it squared because its square root is the metric. It is a squared MMD (Section 37.4.1), and it is zero only if $P = Q$. We estimate it from two samples by the unbiased U-statistic, and every score quoted below is this squared distance.

The cell builds the data, standardized to zero mean and unit variance per coordinate, a held-out sample of $2{,}048$ points, and one noise array `z_eval` that every later comparison reuses. It then scores two references against the held-out sample: a fresh sample of the data, and a point mass at the data mean, which is what one Euler step of the exact field produces.

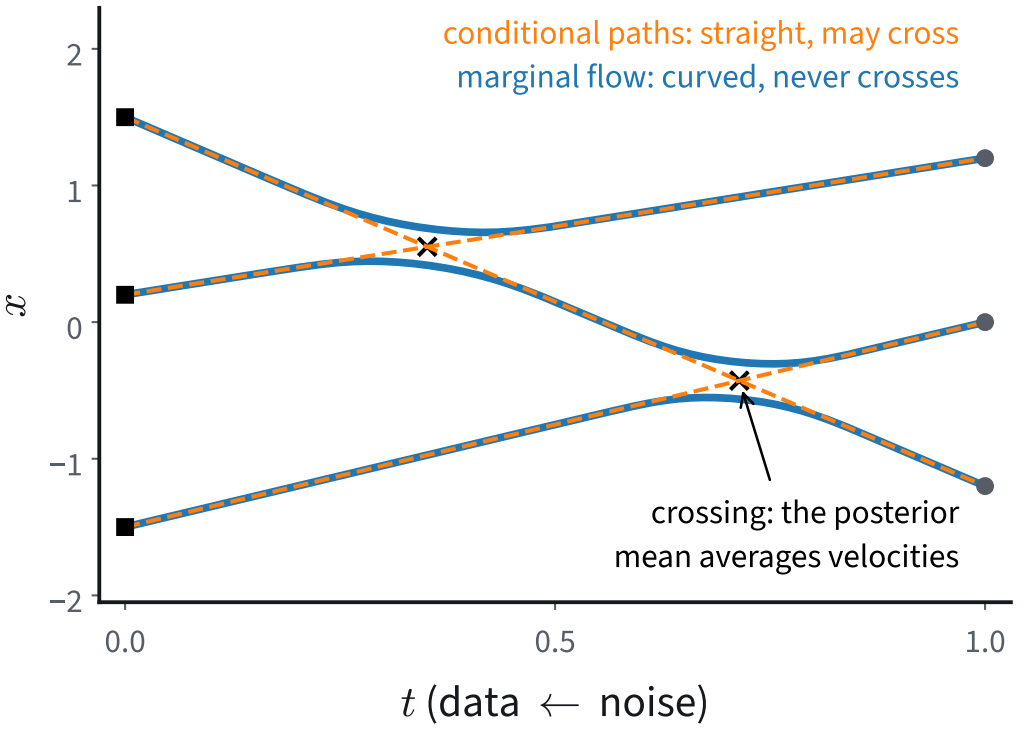


Figure 38.28: Three data–noise pairs in one dimension joined by straight segments (dashed), two pairs of which cross (×). Where segments cross, the marginal velocity averages their directions, so the trajectories of its flow (solid), run from the noise at $t = 1$ to the data at $t = 0$, bend and never meet. Each trajectory ends on a different data point than its own segment, and the trajectories arrive in the order in which they started.

```
def two_moons(n, rng, noise=0.07):
    t = rng.uniform(0.0, np.pi, n)
    m = rng.integers(0, 2, n)                       # which moon
    X = np.stack([np.cos(t) * (1 - 2 * m) + m,
                  np.sin(t) * (1 - 2 * m) + 0.5 * m], axis=1)
    return X + noise * rng.standard_normal((n, 2))

rng = np.random.default_rng()
raw = two_moons(8192, rng)
mu, sd = raw.mean(0), raw.std(0)
moons = (raw - mu) / sd                             # standardized training data
held_out = (two_moons(2048, rng) - mu) / sd
z_eval = rng.standard_normal((2048, 2))             # noise for every comparison

def energy_distance(a, b):                          # squared energy distance, U-statistic
    d = lambda u, v: np.linalg.norm(u[:, None, :] - v[None, :, :], axis=-1)
    n, m = len(a), len(b)
    return (2 * d(a, b).mean() - d(a, a).sum() / (n * (n - 1))
            - d(b, b).sum() / (m * (m - 1)))

print(f'fresh data sample vs held-out: {energy_distance(moons[:2048], held_out):+.4f}')
print(f'point mass at the data mean vs held-out: '
      f'{energy_distance(np.zeros((2048, 2)), held_out):.3f}')
```

```
fresh data sample vs held-out: -0.0000
point mass at the data mean vs held-out: 0.904
```

Two samples of the same law score within a few thousandths of zero, which is the resolution of the metric at this sample size. The point mass scores about 0.9.

The model is a small tanh network that takes $(\mathbf{x}, t)$ and returns a velocity; `field` evaluates it. The function `train_rf` takes a sampler of pairs, so that we can change the pairing later. It returns the network together with its loss averaged over the last 500 batches, because single batches scatter, and over the two coordinates, so the printed value is $\mathcal{L}_{\mathrm{RF}}/2$.

```
def make_net():                         # 3 -> 256 -> 256 -> 2, Glorot weights, zero biases
    net = nn.Sequential(nn.Linear(3, 256), nn.Tanh(), nn.Linear(256, 256), nn.Tanh(),
                        nn.Linear(256, 2))
    for layer in net[::2]:
        nn.init.xavier_uniform_(layer.weight)
        nn.init.zeros_(layer.bias)
    return net

def field(net, x, t):                   # the time enters centered, as 2t - 1
    return net(torch.cat([x, 2 * t - 1], 1))

def train_rf(sample_pairs, steps=10000, batch=2048, tail=500):
    """Regress v(x_t, t) on eps - x0 (papers: -(x1 - x0) at time 1 - t).
    Return the net and the mean tail loss."""
    net = make_net()
    opt = torch.optim.Adam(net.parameters(), lr=3e-3)
    sched = torch.optim.lr_scheduler.CosineAnnealingLR(opt, steps)   # 3e-3 -> 0
```

```
    tail_sum = torch.zeros(())
    for step in range(steps):
        x0, eps = sample_pairs(batch)              # data, noise
        t = torch.rand(batch, 1)
        xt = (1 - t) * x0 + t * eps                # a point on each segment
        loss = ((field(net, xt, t) - (eps - x0))**2).mean()
        opt.zero_grad()
        loss.backward()
        opt.step()
        sched.step()
        if step >= steps - tail:
            tail_sum += loss.detach()              # stays on the device
    return net, (tail_sum / tail).item()
```

The first run pairs noise and data independently. By the CFM theorem its loss cannot reach zero, and where it stops measures how much the segments cross. The floor C is the conditional variance of $\epsilon - \mathbf{x}_0$ given $\mathbf{x}_t$, averaged over t. Can we know it before we train? For a finite data set we can, without any network. Given $\mathbf{x}_t$, the posterior of $\mathbf{x}_0$ puts on each training point $\mathbf{x}_0'$ a weight proportional to $\exp\big(-\|\mathbf{x}_t - (1-t)\,\mathbf{x}_0'\|^2/2t^2\big)$. The target is $\epsilon - \mathbf{x}_0 = (\mathbf{x}_t - \mathbf{x}_0)/t$, so its conditional variance is the posterior variance of $\mathbf{x}_0$ divided by t^2. The function `pairing_floor` averages this quantity over $4{,}000$ draws of $(t, \mathbf{x}_0, \epsilon)$ and halves it, because `train_rf` reports its loss per coordinate.

```
def pairing_floor(data, rng, m=4000, chunk=500):
    """Floor C of independent pairing, per coordinate: E Var(eps - x0 | x_t) / 2."""
    t = rng.uniform(0, 1, (m, 1))
    xt = (1 - t) * data[rng.integers(0, len(data), m)] + t * rng.standard_normal((m, 2))
    var = []
    for i in range(0, m, chunk):              # posterior of x0 over all training points
        ti, xi = t[i:i + chunk], xt[i:i + chunk]
        d2 = ((xi**2).sum(1, keepdims=True) - 2 * (1 - ti) * xi @ data.T
              + (1 - ti)**2 * (data**2).sum(1))          # |x_t - (1 - t) x0'|^2
        logw = -d2 / (2 * ti**2)
        r = np.exp(logw - logw.max(1, keepdims=True))
        r /= r.sum(1, keepdims=True)
        v = r @ (data**2).sum(1) - ((r @ data)**2).sum(1)  # Var(x0 | x_t), summed
        var.append(v / ti[:, 0]**2)           # eps - x0 = (x_t - x0) / t
    return np.concatenate(var).mean() / 2

print(f'floor of the independent pairing, per coordinate: {pairing_floor(moons, rng):.2f}')
```

```
floor of the independent pairing, per coordinate: 1.39
```

The floor is about 1.37 per coordinate, and the estimate moves in the second decimal from one draw to the next. A perfect network would stop there. Ours has two hidden layers of 256 units and $10{,}000$ steps of $2{,}048$ pairs each to get close.

```
data = torch.tensor(moons, dtype=torch.float32)
def independent_pairs(n):                      # random data rows, fresh noise
    return data[torch.randint(0, len(data), (n,))], torch.randn(n, 2)
```

```
net, plateau = train_rf(independent_pairs)
print(f'CFM loss, mean of the last 500 batches: {plateau:.2f}')
```

```
CFM loss, mean of the last 500 batches: 1.40
```

The loss settles at about 1.4 per coordinate, within a few hundredths of the floor. A loss of 1.4 therefore does not mean a poor fit. Almost all of it is the disagreement of crossing segments, which no network can remove. By the CFM theorem only the excess over the floor is error of the field.

Those few hundredths are not as small as they look, though. The target has mean square 2 per coordinate, one each from the noise and the standardized data, and 1.37 of it is floor. The field itself therefore has a mean square of only about 0.6, and its error is a few percent of that. We shall meet this error again in the samples.

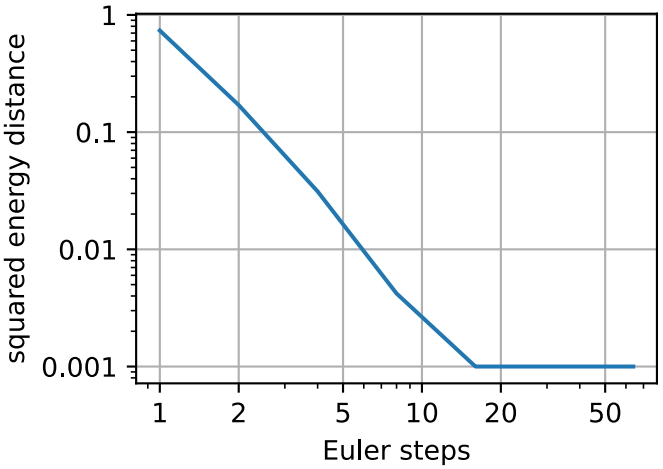
Sampling integrates $\dot{\mathbf{x}} = \mathbf{u}_\theta(\mathbf{x}, t)$ from $t = 1$ down to 0 with Euler's method (Section 38.2): K steps of length $1/K$, each of which subtracts $\mathbf{u}_\theta(\mathbf{x}, t)/K$ at $t = 1, 1-1/K, \dots, 1/K$. The function `euler_path` returns the whole trajectory, of shape `(steps + 1, n, 2)`; its last row is the sample.

```
@torch.inference_mode()
def euler_path(net, z, steps):
    q = torch.tensor(z, dtype=torch.float32)
    path = [q]
    for k in range(steps):                    # t = 1, 1 - 1/steps, ..., 1/steps
        t = torch.full((len(z), 1), 1 - k / steps)
        q = q - field(net, q, t) / steps
        path.append(q)
    return torch.stack(path).numpy()
```

One step of the exact field would give the point mass that scored 0.9. For the trained field, the cell integrates the noise array `z_eval` with $K = 1, 2, 4, \dots, 64$ steps and plots the squared energy distance to the held-out data on log–log axes. The estimate is unbiased, so at the resolution of the metric it can turn negative; the plot clips it at 10^{-3}.

```
steps_list = [1, 2, 4, 8, 16, 32, 64]
eds = [energy_distance(euler_path(net, z_eval, K)[-1], held_out) for K in steps_list]
print('  '.join(f'{K}: {e:.3f}' for K, e in zip(steps_list, eds)))
d2l.plot(steps_list, np.maximum(eds, 1e-3), 'Euler steps',
         'squared energy distance', xscale='log', yscale='log')
```

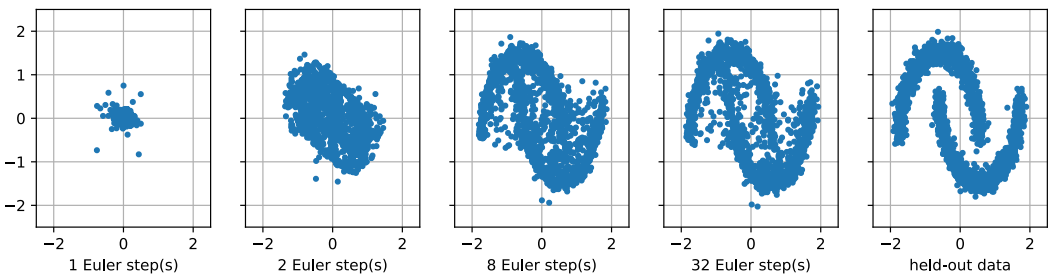
```
1: 0.736  2: 0.170  4: 0.031  8: 0.004  16: -0.000  32: -0.001  64: -0.001
```



One step scores 0.7 to 0.75, the same order as the point mass at the mean (0.9): the network has learned, imperfectly, the collapse that the exact field performs at $t = 1$. Two steps score 0.16 to 0.19, four 0.03 to 0.04, eight about 0.01 or less, and from sixteen steps on the score stays within a few thousandths of zero. The decline is Euler's discretization error. What remains is the error of the trained field, and it is as small as the metric can resolve with $2{,}048$ points. More steps help, as they should for a field fitted without a solver; the field trained through ten Euler steps in Section 38.3.1 got worse when its steps were refined.

A score is one number, and it is worth looking at what it summarizes. The next cell draws the samples of 1, 2, 8 and 32 steps, and beside them the held-out data.

```
panels = [(f'{K} Euler step(s)', euler_path(net, z_eval, K)[-1]) for K in (1, 2, 8, 32)]
fig, axes = d2l.plt.subplots(1, 5, figsize=(12, 2.6), sharex=True, sharey=True)
for ax, (title, s) in zip(axes, panels + [('held-out data', held_out)]):
    d2l.plot(s[:, 0], s[:, 1], title, fmts=('.',),
             xlim=(-2.5, 2.5), ylim=(-2.5, 2.5), axes=ax)
```



One step gives a blob around the mean and two a slanted band. Eight steps spread the cloud into two crescents, with many points still strewn between them. Thirty-two steps clear most of these and draw two separate moons, like the data in the last panel. Set them beside the data, though, and they are a little thicker, with a few points left in the gaps. More steps do not remove them, so these flaws belong to the field and not to the solver: they are what the few hundredths between the loss and its floor look like.

The score hardly registers these flaws. It compares mean distances between points, so it

reports where the mass lies long before it reports how sharp the edges are, and a score at the resolution of the metric does not certify samples as sharp as the data. What more steps do remove is the solver's error, and by the score that takes sixteen to thirty-two network calls per sample. This is the cost of the curvature that the independent pairing builds into the flow.

38.7.2.3 Reflow

The independent pairing is what bends the flow, and the trained model supplies a better one. Run its ODE from fresh noise and pair each noise point ϵ with the endpoint $\hat{\mathbf{x}}_0(\epsilon)$ at which it arrives. Trajectories of one flow never cross, so these pairs cross far less than independent ones. Retraining the same architecture on the new pairs is called **reflow** (Liu et al. 2023).

Does re-pairing do any harm? It could spoil the endpoint laws, or pair points that lie farther apart than before. It does neither. How far a pairing moves its points is measured by its *transport cost* $\mathbb{E}\, c(\mathbf{x}_1 - \mathbf{x}_0)$, the mean over the pairs of a convex function c of the displacement, such as its squared length. In the statement capitals denote random paths: $\mathbf{X}_t$ is the straight segment of a pair, and $\mathbf{Z}_t$ the trajectory of the flow that starts from the same noise point.

Proposition (reflow keeps the marginals and lowers the transport cost). *Let the pair $(\mathbf{X}_0, \mathbf{X}_1)$ of a data point and a noise point have finite second moments, let $\mathbf{X}_t = (1-t)\mathbf{X}_0 + t\mathbf{X}_1$, let $\mathbf{u}_t$ be the marginal velocity of this straight path, and let $\mathbf{Z}_t$ solve $\dot{\mathbf{Z}}_t = \mathbf{u}_t(\mathbf{Z}_t)$ backward in time from the noise end, $\mathbf{Z}_1 = \mathbf{X}_1$. Assume that $\mathbf{X}_t$ has a positive density for every $t > 0$ and that $\mathbf{u}_t$ is continuously differentiable and Lipschitz in $\mathbf{x}$ with a constant independent of t, so that this flow exists. Then (i) $\mathbf{Z}_t$ has the law of $\mathbf{X}_t$ for every t, in particular $\mathbf{Z}_0 \sim p_0$; (ii) for every convex function c,*

$$\mathbb{E}\, c(\mathbf{Z}_1 - \mathbf{Z}_0) \;\le\; \mathbb{E}\, c(\mathbf{X}_1 - \mathbf{X}_0). \tag{38.101}$$

Proof. (i) The law of $\mathbf{X}_t$ satisfies the continuity equation with $\mathbf{u}_t$, by the marginal-velocity proposition with $\mathbf{z} = (\mathbf{X}_0, \mathbf{X}_1)$ in the weak form that covers point masses (Exercise 7). The law of $\mathbf{Z}_t$ satisfies it too, because $\mathbf{Z}_t$ moves with the field $\mathbf{u}_t$. Both equal p_1 at $t = 1$, and two solutions of the continuity equation with such a field that agree at one time agree at every time. This uniqueness is the one step we do not prove in full: Section 38.5.4.3 gave the argument for smooth positive densities, where along each trajectory both log-densities change at the rate $-\nabla \cdot \mathbf{u}_t$, and we use it here without redoing it in the weak form. (ii) Write the displacement as a time integral and apply Jensen's inequality twice:

$$\begin{aligned}
\mathbb{E}\, c(\mathbf{Z}_1 - \mathbf{Z}_0) &= \mathbb{E}\, c\Big(\int_0^1 \mathbf{u}_t(\mathbf{Z}_t)\, dt \Big) \le \int_0^1 \mathbb{E}\, c(\mathbf{u}_t(\mathbf{Z}_t))\, dt \\
&= \int_0^1 \mathbb{E}\, c(\mathbb{E}[\mathbf{X}_1 - \mathbf{X}_0 \mid \mathbf{X}_t])\, dt \le \mathbb{E}\, c(\mathbf{X}_1 - \mathbf{X}_0).
\end{aligned}$$

The first inequality is Jensen's inequality for the time average. The equality uses (i) to replace $\mathbf{Z}_t$ by $\mathbf{X}_t$ and then the definition of the marginal velocity. The last inequality is the conditional Jensen inequality. ■

With $c = \|\cdot\|^2$ the last step loses exactly the conditional variance, so the cost falls at least by the floor of the old pairing: $\mathbb{E}\|\mathbf{Z}_1 - \mathbf{Z}_0\|^2 \le \mathbb{E}\|\mathbf{X}_1 - \mathbf{X}_0\|^2 - C$. The floor C, a nuisance in

training, is the saving that one round of reflow guarantees. Where no segments cross, the floor is zero and re-pairing has nothing to gain.

The hypotheses are stronger than our examples allow. For the two points, and for any finite training set paired independently with Gaussian noise, the marginal field is Lipschitz on $[\delta, 1]$ for each $\delta > 0$ but not up to $t = 0$: the slope -14 of the two-point field at $t = 0.5$ grows without bound as $t \to 0$. The proof then covers the flow down to time δ only: it gives (i) for $t \geq \delta$, and (ii) in the form $\mathbb{E}\, c\big((\mathbf{Z}_1 - \mathbf{Z}_\delta)/(1-\delta)\big) \leq \mathbb{E}\, c(\mathbf{X}_1 - \mathbf{X}_0)$. The limit $\delta \to 0$ we do not prove, and in this sense the examples that follow go beyond the proposition.

For the two points the conclusion can be verified directly. A flow in one dimension preserves order, so one round pairs each noise point ϵ with $a\,\mathrm{sign}(\epsilon)$ (Exercise 4). This pairing costs $a^2 - 2a\sqrt{2/\pi} + 1 = 1.81$ against $a^2 + 1 = 5$ for the independent one, and its segments never cross, so the second round has $C = 0$ and its field is integrated exactly by one Euler step.

For the two moons, the cell draws $8{,}192$ fresh noise points, integrates the trained model for 32 steps to obtain their endpoints, and trains a new network with `train_rf` on these fixed pairs.

```
eps_pairs = rng.standard_normal((8192, 2))            # noise endpoints, kept
x0_pairs = euler_path(net, eps_pairs, 32)[-1]         # the model's own endpoints
X0, Eps = (torch.tensor(p, dtype=torch.float32) for p in (x0_pairs, eps_pairs))
def model_pairs(n):                                   # the same pairs, never re-paired
    i = torch.randint(0, len(X0), (n,))
    return X0[i], Eps[i]

net2, plateau2 = train_rf(model_pairs)
print(f'reflow loss, mean of the last 500 batches: {plateau2:.4f}')
```

```
reflow loss, mean of the last 500 batches: 0.0007
```

The reflow loss settles below a thousandth, where the first loss stopped at 1.4, so the new segments almost never cross. A field without crossings is nearly constant along its trajectories, and then a few Euler steps should do. The next cell samples from both models with 1, 2 and 32 steps from the same noise array.

It also prints the mean squared length of the old and the new pairs, the cost that the proposition bounds. The old pairs cost 4: noise and standardized data each have unit variance in each of two coordinates, and for independent draws the variances add. The floor of the independent pairing, summed over the two coordinates, is twice the value printed above, about 2.75. For the exact marginal flow the proposition therefore promises new pairs that cost at most about 1.25.

```
for K in (1, 2, 32):
    e1 = energy_distance(euler_path(net, z_eval, K)[-1], held_out)
    e2 = energy_distance(euler_path(net2, z_eval, K)[-1], held_out)
    print(f'{K:2d} Euler step(s): squared energy distance  original {e1:.3f}'
          f'  reflowed {e2:.3f}')
x0_indep = moons[rng.integers(0, len(moons), len(eps_pairs))]
print(f'mean squared pair length: independent {np.mean(np.sum((eps_pairs - x0_indep)**2,
↪  1)):.2f}')
```

```
        f', reflowed {np.mean(np.sum((eps_pairs - x0_pairs)**2, 1)):.2f}')
```

```
 1 Euler step(s): squared energy distance  original 0.736  reflowed -0.001
 2 Euler step(s): squared energy distance  original 0.170  reflowed -0.001
32 Euler step(s): squared energy distance  original -0.001  reflowed -0.001
mean squared pair length: independent 4.04, reflowed 0.29
```

One Euler step of the reflowed model scores the same as thirty-two steps of the original: a thousandth or two at most, the resolution of the metric, where the original's single step scored about 0.7. It could not have done better. The reflowed network was trained to reproduce the original's 32-step endpoints, so it inherits the first model's error along with its samples.

The mean squared length of the pairs drops from 4 to about 0.3, far below the 1.25 that the bound guarantees. The slack is in the first step of the proof: Jensen's inequality for the time average is far from tight on the curved trajectories of the first flow.

Straightness can be measured as well, by the mean distance of each 32-step trajectory from the chord joining its two ends.

```
def chord_gap(P):                                    # P: (steps + 1, n, 2) trajectory
    chord = P[0] + (P[-1] - P[0]) * np.linspace(0, 1, len(P))[:, None, None]
    return np.linalg.norm(P - chord, axis=-1).mean()

print(f'mean distance from the chord: original {chord_gap(euler_path(net, z_eval, 32)):.3f}'
      f', reflowed {chord_gap(euler_path(net2, z_eval, 32)):.3f}')
```

```
mean distance from the chord: original 0.249, reflowed 0.005
```

The original trajectories wander about a quarter of a unit from their chords. They set out toward the mean, where a single Euler step left them, and turn out to the moons only later. The reflowed ones stay about a hundredth of theirs, which is why one step suffices. Reflow straightens a flow, but it does not make it optimal. In more than one dimension a flow can be straight and crossing-free and still move mass farther than it must.

38.7.3 Optimal Transport and Straightness

38.7.3.1 Couplings and the Wasserstein Distance

Reflow lowered the transport cost. How low can it go, and is the flow of the cheapest pairing straight? We restate the vocabulary of Section 37.4.2. A **coupling** π of p_0 and p_1 is a joint law of pairs with these marginals, a plan that says how much mass travels from each source point to each target point. The **2-Wasserstein distance** is the cost of the cheapest plan under squared distance,

$$W_2^2(p_0, p_1) = \min_{\pi \in \Pi(p_0, p_1)} \mathbb{E}_{(\mathbf{x}_0, \mathbf{x}_1) \sim \pi} \left[\|\mathbf{x}_1 - \mathbf{x}_0\|^2 \right], \tag{38.102}$$

where $\Pi(p_0, p_1)$ is the set of couplings. So W_2^2 is the least mean squared displacement that turns one law into the other.

What does the cheapest plan look like? When one of the two laws has a density and both have finite second moments, it is unique and it does not split mass: it is a map from the law with the density. For us that law is the noise p_1, and the map from noise to data is the gradient of a convex function, the *Brenier map*. This is Brenier's theorem (Brenier 1991), which we cite without proof. The data need no density: the two points have none, and neither has a finite training set such as our $8{,}192$ moons. On a line the theorem is easy to picture. The derivative of a convex function of one variable never decreases, so the optimal map keeps the points in order.

The smallest case is a Gaussian in one dimension, with data $\mathcal{N}(m, s^2)$, $s > 0$, and noise $\mathcal{N}(0, 1)$. The sorted pairing sends each quantile to the same quantile, $x_0 = m + s\epsilon$, and costs $\mathbb{E}(m + s\epsilon - \epsilon)^2 = m^2 + (s-1)^2$. The independent pairing costs $\mathbb{E}(\epsilon - x_0)^2 = m^2 + 1 + s^2$. In one dimension the sorted pairing is optimal for every convex cost: if two pairs cross, swapping their targets does not increase the cost (Exercise 2d of Section 37.4). Hence

$$W_2^2(\mathcal{N}(0,1), \mathcal{N}(m, s^2)) = m^2 + (s-1)^2. \tag{38.103}$$

The distance separates the cost of shifting the mean from the cost of stretching the width. Between $\mathcal{N}(0, 4)$ and $\mathcal{N}(0, 1)$ it is 1, against 5 for independent pairs (panels (a) and (b) of Figure 38.29): the independent pairing spends four fifths of its cost moving mass past other mass.

This explains where the trajectories of Figure 38.28 end. In one dimension the trajectories of a flow cannot cross, so a flow preserves order, and the order-preserving pairing is the optimal one. On a line, one round of reflow therefore reaches optimal transport (Exercise 4).

In two dimensions, freedom from crossings is not enough. Rotate a standard Gaussian cloud about the origin by a quarter turn in unit time, with the field that assigns to the point (ξ_1, ξ_2) the velocity $\frac{\pi}{2}(-\xi_2, \xi_1)$. No two trajectories cross, and the cloud ends where it started, so $W_2^2(p_0, p_1) = 0$. Yet every particle moves, at speed $\frac{\pi}{2}\|\mathbf{x}\|$. It is a carousel: everyone travels, and the crowd as a whole goes nowhere (panel (c) of Figure 38.29).

Straightening the arcs does not cure this. Pair each $\mathbf{x}_0$ with its quarter turn $\mathbf{R}\mathbf{x}_0$ and join the two by a segment. The segments $((1-t)\mathbf{I} + t\mathbf{R})\mathbf{x}_0$ never cross, because that matrix is invertible for every t, and each is traversed at constant speed. So $C = 0$ and reflow leaves the pairing unchanged, yet it costs $\mathbb{E}\|\mathbf{R}\mathbf{x}_0 - \mathbf{x}_0\|^2 = 4$ against $W_2^2 = 0$. On a line, crossings are the only way to waste motion. In the plane a pairing can be straight, crossing-free and still wrong. Brenier's theorem says what this one lacks. An optimal map is the gradient of a convex function, and a quarter turn is not the gradient of anything: the derivative of a gradient is a symmetric matrix, and $\mathbf{R}$ is not symmetric.

A pairing records only where each particle ends. How should the motion in between be charged? Consider one particle that travels from $\mathbf{x}_1$ to $\mathbf{x}_0$ in unit time along some path $\mathbf{x}_t$. By Jensen's inequality for the time average,

$$\int_0^1 \|\dot{\mathbf{x}}_t\|^2 \, dt \;\geq\; \Big\| \int_0^1 \dot{\mathbf{x}}_t \, dt \Big\|^2 = \|\mathbf{x}_1 - \mathbf{x}_0\|^2,$$

with equality only if the velocity is constant. The integral of the squared speed is smallest on

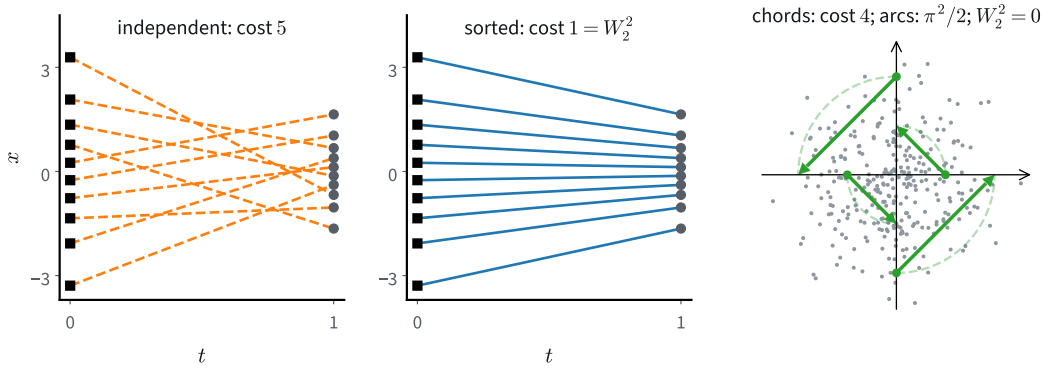


Figure 38.29: Three ways to move a Gaussian. (a) Independent pairing of data $\mathcal{N}(0, 4)$ at $t = 0$ with noise $\mathcal{N}(0, 1)$ at $t = 1$, drawn for ten equally spaced quantile levels $(0.05, 0.15, \dots, 0.95)$: segments cross, and the mean squared displacement is 5. (b) The sorted pairing $x_0 = 2\epsilon$ never crosses and costs 1, the squared Wasserstein distance. (c) A standard Gaussian cloud in the plane, turned by a quarter. The arcs of the rotation (dashed) have kinetic energy $\pi^2/2$. The chords that pair each point with its quarter turn (arrows) are straight, never cross and cost 4. The cloud ends where it started, so $W_2^2 = 0$: in two dimensions a straight, crossing-free pairing need not be optimal.

the straight line at constant speed, and it charges for every detour and every change of pace. Its mean over the particles is the **kinetic energy** of a flow, $\int_0^1 \mathbb{E}\|\mathbf{u}_t(\mathbf{x}_t)\|^2 \, dt$. The carousel has kinetic energy $(\pi/2)^2 \, \mathbb{E}\|\mathbf{x}\|^2 = \pi^2/2 \approx 4.9$, all of it wasted.

Kinetic energy also ranks flows with the same endpoints. Take three flows between data $\mathcal{N}(0, 4)$ and noise $\mathcal{N}(0, 1)$: the straight flow of the sorted pairing, the marginal flow of straight segments under the independent pairing, and the cosine path of Section 38.7.1.3. Which is cheapest, and does a flow assembled from straight segments beat one that is not? All three have Gaussian marginals $\mathcal{N}(0, s_t^2)$, so their velocity is $(\dot{s}_t/s_t)\, x$ and the kinetic energy is $\int_0^1 \dot{s}_t^2 \, dt$ (Exercise 3). The cell first checks the two pairing costs, 5 and 1, by sorting samples.

```
n = 200_000
x0, eps = 2 * rng.standard_normal(n), rng.standard_normal(n)
print(f'data N(0,4), noise N(0,1): independent pairing {np.mean((eps - x0)**2):.2f}, '
      f'sorted pairing {np.mean((np.sort(eps) - np.sort(x0))**2):.3f}')

def kinetic_energy(s, t):                    # marginals N(0, s_t^2): u_t(x) = (s'/s) x
    return np.sum(np.diff(s)**2 / np.diff(t))  # the integral of (ds/dt)^2

t = np.linspace(0, 1, 100_001)
for name, s in [('sorted, straight', 2 - t),
                ('independent, straight', np.sqrt(4 * (1 - t)**2 + t**2)),
                ('cosine path', np.sqrt(1 + 3 * np.cos(np.pi * t / 2)**2))]:
    print(f'{name:22s} kinetic energy {kinetic_energy(s, t):.3f}')
```

```
data N(0,4), noise N(0,1): independent pairing 4.99, sorted pairing 0.985
```

```
sorted, straight       kinetic energy 1.000
independent, straight  kinetic energy 1.858
cosine path            kinetic energy 1.234
```

The sorted straight flow is cheapest, and its kinetic energy is $W_2^2 = 1$. The surprise is the order of the other two. The straight segments of the independent pairing produce a marginal flow that costs 1.86, more than the cosine path's 1.23. Its variance $s_t^2 = 4(1-t)^2 + t^2$ first shrinks, as t falls from 1 to 0.8, and then grows, so its particles move inward and back out. Straight conditional segments under the wrong pairing do not make a straight flow.

38.7.3.2 The Benamou–Brenier Theorem

The cheapest of the three flows cost exactly W_2^2, the cost of the cheapest pairing. This is no accident of Gaussians. The cheapest flow between two laws always costs what the cheapest pairing costs, and it is the flow that carries each pair of that pairing along its segment at constant speed. This is the deepest fact of the section. It connects the static problem Equation 38.102, which pairs endpoints, to the objects that flow matching trains, which are flows.

Theorem (Benamou–Brenier). *Let p_0 and p_1 have finite second moments, and let one of them, here the noise p_1, have a density. Call a pair $(p_t, \mathbf{u}_t)$ admissible if $\mathbf{u}_t$ is continuously differentiable and Lipschitz in $\mathbf{x}$ with a constant independent of t, and the pair satisfies the continuity equation with the endpoints p_0 and p_1. (i) Every admissible pair has kinetic energy at least $W_2^2(p_0, p_1)$. (ii) Join the pairs of an optimal coupling by straight segments. If the marginal velocity of this path is admissible, its kinetic energy is $W_2^2(p_0, p_1)$, so that, with the minimum over admissible pairs,* (Benamou and Brenier 2000)

$$W_2^2(p_0, p_1) = \min_{(p_t, \mathbf{u}_t)} \int_0^1 \int \|\mathbf{u}_t(\mathbf{x})\|^2 \, p_t(\mathbf{x}) \, d\mathbf{x} \, dt. \tag{38.104}$$

The condition in (ii) is met, for example, when the Brenier map is a smooth diffeomorphism whose derivative and inverse derivative are bounded. Without it the equality still holds, with the minimum taken over weak solutions of the continuity equation with square-integrable velocity. We do not prove this general form, and our examples need it: the optimal field for the two points has slope $1/t$ and is not admissible.

Proof. We bound the least kinetic energy by W_2^2 from below and from above. Step II is the lower bound (i), and step III gives (ii).

I. *Every flow defines a coupling.* Let $\mathbf{Z}_t$ solve $\dot{\mathbf{Z}}_t = \mathbf{u}_t(\mathbf{Z}_t)$ backward in time from $\mathbf{Z}_1 \sim p_1$. By the uniqueness fact used in the reflow proposition, $\mathbf{Z}_t \sim p_t$ for every t, so $(\mathbf{Z}_0, \mathbf{Z}_1)$ is a coupling of p_0 and p_1.

II. *Lower bound.* The cost of this coupling is at least W_2^2, and Jensen's inequality for the time average bounds it by the kinetic energy:

$$W_2^2(p_0,p_1) \le \mathbb{E}\,\|\mathbf{Z}_1-\mathbf{Z}_0\|^2 = \mathbb{E}\left\|\int_0^1 \mathbf{u}_t(\mathbf{Z}_t)\,dt\right\|^2$$
$$\le \int_0^1 \mathbb{E}\,\|\mathbf{u}_t(\mathbf{Z}_t)\|^2\,dt = \int_0^1\int \|\mathbf{u}_t\|^2\,p_t\,d\mathbf{x}\,dt.$$

III. *Upper bound.* Take an optimal coupling $\pi^\star$ and its straight interpolation $\mathbf{X}_t = (1-t)\mathbf{X}_0 + t\mathbf{X}_1$, with marginal velocity $\mathbf{u}_t(\mathbf{x}) = \mathbb{E}[\mathbf{X}_1-\mathbf{X}_0 \mid \mathbf{X}_t=\mathbf{x}]$. By the marginal-velocity proposition in its weak form this pair satisfies the continuity equation with the right endpoints, and by the conditional Jensen inequality its kinetic energy is

$$\int_0^1 \mathbb{E}\big\|\mathbb{E}[\mathbf{X}_1-\mathbf{X}_0 \mid \mathbf{X}_t]\big\|^2\,dt \le \mathbb{E}\|\mathbf{X}_1-\mathbf{X}_0\|^2 = W_2^2(p_0,p_1).$$

The two bounds meet. ■

What does the theorem say about the model? Train rectified flow on an optimal pairing and the wish of the opening is granted: the floor is zero, the exact field is constant along each trajectory, and one Euler step of length one integrates it without error. Both properties are cases of equality in the proof, whose two inequalities we met in the reflow proposition. When the flow of an optimal coupling is admissible it attains the lower bound, so the inequality in III is an equality for it and the conditional variance is zero: the segments of an optimal coupling do not cross. Jensen's inequality in II is tight only if each particle moves at constant velocity, so every minimizer moves its particles along straight lines at constant speed. The converse fails, as the quarter turn showed. A flow can be straight, free of crossings and exact in one step without being cheapest; it attains W_2^2 only if its pairing is optimal as well (Exercise 7).

An optimal coupling is the target, then, and practice has two approximations to it. Reflow repairs by the model. A *minibatch optimal-transport* pairing (Tong et al. 2024; Pooladian et al. 2023) needs no trained model: before each training step it re-pairs the noise and data points of the batch by the assignment of least total squared distance. Exercise 5 trains the two moons with this pairing and measures what it does to the loss plateau, to the one-step score and to the length of the pairs. Neither method finds the optimal coupling of the two laws. A batch plan depends on the batch. Reflow stops at any pairing without crossings, optimal or not, as the quarter turn showed, and it inherits the errors of the first model besides.

38.7.4 Sampling with a Learned Field

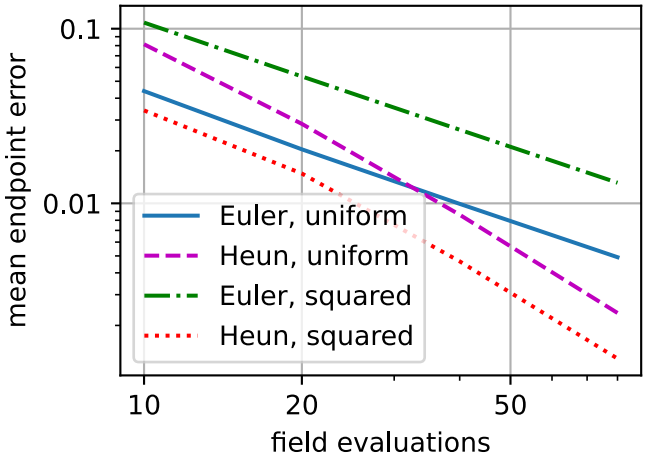
Straightening the flow is one way to save network calls. The other is to use the calls better: a solver of higher order, or steps placed where the path moves. This is open to a learned score as well as to a learned velocity, because the probability-flow ODE of Section 38.6.3 turns a score into a velocity field, and on a variance-preserving path into the very field that flow matching would learn. A sample costs one field evaluation per solver stage, and the question is how to use a budget of them. We run one experiment, on a field with no learning error, so that every error in it is the solver's.

Heun's method is second order but evaluates the field twice per step. Is it worth two evaluations? We take the probability-flow ODE with the *exact* score of the running mixture, components at ± 2 with variance 0.25, under the variance-preserving schedule with $\beta(t)$

rising linearly from 0.1 to 20. The cell integrates from $t = 1$ to $t = 0$ with Euler and with Heun's method (Section 38.2), and compares the endpoints of $8{,}000$ starting points with a Heun solution of 800 steps. Both methods get the same budgets of 10 to 80 field evaluations (the row NFE of the printed table, for "number of function evaluations"), spent on two grids of times: equal steps in t, and the *squared* grid $t_k = (1 - k/K)^2$ for $k = 0, \dots, K$, whose steps shrink toward the data end $t = 0$. Heun's error falls like the square of the step and Euler's like the step, so with enough evaluations Heun must win. Are ten enough?

```
bmin, bmax = 0.1, 20.0
beta_fn = lambda t: bmin + t * (bmax - bmin)           # VP noise schedule
abar_fn = lambda t: np.exp(-(bmin * t + 0.5 * (bmax - bmin) * t**2))
def score_vp(q, t, means=(-2.0, 2.0), var=0.25):       # exact mixture score
    a = abar_fn(t)
    v = a * var + (1 - a)
    w = np.stack([np.exp(-(q - np.sqrt(a) * m)**2 / (2 * v)) for m in means])
    return (w * np.stack([(np.sqrt(a) * m - q) / v
                          for m in means])).sum(0) / w.sum(0)
def pf_ode(q, t):                                      # probability-flow ODE
    return -0.5 * beta_fn(t) * (q + score_vp(q, t))
def solve(q, ts, heun=False):                          # steps along grid ts
    for t1, t2 in zip(ts[:-1], ts[1:]):
        h, d1 = t2 - t1, pf_ode(q, t1)
        q = q + 0.5 * h * (d1 + pf_ode(q + h * d1, t2)) if heun else q + h * d1
    return q
grids = {'uniform': lambda K: np.linspace(1, 0, K + 1),     # equal steps in t
         'squared': lambda K: np.linspace(1, 0, K + 1)**2}  # dense near t = 0
z = rng.standard_normal(8000)
ref = solve(z, grids['uniform'](800), heun=True)       # fine reference
nfes = [10, 20, 40, 80]                                # Heun: n // 2 steps
errs = {f'{m}, {g}': [np.abs(solve(z, grid(n // k), k == 2) - ref).mean()
                      for n in nfes]
        for g, grid in grids.items() for m, k in (('Euler', 1), ('Heun', 2))}
print(f'{"NFE":16s}' + ''.join(f'{n:8d}' for n in nfes))
for name, e in errs.items():
    print(f'{name:16s}' + ''.join(f'{v:8.4f}' for v in e))
d2l.plot(nfes, list(errs.values()), 'field evaluations', 'mean endpoint error',
         legend=list(errs), xscale='log', yscale='log')
```

```
NFE                10      20      40      80
Euler, uniform   0.0439  0.0204  0.0099  0.0049
Heun, uniform    0.0814  0.0285  0.0086  0.0024
Euler, squared   0.1084  0.0533  0.0264  0.0131
Heun, squared    0.0340  0.0148  0.0047  0.0013
```



On the uniform grid the orders show in the rates. Each doubling of the budget divides Euler's error by about two and Heun's by a factor that grows from about three toward four. At equal cost, though, Heun loses at 10 evaluations (about 0.08 against Euler's 0.04) and at 20 (nearly 0.03 against 0.02), and leads only from 40. On the squared grid Heun wins at every budget, and at 10 evaluations its error, about 0.03, is below Euler's on either grid. Euler moves the other way: the squared grid more than doubles its error. On this field, at small budgets, the grid decides which solver wins.

How can a first-order method beat a second-order one? The endpoint error is a sum of signed step errors, and signed errors can cancel. The next cell measures how much. It takes the error of the k-th coarse step, started from a point of the exact path, and multiplies it by the sensitivity of the endpoint to a displacement at that time, read off a neighboring trajectory of the exact flow. The product c_k is the step's contribution to the endpoint error, to first order. The cell prints two tables: the share $|\sum_k c_k| / \sum_k |c_k|$ that survives the summation, and the total $\sum_k |c_k|$ before any cancellation, averaged over the starting points. Its last line says how much of each path's length lies between $t = 1$ and $t = 0.5$.

```
def exact_path(q, ts):                  # fine Heun path, and the gain dx(0)/dx(t)
    qs = [np.stack([q, q + 1e-6])]      # the path and a close neighbor
    for t1, t2 in zip(ts[:-1], ts[1:]):
        qs.append(solve(qs[-1], [t1, t2], heun=True))
    qs = np.array(qs)
    return qs[:, 0], (qs[-1, 1] - qs[-1, 0]) / (qs[:, 1] - qs[:, 0])

def step_errors(ts, qs, gain, steps, heun):   # c_k, shape (steps, starting points)
    j = np.arange(0, len(ts), (len(ts) - 1) // steps)    # coarse nodes
    return np.array([gain[b] * (solve(qs[a], ts[[a, b]], heun) - qs[b])
                     for a, b in zip(j[:-1], j[1:])])   # step error, carried to 0

paths = {g: exact_path(z, grid(400)) for g, grid in grids.items()}   # 400 fine steps
cs = {f'{m}, {g}': [step_errors(grids[g](400), qs, gain, n // k, k == 2) for n in nfes]
      for g, (qs, gain) in paths.items() for m, k in (('Euler', 1), ('Heun', 2))}
stats = {'surviving share': (lambda c: np.abs(c.sum(0)).mean() / np.abs(c).sum(0).mean(),
 ↪  '.2f'),
         'total of |c_k|': (lambda c: np.abs(c).sum(0).mean(), '.4f')}   # sum_k |c_k|
```

```
    for title, (stat, fmt) in stats.items():
        print(f'{title:16s}' + ''.join(f'{n:8d}' for n in nfes))
        for name, c in cs.items():
            print(f'{name:16s}' + ''.join(f'{stat(c_n):8{fmt}}' for c_n in c))
    d = np.abs(np.diff(paths['uniform'][0], axis=0))    # |fine step| along each path
    share = (d[:200].sum(0) / d.sum(0)).mean()           # first 200 steps: t from 1 to 0.5
    print(f'share of the path length covered by t = 0.5: {share:.2f}')
```

```
surviving share       10      20      40      80
Euler, uniform      0.11    0.11    0.12    0.12
Heun, uniform       0.59    0.66    0.71    0.73
Euler, squared      0.31    0.32    0.34    0.35
Heun, squared       0.25    0.36    0.41    0.42
total of |c_k|        10      20      40      80
Euler, uniform    0.2867  0.1502  0.0769  0.0389
Heun, uniform     0.1317  0.0414  0.0119  0.0032
Euler, squared    0.2472  0.1327  0.0691  0.0353
Heun, squared     0.1258  0.0397  0.0113  0.0030
share of the path length covered by t = 0.5: 0.09
```

By $t = 0.5$ the paths have covered less than a tenth of their length, because for most of that time the noised law is close to $\mathcal{N}(0, 1)$, whose score cancels the drift. Equal steps in t use half the budget on that tenth, which is the reason for trying a crowded grid at all.

The shares explain Euler's lead. On equal steps only about an eighth of Euler's step errors survives in the endpoint, at every budget, against more than half of Heun's. Before cancellation Euler's step errors are the larger ones: its totals in the second table are more than twice Heun's at every budget. Had as large a share of them survived, Euler would have lost everywhere. Its lead is cancellation, a property of this field on this grid and not of the method. The squared grid takes it away: about a third of Euler's step errors survives there, and a quarter to two fifths of Heun's.

Surprisingly, this is nearly all that the squared grid changes. The totals hardly notice the grid. For Heun at 10 evaluations the total is 0.13 on equal steps and 0.125 on the squared grid, and the two stay within a few percent of each other at every budget. Euler's total is even smaller on the squared grid, by about a seventh at 10 evaluations and a tenth at 80, and yet its endpoint error more than doubles. So crowding the steps made the step errors only a little smaller in total. What it changed is how much of them cancels, to Heun's gain and Euler's loss. The squared grid is cruder than it looks: its step at time t is about $2\sqrt{t}$ times the equal step, shorter below $t = 1/4$ and longer above.

Samplers for image models crowd their steps toward the data as well, and they tune how much by measurement (Karras et al. 2022), as Section 17.5 describes. Our tables show why they have to measure: a comparison of solvers on one grid is a statement about that grid.

Two questions stay open. Nobody knows how to find the optimal pairing cheaply in high dimension; reflow and batch re-pairing only approach it. And the question the chapter began with, how few steps a model can take, is still answered by trial for each new field: the number of steps and their placement are tuned, not derived. Chapter 17 takes up both questions for images.

38.7.5 Summary

The path from noise to data can be chosen. The velocity that realizes it is the posterior mean of the velocities of per-sample paths, so conditional flow matching trains it by least squares, with a loss floor C that measures how much those paths disagree. Every model of this section and the previous one is trained this way: it fits a conditional mean of a per-sample quantity that is known in closed form, and it samples with a solver. On a Gaussian path the score, the noise and the velocity are affine in one posterior mean, so one network can be read as any of them. Straight conditional paths give a curved flow wherever segments cross; reflow re-pairs noise with the flow's own endpoints, does not increase the transport cost, and on the two moons lets one Euler step match thirty-two. The cost cannot fall below the squared Wasserstein distance, the least kinetic energy of any flow between the endpoints, which straight constant-speed motion attains only under an optimal pairing. How many steps a field needs also depends on where they are placed, because the grid decides how much of the step errors cancels. On the exact field of the running mixture, Heun's method loses to Euler below about 40 evaluations on a uniform grid and wins at every budget on a grid crowded toward the data.

38.7.6 Exercises

1. **Stochastic interpolants.** ● Albergo et al. (2023) write the paths of this section and of the previous one as $\mathbf{x}_t = a_t\mathbf{x}_0 + b_t\mathbf{x}_1 + c_t\mathbf{w}$, where $(\mathbf{x}_0, \mathbf{x}_1)$ is drawn from a coupling of p_0 and p_1, $\mathbf{w} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ is independent of both, and the smooth schedules satisfy $a_0 = b_1 = 1$ and $a_1 = b_0 = c_0 = c_1 = 0$. With a Gaussian p_1 the diffusion and rectified-flow paths are the one-sided case $c \equiv 0$; the latent term $c_t\mathbf{w}$ serves two-sided paths between arbitrary p_0 and p_1.
   1. Take $\mathbf{z} = (\mathbf{x}_0, \mathbf{x}_1, \mathbf{w})$ in the marginal velocity Equation 38.94. Show that it is $\mathbb{E}[\dot{a}_t\mathbf{x}_0 + \dot{b}_t\mathbf{x}_1 + \dot{c}_t\mathbf{w} \mid \mathbf{x}_t = \mathbf{x}]$, and write the corresponding CFM loss Equation 38.95.
   2. Give schedules that recover rectified flow Equation 38.99 and the cosine path, whose data and noise coefficients are $\cos\frac{\pi t}{2}$ and $\sin\frac{\pi t}{2}$, both with independent endpoints and $p_1 = \mathcal{N}(\mathbf{0}, \mathbf{I})$.
   3. For $p_1 = \mathcal{N}(\mathbf{0}, \mathbf{I})$, independent endpoints, and $c_t > 0$, show that the interpolant has the same marginal velocity as the interpolant with $c \equiv 0$ and $\sqrt{b_t^2 + c_t^2}$ in place of b_t.
   4. For $c_t > 0$, show that $\nabla \log p_t(\mathbf{x}) = -\mathbb{E}[\mathbf{w} \mid \mathbf{x}_t = \mathbf{x}]/c_t$. What does interior noise provide when p_1 is an arbitrary distribution rather than a Gaussian?
2. **Late decisions.** Take the two-point example with data $\pm a$.
   1. Give the point $+a$ the weight w and the point $-a$ the weight $1 - w$. Repeat the derivation of Equation 38.98 and show that the posterior mean becomes $a\tanh\left((1-t)\,ax/t^2 + \frac{1}{2}\log\frac{w}{1-w}\right)$. Where does one Euler step from $t = 1$ to $t = 0$ land now?
   2. For equal weights, show that $\partial_x u_t(0) = \left(1 - (1-t)\,a^2/t^2\right)/t$ and that, as t falls, the origin turns from attracting to repelling at $t^* = \frac{1}{2}\left(\sqrt{a^4 + 4a^2} - a^2\right)$, which is $2\sqrt{2} - 2 \approx 0.83$ for $a = 2$. How does t^* behave as $a \to \infty$ and as $a \to 0$? Interpret both limits.
   3. Show that away from the origin the slope $\partial_x u_t$ tends to $1/t$, and that an Euler step from $t = h$ down to $t = 0$ lands exactly on $\pm a$ from any point where the tanh is saturated. Which part of the flow, then, limits a fixed-step solver, and at

which times?

4. The floor is $C = \mathbb{E}_t \mathbb{E}_x \mathrm{Var}(x_0 \mid x_t)/t^2$ with $\mathrm{Var}(x_0 \mid x_t = x) = a^2 \operatorname{sech}^2\left((1-t)\,ax/t^2\right)$. Show that $C \le a^2 + 1$. Then explain why the times near $t = 0$ contribute almost nothing to C although the conditional velocity $(x - x_0)/t$ is unbounded there. Use the fact that the density of x_t near 0 is of order $e^{-(1-t)^2 a^2/2t^2}/t$.

3. **Kinetic energy of Gaussian paths.**
   1. Show that if $p_t = \mathcal{N}(0, s_t^2)$ in one dimension, the velocity $u_t(x) = (\dot{s}_t/s_t)\,x$ satisfies the continuity equation, and that its kinetic energy is $\int_0^1 \dot{s}_t^2\,dt$. Why does a Gaussian cloud need no other velocity?
   2. For data $\mathcal{N}(0, 4)$ and noise $\mathcal{N}(0, 1)$, write down s_t for the three flows of the cell: the straight flow of the sorted pairing, the marginal flow of straight segments under the independent pairing, and the cosine path. Show that the first two kinetic energies are exactly 1 and $5 - \pi$.
   3. Repeat the independent pairing for data $\mathcal{N}(0, 1)$. Show that its marginal flow has kinetic energy $2 - \pi/2 \approx 0.43$ although $W_2^2 = 0$. Which pairing costs nothing, and what is its flow?
4. code **One reflow round on two points.** Take data $\pm a$, noise $\mathcal{N}(0, 1)$, the independent pairing, and the marginal flow Equation 38.98 of the two-point example.
   1. Using the order-preservation proposition of Section 38.3.1.2 and symmetry, show that the flow map from $t = 1$ to $t = 0$ is $\epsilon \mapsto a\,\mathrm{sign}(\epsilon)$, so one reflow round produces the sorted pairing.
   2. Show that its cost is $a^2 - 2a\sqrt{2/\pi} + 1$, about 1.81 for $a = 2$, against $a^2 + 1 = 5$ for the independent pairing, and explain why this cost is W_2^2.
   3. Show that the reflowed segments $(1-t)\,a\,\mathrm{sign}(\epsilon) + t\,\epsilon$ never cross, so that the floor of the second round is $C = 0$, and write down its field. Is one Euler step of it exact?
   4. Integrate the closed-form field with RK4 from 20,000 Gaussian draws at $t = 1$ down to $t = 10^{-3}$ (the slope of the field grows like $1/t$, so stop short of 0), and check the endpoints and the cost.
   5. The floor of the independent pairing is $C \approx 2.03$ for $a = 2$. Compute it by quadrature from the formula in part (d) of Exercise 2, and find the time above which 99% of it is collected. Check the bound $\mathbb{E}\|\mathbf{Z}_1 - \mathbf{Z}_0\|^2 \le \mathbb{E}\|\mathbf{X}_1 - \mathbf{X}_0\|^2 - C$ of the reflow proposition. How much slack does it leave here?
5. code **Minibatch optimal-transport pairing.** Replace `independent_pairs` by a sampler that draws its batch in blocks of 256 noise points and 256 data points and re-pairs each block by the assignment of least total squared distance (`scipy.optimize.linear_sum_assignment` on the 256×256 cost matrix), and train with `train_rf`.
   1. Report the loss plateau, the squared energy distance at 1, 2, 8 and 32 steps, and the mean squared pair length before and after re-pairing. Compare with the independent pairing and with reflow.
   2. Show that a re-paired noise point is still a standard Gaussian draw and a re-paired data point is still a draw from the data, so the marginal field still carries p_1 to p_0.
   3. Explain why the batch pairing is not an optimal coupling of the two distributions, and what happens as the block size grows. Which of the two re-pairing methods needs a trained model, and which inherits its errors?
6. **Conditioning of prediction targets.** Take the Gaussian path Equation 38.96 with one-dimensional data $x_0 \sim \mathcal{N}(0, 1)$ and the cosine path.
   1. Compute, as functions of t, the variances of the sampled training targets in Ta-

ble 38.2: the conditional score $-\epsilon/\sigma_t$, the noise ϵ, the clean point x_0, the velocity $\dot\alpha_t x_0 + \dot\sigma_t \epsilon$, and the **v**-target $\alpha_t\epsilon - \sigma_t x_0$. Which blow up, and at which end of the path?

2. A network trained on the noise outputs $\hat\epsilon$ with error δ. Show that the implied clean point $(x - \sigma_t\hat\epsilon)/\alpha_t$ carries the error $\sigma_t\delta/\alpha_t$, and that the noise implied by a clean-point prediction with error δ carries $\alpha_t\delta/\sigma_t$. At which end of the path is each amplified?
3. Show that on the cosine path the velocity target equals $\pi/2$ times the **v**-target, so that **v**-prediction is velocity prediction, and that the velocity implied by a noise prediction carries the error $\pi\delta/(2\alpha_t)$. Which parameterizations keep the error bounded at both ends?
4. The conversions of Table 38.2 hold for any data. Check Equation 38.97 numerically on the mixture $\frac{1}{2}\mathcal{N}(-2, 0.25) + \frac{1}{2}\mathcal{N}(2, 0.25)$ on the cosine path at $t = 0.4$. On a grid of points x, compute the marginal velocity once as the posterior mean of $\dot\alpha_t x_0 + \dot\sigma_t\epsilon$, weighting each component by its posterior probability, and once from the score of the noised mixture. The two should agree to rounding error. Which numbers in the computation change when the mixture weights become 0.8 and 0.2, and which do not?

7. **Reflow preserves marginals and can stall.**
   1. Prove part (i) of the reflow proposition in detail for the straight path, whose conditional laws are point masses. For a smooth test function φ with compact support, show that $\frac{d}{dt}\mathbb{E}\,\varphi(\mathbf{X}_t) = \mathbb{E}[\nabla\varphi(\mathbf{X}_t)\cdot\mathbf{u}_t(\mathbf{X}_t)]$, which is the continuity equation in weak form. Where is the tower property used, and where the Lipschitz hypothesis?
   2. Let π_k be the pairing after k rounds. Show that $\mathbb{E}_{\pi_k}\|\mathbf{x}_1 - \mathbf{x}_0\|^2$ is non-increasing and bounded below by W_2^2, hence convergent.
   3. Show that if a round changes nothing, $\pi_{k+1} = \pi_k$, then $C = 0$ for π_k. Identify the fixed point in one dimension.
   4. In two dimensions pair each $\mathbf{x}_0 \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ with $\mathbf{R}\mathbf{x}_0$, where $\mathbf{R}$ is the quarter-turn rotation. Show that the segments $((1-t)\mathbf{I} + t\mathbf{R})\mathbf{x}_0$ never cross (the matrix is invertible for every t, because its eigenvalues are $(1-t) \pm \mathrm{i}\,t$), so that $C = 0$ and reflow changes nothing, while the cost is $\mathbb{E}\|\mathbf{R}\mathbf{x}_0 - \mathbf{x}_0\|^2 = 4$ against $W_2^2 = 0$. For which matrices $\mathbf{A}$ is the linear map $\mathbf{x} \mapsto \mathbf{A}\mathbf{x}$ the gradient of a convex function, and hence an optimal pairing of $\mathcal{N}(\mathbf{0}, \mathbf{I})$ with its image?

8. code **A stochastic sampler for a flow-matching model.** ●
   1. On the straight path $\alpha_t = 1-t$, $\sigma_t = t$, invert Equation 38.97 to get $\nabla \log p_t(\mathbf{x}) = -(\mathbf{x} + (1-t)\,\mathbf{u}_t(\mathbf{x}))/t$.
   2. Show that for every $g \geq 0$ the SDE $d\mathbf{X} = \big[\mathbf{u}_t(\mathbf{X}) - \frac{1}{2}g^2\nabla \log p_t(\mathbf{X})\big]dt + g\,d\bar{\mathbf{W}}$, integrated from $t = 1$ down to $t = 0$ with a Brownian motion $\bar{\mathbf{W}}$ that runs backward in time, has the marginals p_t. Substitute $\tau = 1-t$, write the Fokker–Planck equation in τ, and use $\nabla\cdot(p\nabla\log p) = \Delta p$. Why does the score term enter with a minus sign here? A trained velocity field therefore defines a family of stochastic samplers.
   3. Implement the sampler with Euler–Maruyama for the trained two-moons field, with $g \in \{0, \frac{1}{2}, 1\}$ and $K \in \{8, 32, 128\}$ steps. Evaluate the score at $t = 1, 1 - 1/K, \ldots, 1/K$ only, because it is singular at $t = 0$, and report the squared energy distance of each run. Which g needs the fewest steps for a given quality, and does noise help at $K = 8$?
   4. Repeat for the reflowed field. Explain, from the straightness of its trajectories,

why no one-step stochastic sample can match its one-step deterministic sample.

38.8 Further Reading

Seven sections took us from the trajectory of one point to a flow that turns noise into data, and they passed a good many questions without stopping. Which stochastic integral describes the noise in a physical system? What becomes of the adjoint when the model is itself an SDE? Each entry below takes up one such question and says where its answer is.

The practice of diffusion models belongs to Chapter 17 and its further reading (Section 17.10), so a method treated there gets a pointer and no more. What remains is mathematics. We would read three entries first: exponential integrators, one reason a diffusion sampler gets by with few steps; Wasserstein gradient flows, which say what a diffusing density descends; and Schrödinger bridges, the stochastic counterpart of an optimal pairing. The rest answer questions a reader may or may not have. A reader who stops at Section 38.7 misses nothing that the rest of the book uses.

38.8.1 Solvers

38.8.1.1 Exponential Integrators

The probability-flow ODE of the VP process with a learned score (Section 38.6.3), $\dot{\mathbf{x}} = -\frac{1}{2}\beta\,\mathbf{x} - \frac{1}{2}\beta\,\mathbf{s}_\theta(\mathbf{x}, t)$, is semilinear: a linear term known in closed form plus a learned part. A Runge–Kutta step discretizes both, so the linear term adds to its error and, through the stability limit of Section 38.2.3, bounds its step, however smooth the learned part is. That is a waste, since the linear term can be solved. An exponential integrator treats the learned part as the input of a driven linear equation (Section 38.1.2.3): the variation-of-constants formula carries the state through the linear term exactly, and only the integral of the learned part is approximated.

Hold the learned part fixed over the step and the result is the zero-order hold of that subsection, with the network's output as the input. The step would be exact if that output did not change, and it is of first order because it does. The chapter has taken this step once without the name: dividing by $\sqrt{\bar{\alpha}_t}$ in Section 38.6.4 removed the linear term, and DDIM holds the noise prediction fixed over a stride. DDIM is the exponential integrator of first order.

The samplers of this family differ in which output they hold (the score, the noise prediction or the prediction of the clean point) and in which clock. Methods of higher order replace the constant by a polynomial through several predictions. Hochbruck and Ostermann (2010) survey the theory. The diffusion samplers built on it include DEIS (Zhang and Chen 2023) and DPM-Solver (C. Lu, Zhou, et al. 2022). The second is the subject of Exercise 3 of Section 17.5, and its successors are listed in Section 17.10. For an SDE the same idea gives the exponential Euler–Maruyama step of Exercise 7 of Section 38.4.

38.8.1.2 Geometric Integration and Leapfrog

On a frictionless oscillator forward Euler gains energy at every step and backward Euler loses it, so in a long run the first flies off and the second comes to rest. A method of higher order only postpones this. *Geometric integration* asks something else of a method: that it preserve a structure of the exact flow, whether or not it follows the flow closely. A *symplectic*

method, such as symplectic Euler or leapfrog, preserves area in the plane of position and momentum, as the exact flow does (with more coordinates the area becomes a volume).

Backward error analysis then shows that such a method follows, up to an error that is exponentially small in $1/h$, the exact flow of a slightly different energy. Its error in the true energy therefore stays bounded over very long times instead of growing with the horizon. Exercise 4 of Section 38.2 finds the different energy for symplectic Euler on the oscillator: its level sets are ellipses close to the circles of the true one. Gradient descent has the same analysis with a loss in place of the energy, and the term it adds to the loss is the implicit gradient regularizer of Barrett and Dherin (2021) (Exercise 5 of the same section). Hairer et al. (2006) develop the theory.

Leapfrog is also the integrator inside Hamiltonian Monte Carlo (Duane et al. 1987; Neal 2011). This sampler proposes a distant move by simulating a frictionless particle whose potential energy is the negative log-density of the target, and accepts or rejects it by the Metropolis rule (Section 36.8.3). The move can be accepted because the energy error of leapfrog stays bounded. And since a leapfrog step is reversible and preserves volume, the rule compares two energies and needs no Jacobian.

38.8.2 Continuous-Time Models

38.8.2.1 Flows with Cheap Determinants

Discrete normalizing flows restrict each layer so that the determinant of its Jacobian is cheap, and Section 38.3.3 named the usual restriction, a triangular Jacobian, without saying how a layer comes by one. A coupling layer (Dinh et al. 2014; Dinh, Sohl-Dickstein, et al. 2017) leaves half of the coordinates unchanged and shifts and scales the other half by functions of the first. Its Jacobian is triangular, and the layer is inverted by a subtraction and a division. One layer changes only half of the state, so many are stacked, and the halves trade places from layer to layer.

An invertible residual network (Behrmann et al. 2019) restricts something else. A residual block is an Euler step (Section 38.3.1), and a branch whose Lipschitz constant is below one makes that step invertible, as the flow of a Lipschitz field is (Section 38.1.1.4). The Jacobian keeps its general shape, and its log-determinant is estimated by the power series of Exercise 4 of Section 38.3. Papamakarios et al. (2021) survey the families and what each gives up.

38.8.2.2 Reversible Solvers and Regularized Flows

The adjoint method of Section 38.3.2 saves memory by rebuilding the forward states on the way back, and a contracting flow amplifies the rounding error of that reconstruction. Rebuilding has a second error, which needs no contraction. A backward step of the solver is close to the inverse of a forward step but is not the inverse. The states that come back are therefore not the ones that went out, and the gradient is not that of the computation that was run. A reversible solver is built so that each step can be undone algebraically, which removes this second error. Rounding is still amplified, so for a stiff field checkpointing remains the safer choice; Kidger (2022) (ch. 5) compares the two.

A flow trained by likelihood has no reason to be cheap to integrate: the likelihood depends on where each point ends up, not on the route it took. Finlay et al. (2020) therefore penalize the kinetic energy $\int \mathbb{E}\|\mathbf{f}_\theta\|^2 \, dt$ during training. Between two fixed endpoints the straight path at constant speed has the least kinetic energy, so the penalty straightens the paths, and

a solver crosses a straight path in few steps. Flow matching pursues the same goal through its straight regression targets and the pairing of noise with data (Section 38.7.3).

38.8.2.3 Latent and Controlled Differential Equations

The chapter's models map an initial state to a final one. A time series, such as a patient's record, arrives at irregular times, and a model of it should take in each observation as it comes. A latent ODE (Rubanova et al. 2019) does half of this. It encodes the observations into one initial state and decodes the trajectory at any set of times, so an observation that arrives afterwards cannot change the course.

A neural controlled differential equation (Kidger et al. 2020) lets every observation act. It drives the state by the interpolated data path $\mathbf{X}_t$ itself, $d\mathbf{z} = \mathbf{f}_\theta(\mathbf{z})\, d\mathbf{X}_t$, with a matrix-valued $\mathbf{f}_\theta$. The equation has the form of an SDE with the data in place of the Brownian path, but the interpolated path is smooth, so the integral is the ordinary one. Its Euler step, $\mathbf{z}_{n+1} = \mathbf{z}_n + \mathbf{f}_\theta(\mathbf{z}_n)(\mathbf{X}_{n+1} - \mathbf{X}_n)$, is a recurrent network that reads the increments of the data.

38.8.2.4 Transformers as Interacting Particles

A residual network moves one state by Euler steps (Section 38.3.1). A transformer moves all the tokens of a sequence at once, and attention makes the velocity of each token depend on where the others are. The tokens are particles that interact. Geshkovski et al. (2023) study this system in an idealized model, in which layer normalization keeps every token on the unit sphere, and show that the tokens gather into clusters as depth grows. The one difference from the chapter is that the field depends on the density it moves, so the continuity equation is no longer linear in the density.

38.8.3 Noise

38.8.3.1 Wasserstein Gradient Flows

A point on a gradient flow descends its loss (Section 38.2.4). The density of a diffusion also moves deterministically, by the Fokker–Planck equation. Does it descend anything? It does when the drift is a gradient, $\mathbf{f} = -\nabla V$, and g is constant. The density then descends the free energy $\mathcal{F}(p) = \int V p \, d\mathbf{x} + \frac{g^2}{2} \int p \log p \, d\mathbf{x}$. The first term is the mean potential energy, which is lowest when all the mass sits at the bottom of V. The second is the negative entropy, which is lowest when the mass is spread out. The drift serves the first and the noise the second, and the two balance at the stationary density, which is proportional to e^{-2V/g^2}.

Steepest descent depends on how distance is measured, and here the distance between two densities has to be the Wasserstein-2 distance, which measures the cost of transporting one into the other. The backward Euler step of Section 38.2.4 shows where the distance enters: for a convex loss it minimizes $\mathcal{L}(\mathbf{x}) + \frac{1}{2\eta}\|\mathbf{x} - \mathbf{x}_k\|^2$. The JKO scheme (Jordan et al. 1998) is this step with densities in place of points and the cost of transport in place of the squared distance, $p_{k+1} = \arg\min_p \, \mathcal{F}(p) + \frac{1}{2\eta} W_2^2(p, p_k)$. As $\eta \to 0$ its steps converge to the solution of the Fokker–Planck equation.

Langevin dynamics is the case to remember. Its drift is half the score of its target and its noise has $g = 1$ (Section 38.6.2). Write p_∞ for the target, to keep p for the density of the cloud. Then $V = -\frac{1}{2}\log p_\infty$, and the free energy is $\frac{1}{2} D_{\mathrm{KL}}(p \,\|\, p_\infty)$. A cloud of Langevin particles runs gradient descent on its KL divergence to the target, with distance measured

by transport. This is why transport, from the continuity equation to Benamou–Brenier (Benamou and Brenier 2000), keeps reappearing in the chapter.

38.8.3.2 Physical Noise and the Wong–Zakai Theorem

Section 38.4.2.2 defines the Itô integral with the left endpoint and notes that the midpoint rule, which defines the Stratonovich integral, keeps the ordinary chain rule. Which of the two describes a physical system? The noise in a physical system is not a Brownian path. It has a short memory and a smooth path, and for a smooth path there is only one integral, the ordinary one. Wong and Zakai (1965) proved that as such paths approach a Brownian path, the solutions of the ordinary equations converge to the Stratonovich solution. In one dimension the Stratonovich reading of $dX = f\,dt + g\,dW$ is the Itô equation with the drift $f + \frac{1}{2} g\,\partial_x g$.

The two readings agree for the additive noise of diffusion models. For multiplicative noise they differ, and the theorem says how to choose. An equation that idealizes smooth physical noise is read as Stratonovich. A model built in discrete time, whose coefficient cannot see the increment it multiplies, is read as Itô, and a step of stochastic gradient descent is of this kind.

38.8.3.3 Stochastic Gradient Descent as an SDE

Gradient descent is forward Euler on the gradient flow (Section 38.2.4). Of what is stochastic gradient descent the discretization? A minibatch gradient is the full gradient plus noise, with a covariance Σ that shrinks as $1/B$ with the batch size B. Q. Li et al. (2017) propose the SDE $d\theta = -\nabla \mathcal{L}\,dt + \sqrt{\eta}\,\Sigma^{1/2} d\mathbf{W}$. Take an Euler–Maruyama step of length η. It multiplies the noise coefficient by another $\sqrt{\eta}$, and the result is one step of stochastic gradient descent with learning rate η.

The learning rate therefore appears twice, as the length of the step and inside the noise. When Σ is the same multiple $c\,\mathbf{I}$ of the identity everywhere, the free energy above applies with $V = \mathcal{L}$ and $g^2 = \eta c$, which weighs the entropy against the loss. In this sense the learning rate is a temperature. On a quadratic loss the iterates settle into the stationary Gaussian of an OU process (Section 38.4.3), whose variance grows with the ratio η/B. Mandt et al. (2017) use this to run stochastic gradient descent as an approximate posterior sampler. The model needs small learning rates and nearly Gaussian gradient noise, and both assumptions can fail in practice.

38.8.3.4 Neural SDEs

What becomes of the adjoint when the learned model is itself an SDE, with a learned drift and a learned noise amplitude g? It becomes a backward SDE driven by the same Brownian path, which must be stored or regenerated from its random seed (X. Li et al. 2020). The loss changes as well. The output of such a model is random, so one endpoint cannot be compared with one target. The training signal is a divergence between the laws of whole paths, or a discriminator that compares them (Kidger 2022).

Girsanov's theorem (Øksendal 2003) gives this divergence in closed form for two processes with the same g and the drifts $\mathbf{f}$ and $\mathbf{f}'$. The KL divergence between their path laws is the KL divergence between their starting laws plus $\frac{1}{2}\int \mathbb{E}\|\mathbf{f} - \mathbf{f}'\|^2/g^2\,dt$, with the expectation over paths of the first process. The condition on g cannot be dropped: two processes with

different g can be told apart from a single path, by its quadratic variation (Section 38.4.2.1), and the KL divergence between their path laws is infinite.

The laws of the endpoints cannot differ by more than the laws of the paths, by the data-processing inequality of Section 37.3.3.3. Applied to the exact and the learned reverse drift of a diffusion model, Girsanov's formula therefore gives the likelihood bound that Exercise 7 of Section 38.6 derives without path laws.

38.8.4 Learned Generators

Most of what follows the chapter in this direction is taught in Chapter 17. Prediction targets, loss weightings and schedules are in Section 17.6, where the logarithm of the signal-to-noise ratio, $-2\log\bar{\sigma}_t$ in the notation of Section 38.6.4, orders the choices (Kingma et al. 2021). Generation in one or a few steps is in Section 17.8. Four topics lie beyond both chapters.

38.8.4.1 Sliced Score Matching

Section 38.6.1 set implicit score matching aside for the cost of its trace, and mentioned a way out that it did not take: Hutchinson's estimate, one vector–Jacobian product per random direction. Trained with that estimate, the method is called sliced score matching (Song et al. 2019). It learns the score of the data themselves, with no noise level to choose. A diffusion model has no use for this advantage. It needs the scores of the noised data at every level, and denoising score matching delivers those without a trace, which is why denoising became the standard.

38.8.4.2 Score Error and Sampling Error

Nearly every sampler of the chapter ran on an exact score, and a trained score is exact nowhere. How good must it be? Good on average, and no more. Sitan Chen et al. (2023) bound the total variation between the data and the output of the discretized reverse SDE by a sum of three terms, and the chapter has met each of them alone. The distance of p_T from the Gaussian at which the sampler starts decays like e^{-T}, and the discretization error grows with the dimension and falls with the step; Section 38.5.5 measured both. The third term is the root mean squared error of the score under the noised data, which is what a denoising loss measures above its floor. The theorem says that nothing else enters, and it asks little: a finite second moment of the data and Lipschitz scores of the noised densities. A single mode is not required, so the two modes that defeated Langevin dynamics in Section 38.6.2 are covered.

38.8.4.3 Schrödinger Bridges and Entropic Transport

In 1931 Schrödinger asked a question about particles. A cloud that diffuses freely is seen in one shape now and in another, improbable one later: what is the most likely way it got there? For a large cloud of independent particles the probability of seeing the cloud follow some other law of paths falls exponentially, at a rate given by the KL divergence of that law from the free diffusion. The answer is therefore the law of paths that is closest in KL divergence to the free diffusion among those with the two given marginals. It is called the *Schrödinger bridge*.

Set the bridge beside the minibatch pairing of Section 38.7.3. That pairing is an exact assignment on each batch, and the generator it trains is an ODE. The bridge pairs its two ends

by the plan of entropy-regularized optimal transport, computed by the Sinkhorn iterations of Section 37.4.2.4 (Cuturi 2013; Peyré and Cuturi 2019), and its generator is an SDE. As the noise of the free diffusion shrinks, the plan tends to an optimal one.

Set beside a diffusion model, the bridge is the harder problem. A diffusion model fixes its forward process, learns only the reverse, and reaches its prior only approximately. The bridge joins two given laws in finite time, so both directions must be learned. De Bortoli et al. (2021) fit forward and backward diffusions in alternation, and the first round of the alternation is the training of a diffusion model. Albergo et al. (2023) reach both kinds of generator, ODE and SDE, without solving a bridge problem. Take samples $\mathbf{x}_0$ and $\mathbf{x}_1$ of any two distributions and an independent Gaussian $\mathbf{w}$. The path $a_t\mathbf{x}_0 + b_t\mathbf{x}_1 + c_t\mathbf{w}$, with c_t zero at both ends, has marginals that an ODE or a family of SDEs can generate (Exercise 1 of Section 38.7).

38.8.4.4 Straight Couplings and Optimal Couplings

A pairing that reflow leaves unchanged has straight paths that do not cross, and in more than one dimension it need not be optimal. The example of Section 38.7.3 pairs each point of a Gaussian cloud with its quarter turn. The segments never cross, and they cost 4 where $W_2^2 = 0$. What that pairing has and an optimal one lacks is rotation. The velocity field of an optimal flow is a gradient (Benamou and Brenier 2000), and a field that turns a cloud about the origin is not.

Liu (2022) therefore restricts the velocity field to a gradient $\nabla\phi_t$ and argues that the pairings this restricted reflow leaves unchanged are the optimal ones. The statement needs more than it assumes. Hertrich et al. (2025) prove it when the cloud in transit covers all of space at every time, and give counterexamples in which the cloud consists of separate pieces. So the restriction does not settle the question that Section 38.7.3 left: how to find an optimal pairing by training alone.

38.8.5 General Dynamics and Generative Flow Resources

One warning before opening the references below: the MIT notes and course, the Flow Matching Guide, and the two flow-matching papers run time from noise at 0 to data at 1, whereas this chapter runs it the other way (Figure 38.26).

Books and lecture notes

- Nonlinear Dynamics and Chaos — Steven Strogatz: a geometric introduction to flows, fixed points and their stability, the material of Section 38.1.
- On Neural Differential Equations — Patrick Kidger (2022): the book-length treatment of neural ODEs, controlled and stochastic differential equations, with the adjoint machinery of Section 38.3 laid out in full and advice on choosing among the solvers of Section 38.2.
- Applied Stochastic Differential Equations — Särkkä & Solin: an application-minded treatment of Itô calculus, SDEs and the Fokker–Planck equation for Section 38.4 and Section 38.5; the authors provide the full PDF.
- Stochastic Differential Equations: An Introduction with Applications — Bernt Øksendal: a rigorous reference for Brownian motion, the Itô integral and existence theory, behind Section 38.4.
- Numerical Solution of Stochastic Differential Equations — Kloeden & Platen: the reference on Euler–Maruyama, Milstein and higher-order schemes and their strong and

weak orders, for the simulation part of Section 38.4.
- An Introduction to Flow Matching and Diffusion Models (MIT 6.S184 lecture notes) — Holderrieth & Erives: self-contained notes that derive diffusion and flow matching from SDEs and the continuity equation, companions to Section 38.6 and Section 38.7.

Courses and tutorials

- MIT 6.S184: Generative AI with Stochastic Differential Equations: lectures, slides and labs that build a diffusion model from first principles, following the route of Section 38.5 to Section 38.7; openly licensed.
- Understanding Diffusion Models: A Unified Perspective — Calvin Luo (2022): derives the variational and score-based views of diffusion side by side, for Section 38.6.
- Flow Matching Guide and Code — Lipman et al. (2024): a detailed reference on flow matching for Section 38.7, paired with the `facebookresearch/flow_matching` PyTorch library.

Foundational papers

- Neural Ordinary Differential Equations — Chen et al. (NeurIPS 2018): parameterizes a network's dynamics as an ODE and trains it with the adjoint method, as in Section 38.3.
- Generative Modeling by Estimating Gradients of the Data Distribution — Song & Ermon (NeurIPS 2019): score-based generation with denoising score matching and annealed Langevin sampling, for Section 38.6.
- Denoising Diffusion Probabilistic Models — Ho, Jain & Abbeel (NeurIPS 2020): the discrete-time DDPM formulation that Section 38.6 reads as a discretized SDE.
- Score-Based Generative Modeling through SDEs — Song et al. (ICLR 2021): unifies score matching and diffusion as forward and reverse SDEs and introduces the probability-flow ODE of Section 38.5.
- Denoising Diffusion Implicit Models — Song, Meng & Ermon (ICLR 2021): the deterministic sampler of Section 38.6, which reuses a trained DDPM with fewer steps.
- Flow Matching for Generative Modeling — Lipman et al. (ICLR 2023): simulation-free training of continuous flows by regressing conditional vector fields, the method of Section 38.7.
- Flow Straight and Fast: Rectified Flow — Liu, Gong & Liu (ICLR 2023): straight interpolation paths and reflow for few-step sampling, for Section 38.7.

Blogs and visual explainers

- Generative Modeling by Estimating Gradients of the Data Distribution — Yang Song: the author's walkthrough connecting score matching, Langevin dynamics and the SDE picture of Section 38.6, with figures.
- What are Diffusion Models? — Lilian Weng: a survey of diffusion from DDPM through score SDEs and guidance, updated through 2024, for Section 38.6.
- Perspectives on Diffusion — Sander Dieleman: shows how a diffusion model is at once a denoiser, a score estimator, a reverse-SDE solver and a flow, the dictionary of Section 38.7.1.3.
- The Annotated Diffusion Model — Hugging Face: a line-by-line PyTorch implementation of DDPM that connects the mathematics of Section 38.6 to runnable code.

References

Albergo, Michael S., Nicholas M. Boffi, and Eric Vanden-Eijnden. 2023. "Stochastic Interpolants: A Unifying Framework for Flows and Diffusions." *arXiv Preprint arXiv:2303.08797*.

Alemi, Alexander A., Ian Fischer, Joshua V. Dillon, and Kevin Murphy. 2017. "Deep Variational Information Bottleneck." *International Conference on Learning Representations*.

Amari, Shun-ichi. 1998. "Natural Gradient Works Efficiently in Learning." *Neural Computation* 10 (2): 251–76.

Amari, Shun-ichi. 2016. *Information Geometry and Its Applications*. Springer.

Anderson, Brian D. O. 1982. "Reverse-Time Diffusion Equation Models." *Stochastic Processes and Their Applications* 12 (3): 313–26.

Angelopoulos, Anastasios N., and Stephen Bates. 2023. "Conformal Prediction: A Gentle Introduction." *Foundations and Trends in Machine Learning* 16 (4): 494–591. https://doi.org/10.1561/2200000101.

Arıkan, Erdal. 2009. "Channel Polarization: A Method for Constructing Capacity-Achieving Codes for Symmetric Binary-Input Memoryless Channels." *IEEE Transactions on Information Theory* 55 (7): 3051–73. https://doi.org/10.1109/TIT.2009.2021379.

Arjovsky, Martin, Soumith Chintala, and Léon Bottou. 2017. "Wasserstein Generative Adversarial Networks." *Proceedings of the 34th International Conference on Machine Learning*, 214–23.

Armijo, Larry. 1966. "Minimization of Functions Having Lipschitz Continuous First Partial Derivatives." *Pacific Journal of Mathematics* 16 (1): 1–3.

Ba, Jimmy Lei, Jamie Ryan Kiros, and Geoffrey E Hinton. 2016. "Layer Normalization." *ArXiv:1607.06450*. https://arxiv.org/abs/1607.06450.

Bach, Francis, and Eric Moulines. 2013. "Non-Strongly-Convex Smooth Stochastic Approximation with Convergence Rate $O(1/n)$." *Advances in Neural Information Processing Systems* 26.

Bai, Shaojie, J. Zico Kolter, and Vladlen Koltun. 2019. "Deep Equilibrium Models." *Advances*

in Neural Information Processing Systems 32.

Baik, Jinho, and Jack W. Silverstein. 2006. "Eigenvalues of Large Sample Covariance Matrices of Spiked Population Models." *Journal of Multivariate Analysis* 97 (6): 1382–408. https://doi.org/10.1016/j.jmva.2005.08.003.

Baldi, Pierre, and Kurt Hornik. 1989. "Neural Networks and Principal Component Analysis: Learning from Examples Without Local Minima." *Neural Networks* 2 (1): 53–58.

Ballé, Johannes, Valero Laparra, and Eero P. Simoncelli. 2017. "End-to-End Optimized Image Compression." *International Conference on Learning Representations*.

Ballé, Johannes, David Minnen, Saurabh Singh, Sung Jin Hwang, and Nick Johnston. 2018. "Variational Image Compression with a Scale Hyperprior." *International Conference on Learning Representations*.

Barber, David, and Felix Agakov. 2003. "The IM Algorithm: A Variational Approach to Information Maximization." *Advances in Neural Information Processing Systems* 16.

Barrett, David G. T., and Benoit Dherin. 2021. "Implicit Gradient Regularization." *International Conference on Learning Representations*.

Bartlett, Peter L., Philip M. Long, Gábor Lugosi, and Alexander Tsigler. 2020. "Benign Overfitting in Linear Regression." *Proceedings of the National Academy of Sciences* 117 (48): 30063–70.

Bartlett, Peter L., and Shahar Mendelson. 2002. "Rademacher and Gaussian Complexities: Risk Bounds and Structural Results." *Journal of Machine Learning Research* 3: 463–82.

Baur, Walter, and Volker Strassen. 1983. "The Complexity of Partial Derivatives." *Theoretical Computer Science* 22 (3): 317–30.

Baydin, Atılım Güneş, Barak A. Pearlmutter, Don Syme, Frank Wood, and Philip Torr. 2022. "Gradients Without Backpropagation." *arXiv Preprint arXiv:2202.08587*.

Beck, Amir, and Marc Teboulle. 2003. "Mirror Descent and Nonlinear Projected Subgradient Methods for Convex Optimization." *Operations Research Letters* 31 (3): 167–75. https://doi.org/10.1016/S0167-6377(02)00231-6.

Beck, Amir, and Marc Teboulle. 2009. "A Fast Iterative Shrinkage-Thresholding Algorithm for Linear Inverse Problems." *SIAM Journal on Imaging Sciences* 2 (1): 183–202.

Beck, Amir, and Luba Tetruashvili. 2013. "On the Convergence of Block Coordinate Descent Type Methods." *SIAM Journal on Optimization* 23 (4): 2037–60.

Behrmann, Jens, Will Grathwohl, Ricky T. Q. Chen, David Duvenaud, and Jörn-Henrik Jacobsen. 2019. "Invertible Residual Networks." *International Conference on Machine Learning*, 573–82.

Belghazi, Mohamed Ishmael, Aristide Baratin, Sai Rajeshwar, et al. 2018. "Mutual Infor-

mation Neural Estimation." *Proceedings of the 35th International Conference on Machine Learning*, 531–40.

Benamou, Jean-David, and Yann Brenier. 2000. "A Computational Fluid Mechanics Solution to the Monge–Kantorovich Mass Transfer Problem." *Numerische Mathematik* 84 (3): 375–93.

Benaych-Georges, Florent, and Raj Rao Nadakuditi. 2012. "The Singular Values and Vectors of Low Rank Perturbations of Large Rectangular Random Matrices." *Journal of Multivariate Analysis* 111: 120–35. https://doi.org/10.1016/j.jmva.2012.04.019.

Bengio, Yoshua, Nicholas Léonard, and Aaron Courville. 2013. "Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation." *arXiv Preprint arXiv:1308.3432*.

Bengio, Yoshua, Patrice Simard, and Paolo Frasconi. 1994. "Learning Long-Term Dependencies with Gradient Descent Is Difficult." *IEEE Transactions on Neural Networks* 5 (2): 157–66. https://doi.org/10.1109/72.279181.

Benjamini, Yoav, and Yosef Hochberg. 1995. "Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing." *Journal of the Royal Statistical Society: Series B (Methodological)* 57 (1): 289–300.

Bernstein, Jeremy, and Laker Newhouse. 2024. "Old Optimizer, New Norm: An Anatomy." *arXiv Preprint arXiv:2409.20325*.

Bertoin, David, Jérôme Bolte, Sébastien Gerchinovitz, and Edouard Pauwels. 2021. "Numerical Influence of ReLU'(0) on Backpropagation." *Advances in Neural Information Processing Systems* 34: 468–79. https://arxiv.org/abs/2106.12915.

Billingsley, Patrick. 1995. *Probability and Measure*. 3rd ed. Wiley.

Biroli, Giulio, Tony Bonnaire, Valentin de Bortoli, and Marc Mézard. 2024. "Dynamical Regimes of Diffusion Models." *Nature Communications* 15: 9957. https://arxiv.org/abs/2402.18491.

Bishop, Christopher M. 2006. *Pattern Recognition and Machine Learning*. Springer. https://doi.org/10.1007/978-0-387-45528-0.

Black, Fischer, and Myron Scholes. 1973. "The Pricing of Options and Corporate Liabilities." *Journal of Political Economy* 81: 637–54. https://doi.org/10.1086/260062.

Blier, Léonard, and Yann Ollivier. 2018. "The Description Length of Deep Learning Models." *Advances in Neural Information Processing Systems* 31. https://arxiv.org/abs/1802.07044.

Blum, Avrim, and Moritz Hardt. 2015. "The Ladder: A Reliable Leaderboard for Machine Learning Competitions." *Proceedings of the 32nd International Conference on Machine Learning*, Proceedings of machine learning research, vol. 37: 1006–14.

Bogachev, Vladimir I., Nicolai V. Krylov, Michael Röckner, and Stanislav V. Shaposhnikov.

2015. *Fokker–Planck–Kolmogorov Equations*. Vol. 207. Mathematical Surveys and Monographs. American Mathematical Society.

Bolte, Jérôme, and Edouard Pauwels. 2021. "Conservative Set Valued Fields, Automatic Differentiation, Stochastic Gradient Methods and Deep Learning." *Mathematical Programming* 188 (1): 19–51.

Bonferroni, Carlo E. 1936. "Teoria Statistica Delle Classi e Calcolo Delle Probabilità." *Pubblicazioni Del R. Istituto Superiore Di Scienze Economiche e Commerciali Di Firenze* 8: 3–62.

Bonneel, Nicolas, Julien Rabin, Gabriel Peyré, and Hanspeter Pfister. 2015. "Sliced and Radon Wasserstein Barycenters of Measures." *Journal of Mathematical Imaging and Vision* 51 (1): 22–45. https://doi.org/10.1007/s10851-014-0506-3.

Boser, Bernhard E., Isabelle M. Guyon, and Vladimir N. Vapnik. 1992. "A Training Algorithm for Optimal Margin Classifiers." *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, 144–52.

Bottou, Léon, Frank E. Curtis, and Jorge Nocedal. 2018. "Optimization Methods for Large-Scale Machine Learning." *SIAM Review* 60 (2): 223–311.

Boucheron, Stéphane, Olivier Bousquet, and Gábor Lugosi. 2005. "Theory of Classification: A Survey of Some Recent Advances." *ESAIM: Probability and Statistics* 9: 323–75. https://doi.org/10.1214/154957805100000046.

Boucheron, Stéphane, Gábor Lugosi, and Pascal Massart. 2013. *Concentration Inequalities: A Nonasymptotic Theory of Independence*. Oxford University Press.

Bousquet, Olivier, and André Elisseeff. 2002. "Stability and Generalization." *Journal of Machine Learning Research* 2: 499–526. https://www.jmlr.org/papers/v2/bousquet02a.html.

Boyd, Stephen, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. 2011. "Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers." *Foundations and Trends in Machine Learning* 3 (1): 1–122. https://doi.org/10.1561/2200000016.

Boyd, Stephen, and Lieven Vandenberghe. 2004. *Convex Optimization*. Cambridge University Press. https://web.stanford.edu/~boyd/cvxbook/.

Bray, Alan J., and David S. Dean. 2007. "Statistics of Critical Points of Gaussian Fields on Large-Dimensional Spaces." *Physical Review Letters* 98 (15): 150201.

Brenier, Yann. 1991. "Polar Factorization and Monotone Rearrangement of Vector-Valued Functions." *Communications on Pure and Applied Mathematics* 44 (4): 375–417.

Bretagnolle, Jean, and Catherine Huber. 1978. "Estimation Des Densités: Risque Minimax." In *Séminaire de Probabilités XII*, vol. 649. Lecture Notes in Mathematics. Springer.

Brin, Sergey, and Lawrence Page. 1998. "The Anatomy of a Large-Scale Hypertextual Web Search Engine." *Computer Networks and ISDN Systems* 30 (1–7): 107–17.

Burda, Yuri, Roger Grosse, and Ruslan Salakhutdinov. 2016. "Importance Weighted Autoencoders." *International Conference on Learning Representations*. https://arxiv.org/abs/1509.00519.

Candès, Emmanuel J., and Benjamin Recht. 2009. "Exact Matrix Completion via Convex Optimization." *Foundations of Computational Mathematics* 9 (6): 717–72.

Canonne, Clément L. 2022. "A Short Note on an Inequality Between KL and TV." *arXiv Preprint arXiv:2202.07198*. https://arxiv.org/abs/2202.07198.

Carson, Erin, and Nicholas J. Higham. 2018. "Accelerating the Solution of Linear Systems by Iterative Refinement in Three Precisions." *SIAM Journal on Scientific Computing* 40 (2): A817–47.

Čencov, N. N. 1982. *Statistical Decision Rules and Optimal Inference*. Vol. 53. Translations of Mathematical Monographs. American Mathematical Society.

Chebyshev, Pafnuty L. 1867. "Des Valeurs Moyennes." *Journal de Mathématiques Pures Et Appliquées* 12 (2): 177–84.

Chechik, Gal, Amir Globerson, Naftali Tishby, and Yair Weiss. 2005. "Information Bottleneck for Gaussian Variables." *Journal of Machine Learning Research* 6: 165–88.

Chen, Lei, and Joan Bruna. 2023. "Beyond the Edge of Stability via Two-Step Gradient Updates." *International Conference on Machine Learning*, 4330–91.

Chen, Ricky T. Q., Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. 2018. "Neural Ordinary Differential Equations." *Advances in Neural Information Processing Systems* 31.

Chen, Sitan, Sinho Chewi, Jerry Li, Yuanzhi Li, Adil Salim, and Anru R. Zhang. 2023. "Sampling Is as Easy as Learning the Score: Theory for Diffusion Models with Minimal Data Assumptions." *International Conference on Learning Representations*.

Chen, Tianqi, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. "Training Deep Nets with Sublinear Memory Cost." *arXiv Preprint arXiv:1604.06174*.

Chen, Ting, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. 2020. "A Simple Framework for Contrastive Learning of Visual Representations." *Proceedings of the 37th International Conference on Machine Learning*, 1597–607.

Cheng, Pengyu, Weituo Hao, Shuyang Dai, Jiachang Liu, Zhe Gan, and Lawrence Carin. 2020. "CLUB: A Contrastive Log-Ratio Upper Bound of Mutual Information." *Proceedings of the 37th International Conference on Machine Learning*, Proceedings of machine learning research, vol. 119: 1779–88.

Church, Kenneth Ward, and Patrick Hanks. 1990. "Word Association Norms, Mutual Information, and Lexicography." *Computational Linguistics* 16 (1): 22–29.

Church, Kenneth W., and William A. Gale. 1995. "Poisson Mixtures." *Natural Language Engineering* 1 (2): 163–90. https://doi.org/10.1017/S1351324900000139.

Chwialkowski, Kacper, Heiko Strathmann, and Arthur Gretton. 2016. "A Kernel Test of Goodness of Fit." *Proceedings of the 33rd International Conference on Machine Learning*, 2606–15.

Coddington, Earl A., and Norman Levinson. 1955. *Theory of Ordinary Differential Equations*. McGraw-Hill.

Cohen, Jeremy M., Simran Kaur, Yuanzhi Li, J. Zico Kolter, and Ameet Talwalkar. 2021. "Gradient Descent on Neural Networks Typically Occurs at the Edge of Stability." *International Conference on Learning Representations*.

Cohen, Joel E., and Charles M. Newman. 1984. "The Stability of Large Random Matrices and Their Products." *The Annals of Probability* 12 (2): 283–310. https://doi.org/10.1214/aop/1176993291.

Cover, T, and JM Thomas. 1999. *Elements of Information Theory*. John Wiley & Sons. https://doi.org/10.1002/047174882X.

Cramér, H. 1946. *Mathematical Methods of Statistics.* Princeton University Press.

Csiszár, Imre. 1967. "Information-Type Measures of Difference of Probability Distributions and Indirect Observations." *Studia Scientiarum Mathematicarum Hungarica* 2: 299–318.

Csiszár, Imre. 2008. "Axiomatic Characterizations of Information Measures." *Entropy* 10 (3): 261–73. https://doi.org/10.3390/e10030261.

Cuturi, Marco. 2013. "Sinkhorn Distances: Lightspeed Computation of Optimal Transport." *Advances in Neural Information Processing Systems* 26.

Dahlquist, Germund G. 1963. "A Special Stability Problem for Linear Multistep Methods." *BIT Numerical Mathematics* 3: 27–43.

Damian, Alex, Eshaan Nichani, and Jason D. Lee. 2023. "Self-Stabilization: The Implicit Bias of Gradient Descent at the Edge of Stability." *International Conference on Learning Representations*.

Darmois, Georges. 1935. "Sur Les Lois de Probabilité à Estimation Exhaustive." *Comptes Rendus de l'Académie Des Sciences* 200: 1265–66.

Dasgupta, Sanjoy, and Anupam Gupta. 2003. "An Elementary Proof of a Theorem of Johnson and Lindenstrauss." *Random Structures & Algorithms* 22 (1): 60–65.

Daskalakis, Constantinos, Andrew Ilyas, Vasilis Syrgkanis, and Haoyang Zeng. 2018. "Training GANs with Optimism." *International Conference on Learning Representations*.

Dauphin, Yann N., Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. 2014. "Identifying and Attacking the Saddle Point Problem in High-Dimensional Non-Convex Optimization." *Advances in Neural Information Processing Systems* 27: 2933–41.

Daxberger, Erik, Agustinus Kristiadi, Alexander Immer, Runa Eschenhagen, Matthias Bauer, and Philipp Hennig. 2021. "Laplace Redux – Effortless Bayesian Deep Learning." *Advances in Neural Information Processing Systems* 34. https://arxiv.org/abs/2106.14806.

De Bortoli, Valentin, James Thornton, Jeremy Heng, and Arnaud Doucet. 2021. "Diffusion Schrödinger Bridge with Applications to Score-Based Generative Modeling." *Advances in Neural Information Processing Systems* 34: 17695–709.

Dean, David S., and Satya N. Majumdar. 2006. "Large Deviations of Extreme Eigenvalues of Random Matrices." *Physical Review Letters* 97 (16): 160201.

DeepSeek-AI. 2024. "DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model." *arXiv Preprint arXiv:2405.04434*.

Defazio, Aaron, Francis Bach, and Simon Lacoste-Julien. 2014. "SAGA: A Fast Incremental Gradient Method with Support for Non-Strongly Convex Composite Objectives." *Advances in Neural Information Processing Systems* 27.

Defazio, Aaron, Ashok Cutkosky, Harsh Mehta, and Konstantin Mishchenko. 2023. "Optimal Linear Decay Learning Rate Schedules and Further Refinements." *arXiv Preprint arXiv:2310.07831*.

Delétang, Grégoire, Anian Ruoss, Paul-Ambroise Duquenne, et al. 2023. "Language Modeling Is Compression." *ArXiv:2309.10668*. https://arxiv.org/abs/2309.10668.

Dempster, Arthur P, Nan M Laird, and Donald B Rubin. 1977. "Maximum Likelihood from Incomplete Data via the EM Algorithm." *Journal of the Royal Statistical Society: Series B* 39 (1): 1–38.

Deshpande, Ishan, Ziyu Zhang, and Alexander G. Schwing. 2018. "Generative Modeling Using the Sliced Wasserstein Distance." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*.

Dhariwal, Prafulla, and Alex Nichol. 2021. "Diffusion Models Beat GANs on Image Synthesis." *Advances in Neural Information Processing Systems* 34.

DiCiccio, Thomas J., and Bradley Efron. 1996. "Bootstrap Confidence Intervals." *Statistical Science* 11 (3): 189–228. https://doi.org/10.1214/ss/1032280214.

Dinh, Laurent, David Krueger, and Yoshua Bengio. 2014. "NICE: Non-Linear Independent Components Estimation." *ArXiv:1410.8516*. https://arxiv.org/abs/1410.8516.

Dinh, Laurent, Razvan Pascanu, Samy Bengio, and Yoshua Bengio. 2017. "Sharp Minima Can Generalize for Deep Nets." *Proceedings of the 34th International Conference on Machine Learning*, Proceedings of machine learning research, vol. 70: 1019–28. https://proceedings.mlr.press/v70/dinh17b.html.

Dinh, Laurent, Jascha Sohl-Dickstein, and Samy Bengio. 2017. "Density Estimation Using Real NVP." *International Conference on Learning Representations*. https://openreview.net/forum?id=HkpbnH9lx.

Domingos, Pedro, and Michael Pazzani. 1997. "On the Optimality of the Simple Bayesian Classifier Under Zero-One Loss." *Machine Learning* 29: 103–30.

Donsker, Monroe D., and S. R. Srinivasa Varadhan. 1983. "Asymptotic Evaluation of Certain Markov Process Expectations for Large Time. IV." *Communications on Pure and Applied Mathematics* 36 (2): 183–212.

Dormand, John R., and Peter J. Prince. 1980. "A Family of Embedded Runge–Kutta Formulae." *Journal of Computational and Applied Mathematics* 6 (1): 19–26.

Dowson, D. C., and B. V. Landau. 1982. "The Fréchet Distance Between Multivariate Normal Distributions." *Journal of Multivariate Analysis* 12 (3): 450–55.

Drori, Yoel, and Marc Teboulle. 2014. "Performance of First-Order Methods for Smooth Convex Minimization: A Novel Approach." *Mathematical Programming* 145 (1–2): 451–82. https://doi.org/10.1007/s10107-013-0653-0.

Duane, Simon, A. D. Kennedy, Brian J. Pendleton, and Duncan Roweth. 1987. "Hybrid Monte Carlo." *Physics Letters B* 195 (2): 216–22.

Duchi, John, Elad Hazan, and Yoram Singer. 2011. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization." *Journal of Machine Learning Research* 12: 2121–59. https://jmlr.org/papers/v12/duchi11a.html.

Duchi, John, Shai Shalev-Shwartz, Yoram Singer, and Tushar Chandra. 2008. "Efficient Projections onto the ℓ_1-Ball for Learning in High Dimensions." *Proceedings of the 25th International Conference on Machine Learning (ICML)*, 272–79.

Duda, Jarek. 2013. *Asymmetric Numeral Systems: Entropy Coding Combining Speed of Huffman Coding with Compression Rate of Arithmetic Coding*. arXiv:1311.2540. https://arxiv.org/abs/1311.2540.

Dupont, Emilien, Arnaud Doucet, and Yee Whye Teh. 2019. "Augmented Neural ODEs." *Advances in Neural Information Processing Systems* 32.

Dwork, Cynthia, Vitaly Feldman, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Aaron Roth. 2015. "Preserving Statistical Validity in Adaptive Data Analysis." *ACM Symposium on Theory of Computing*, 117–26.

Dziugaite, Gintare Karolina, and Daniel M. Roy. 2017. "Computing Nonvacuous Generalization Bounds for Deep (Stochastic) Neural Networks with Many More Parameters Than Training Data." *Proceedings of the 33rd Conference on Uncertainty in Artificial Intelligence (UAI)*. https://arxiv.org/abs/1703.11008.

Eckart, Carl, and Gale Young. 1936. "The Approximation of One Matrix by Another of Lower Rank." *Psychometrika* 1 (3): 211–18.

Efron, Bradley. 1979. "Bootstrap Methods: Another Look at the Jackknife." *Annals of Statistics* 7 (1): 1–26.

Efron, Bradley. 2011. “Tweedie’s Formula and Selection Bias.” *Journal of the American Statistical Association* 106 (496): 1602–14.

Elhage, Nelson, Tristan Hume, Catherine Olsson, et al. 2022. “Toy Models of Superposition.” *Transformer Circuits Thread*.

Endres, Dominik M., and Johannes E. Schindelin. 2003. “A New Metric for Probability Distributions.” *IEEE Transactions on Information Theory* 49 (7): 1858–60.

Erven, Tim van, and Peter Harremoës. 2014. “Rényi Divergence and Kullback–Leibler Divergence.” *IEEE Transactions on Information Theory* 60 (7): 3797–820.

Feydy, Jean, Thibault Séjourné, François-Xavier Vialard, Shun-ichi Amari, Alain Trouvé, and Gabriel Peyré. 2019. “Interpolating Between Optimal Transport and MMD Using Sinkhorn Divergences.” *Proceedings of the 22nd International Conference on Artificial Intelligence and Statistics*, 2681–90.

Finlay, Chris, Jörn-Henrik Jacobsen, Levon Nurbekyan, and Adam M. Oberman. 2020. “How to Train Your Neural ODE: The World of Jacobian and Kinetic Regularization.” *International Conference on Machine Learning*.

Fisher, Ronald A. 1925. “Theory of Statistical Estimation.” *Mathematical Proceedings of the Cambridge Philosophical Society* 22 (5): 700–725.

Fisher, Ronald A. 1935. *The Design of Experiments*. Oliver; Boyd.

Fisher, Ronald A. 1936. “The Use of Multiple Measurements in Taxonomic Problems.” *Annals of Eugenics* 7 (2): 179–88. https://doi.org/10.1111/j.1469-1809.1936.tb02137.x.

Fokker, Adriaan D. 1914. “Die Mittlere Energie Rotierender Elektrischer Dipole Im Strahlungsfeld.” *Annalen Der Physik* 348 (5): 810–20.

Folland, Gerald B. 1999. *Real Analysis: Modern Techniques and Their Applications*. 2nd ed. Wiley.

Frank, Marguerite, and Philip Wolfe. 1956. “An Algorithm for Quadratic Programming.” *Naval Research Logistics Quarterly* 3 (1–2): 95–110. https://doi.org/10.1002/nav.3800030109.

Frey, Brendan J., and Geoffrey E. Hinton. 1997. “Efficient Stochastic Source Coding and an Application to a Bayesian Network Source Model.” *The Computer Journal* 40 (2/3): 157–65.

Friedman, Jerome, Trevor Hastie, and Robert Tibshirani. 2010. “Regularization Paths for Generalized Linear Models via Coordinate Descent.” *Journal of Statistical Software* 33 (1): 1–22.

Furstenberg, Harry, and Harry Kesten. 1960. “Products of Random Matrices.” *The Annals of Mathematical Statistics* 31 (2): 457–69.

Gavish, Matan, and David L. Donoho. 2014. “The Optimal Hard Threshold for Singular

Values Is 4/√3." *IEEE Transactions on Information Theory* 60 (8): 5040–53.

Geman, Stuart. 1986. "The Spectral Radius of Large Random Matrices." *The Annals of Probability* 14 (4): 1318–28.

Geman, Stuart, Elie Bienenstock, and René Doursat. 1992. "Neural Networks and the Bias/Variance Dilemma." *Neural Computation* 4 (1): 1–58.

Geman, Stuart, and Donald Geman. 1984. "Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 6 (6): 721–41.

Genevay, Aude, Gabriel Peyré, and Marco Cuturi. 2018. "Learning Generative Models with Sinkhorn Divergences." *Proceedings of the 21st International Conference on Artificial Intelligence and Statistics*.

Geng, Zhengyang, Xin-Yu Zhang, Shaojie Bai, Yisen Wang, and Zhouchen Lin. 2021. "On Training Implicit Models." *Advances in Neural Information Processing Systems* 34. https://arxiv.org/abs/2111.05177.

Gershgorin, Semyon A. 1931. "Über Die Abgrenzung Der Eigenwerte Einer Matrix." *Izvestiya Akademii Nauk SSSR, Seriya Matematicheskaya*, no. 6: 749–54.

Geshkovski, Borjan, Cyril Letrouit, Yury Polyanskiy, and Philippe Rigollet. 2023. "A Mathematical Perspective on Transformers." *arXiv Preprint arXiv:2312.10794*.

Geweke, John. 1989. "Bayesian Inference in Econometric Models Using Monte Carlo Integration." *Econometrica* 57 (6): 1317–39. https://doi.org/10.2307/1913710.

Ghadimi, Saeed, and Guanghui Lan. 2013. "Stochastic First- and Zeroth-Order Methods for Nonconvex Stochastic Programming." *SIAM Journal on Optimization* 23 (4): 2341–68. https://arxiv.org/abs/1309.5549.

Gibbs, Josiah Willard. 1902. *Elementary Principles of Statistical Mechanics*. Scribner's.

Giles, Michael B. 2008. "Multilevel Monte Carlo Path Simulation." *Operations Research* 56 (3): 607–17.

Giles, Mike B. 2008. *An Extended Collection of Matrix Derivative Results for Forward and Reverse Mode Algorithmic Differentiation*. NA-08/01. Oxford University Computing Laboratory.

Ginibre, Jean. 1965. "Statistical Ensembles of Complex, Quaternion, and Real Matrices." *Journal of Mathematical Physics* 6 (3): 440–49. https://doi.org/10.1063/1.1704292.

Gnedenko, B. V., and A. N. Kolmogorov. 1954. *Limit Distributions for Sums of Independent Random Variables*. Addison-Wesley.

Gneiting, Tilmann, Fadoua Balabdaoui, and Adrian E. Raftery. 2007. "Probabilistic Forecasts, Calibration and Sharpness." *Journal of the Royal Statistical Society Series B: Statistical Methodology* 69 (2): 243–68. https://doi.org/10.1111/j.1467-9868.2007.00587.x.

Gneiting, Tilmann, and Adrian E. Raftery. 2007. "Strictly Proper Scoring Rules, Prediction, and Estimation." *Journal of the American Statistical Association* 102 (477): 359–78. https://doi.org/10.1198/016214506000001437.

Goldberg, David. 1991. "What Every Computer Scientist Should Know about Floating-Point Arithmetic." *ACM Computing Surveys* 23 (1): 5–48.

Goldfeld, Ziv, Ewout van den Berg, Kristjan Greenewald, et al. 2019. "Estimating Information Flow in Deep Neural Networks." *Proceedings of the 36th International Conference on Machine Learning*, Proceedings of machine learning research, vol. 97: 2299–308.

Golub, Gene H, and Charles F Van Loan. 1996. *Matrix Computations*. Johns Hopkins University Press. https://jhupbooks.press.jhu.edu/title/matrix-computations.

Goodfellow, Ian, Jean Pouget-Abadie, Mehdi Mirza, et al. 2014. "Generative Adversarial Nets." *Advances in Neural Information Processing Systems*, 2672–80. https://doi.org/10.5555/2969033.2969125.

Goyal, Priya, Piotr Dollár, Ross Girshick, et al. 2017. "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour." *ArXiv:1706.02677*. https://arxiv.org/abs/1706.02677.

Grathwohl, Will, Ricky T. Q. Chen, Jesse Bettencourt, Ilya Sutskever, and David Duvenaud. 2019. "FFJORD: Free-Form Continuous Dynamics for Scalable Reversible Generative Models." *International Conference on Learning Representations*.

Gretton, Arthur, Karsten M. Borgwardt, Malte J. Rasch, Bernhard Schölkopf, and Alexander Smola. 2012. "A Kernel Two-Sample Test." *Journal of Machine Learning Research* 13: 723–73.

Griewank, Andreas. 1992. "Achieving Logarithmic Growth of Temporal and Spatial Complexity in Reverse Automatic Differentiation." *Optimization Methods and Software* 1 (1): 35–54.

Griewank, Andreas, and Andrea Walther. 2008. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. 2nd ed. Society for Industrial; Applied Mathematics (SIAM). https://doi.org/10.1137/1.9780898717761.

Grobman, David M. 1959. "Homeomorphism of Systems of Differential Equations." *Doklady Akademii Nauk SSSR* 128: 880–81.

Grünwald, Peter D. 2007. *The Minimum Description Length Principle*. MIT Press. https://doi.org/10.7551/mitpress/4643.001.0001.

Gulrajani, Ishaan, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron Courville. 2017. "Improved Training of Wasserstein GANs." *Advances in Neural Information Processing Systems* 30.

Gumbel, Emil J. 1954. *Statistical Theory of Extreme Values and Some Practical Applications*. U.S. National Bureau of Standards.

Guo, Chuan, Geoff Pleiss, Yu Sun, and Kilian Q. Weinberger. 2017. "On Calibration of Modern Neural Networks." *International Conference on Machine Learning*, 1321–30. https://arxiv.org/abs/1706.04599.

Gupta, Suyog, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. "Deep Learning with Limited Numerical Precision." *International Conference on Machine Learning*, 1737–46.

Gupta, Vineet, Tomer Koren, and Yoram Singer. 2018. "Shampoo: Preconditioned Stochastic Tensor Optimization." *Proceedings of the 35th International Conference on Machine Learning (ICML)*, 1842–50.

Hairer, Ernst, Christian Lubich, and Gerhard Wanner. 2006. *Geometric Numerical Integration: Structure-Preserving Algorithms for Ordinary Differential Equations*. 2nd ed. Vol. 31. Springer Series in Computational Mathematics. Springer.

Hairer, Ernst, Syvert P. Nørsett, and Gerhard Wanner. 1993. *Solving Ordinary Differential Equations I: Nonstiff Problems*. 2nd ed. Springer.

Halko, Nathan, Per-Gunnar Martinsson, and Joel A. Tropp. 2011. "Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions." *SIAM Review* 53 (2): 217–88.

Hall, Peter. 1992. *The Bootstrap and Edgeworth Expansion*. Springer Series in Statistics. Springer. https://doi.org/10.1007/978-1-4612-4384-7.

Hanin, Boris, and Mihai Nica. 2020. "Products of Many Large Random Matrices and Gradients in Deep Neural Networks." *Communications in Mathematical Physics* 376 (1): 287–322.

Hardt, Moritz, Ben Recht, and Yoram Singer. 2016. "Train Faster, Generalize Better: Stability of Stochastic Gradient Descent." *Proceedings of the 33rd International Conference on Machine Learning*, Proceedings of machine learning research, vol. 48: 1225–34. https://proceedings.mlr.press/v48/hardt16.html.

Hartman, Philip. 1960. "A Lemma in the Theory of Structural Stability of Differential Equations." *Proceedings of the American Mathematical Society* 11 (4): 610–20.

Hastings, W. Keith. 1970. "Monte Carlo Sampling Methods Using Markov Chains and Their Applications." *Biometrika* 57 (1): 97–109.

Haveliwala, Taher H., and Sepandar D. Kamvar. 2003. *The Second Eigenvalue of the Google Matrix*. Stanford University. http://ilpubs.stanford.edu:8090/582/.

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification." *Proceedings of the IEEE International Conference on Computer Vision*, 1026–34. https://doi.org/10.1109/iccv.2015.123.

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016a. "Deep Residual Learning for Image Recognition." *Proceedings of the IEEE Conference on Computer Vision and Pattern*

Recognition, 770–78. https://doi.org/10.1109/cvpr.2016.90.

Held, Michael, Philip Wolfe, and Harlan P. Crowder. 1974. "Validation of Subgradient Optimization." *Mathematical Programming* 6: 62–88.

Hertrich, Johannes, Antonin Chambolle, and Julie Delon. 2025. "On the Relation Between Rectified Flows and Optimal Transport." *arXiv Preprint arXiv:2505.19712*.

Hestenes, Magnus R., and Eduard Stiefel. 1952. "Methods of Conjugate Gradients for Solving Linear Systems." *Journal of Research of the National Bureau of Standards* 49 (6): 409–36.

Higham, Nicholas J. 2002. *Accuracy and Stability of Numerical Algorithms*. 2nd ed. SIAM.

Higham, Nicholas J., and Theo Mary. 2019. "A New Approach to Probabilistic Rounding Error Analysis." *SIAM Journal on Scientific Computing* 41 (5): A2815–35. https://doi.org/10.1137/18M1226312.

Hill, Bruce M. 1975. "A Simple General Approach to Inference about the Tail of a Distribution." *The Annals of Statistics* 3 (5): 1163–74. https://doi.org/10.1214/aos/1176343247.

Hinton, Geoffrey E., and Drew van Camp. 1993. "Keeping the Neural Networks Simple by Minimizing the Description Length of the Weights." *Proceedings of the Sixth Annual Conference on Computational Learning Theory*, 5–13.

Hinton, Geoffrey, Oriol Vinyals, and Jeff Dean. 2015. "Distilling the Knowledge in a Neural Network." *arXiv Preprint arXiv:1503.02531*.

Ho, Jonathan, Ajay Jain, and Pieter Abbeel. 2020. "Denoising Diffusion Probabilistic Models." *Advances in Neural Information Processing Systems* 33: 6840–51. https://arxiv.org/abs/2006.11239.

Ho, Jonathan, and Tim Salimans. 2022. "Classifier-Free Diffusion Guidance." *arXiv Preprint arXiv:2207.12598*.

Hochbruck, Marlis, and Alexander Ostermann. 2010. "Exponential Integrators." *Acta Numerica* 19: 209–86.

Hoeffding, Wassily. 1963. "Probability Inequalities for Sums of Bounded Random Variables." *Journal of the American Statistical Association* 58 (301): 13–30.

Hoffman, Matthew D., and Andrew Gelman. 2014. "The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo." *Journal of Machine Learning Research* 15: 1593–623. https://jmlr.org/papers/v15/hoffman14a.html.

Hoffmann, Jordan, Sebastian Borgeaud, Arthur Mensch, et al. 2022. "Training Compute-Optimal Large Language Models." *Advances in Neural Information Processing Systems* 35. https://arxiv.org/abs/2203.15556.

Horn, Roger A., and Charles R. Johnson. 2012. *Matrix Analysis*. 2nd ed. Cambridge University Press.

Howard, Steven R., Aaditya Ramdas, Jon McAuliffe, and Jasjeet Sekhon. 2021. "Time-Uniform, Nonparametric, Nonasymptotic Confidence Sequences." *The Annals of Statistics* 49 (2): 1055–80. https://doi.org/10.1214/20-AOS1991.

Hu, Edward J., Yelong Shen, Phillip Wallis, et al. 2021. "LoRA: Low-Rank Adaptation of Large Language Models." *arXiv Preprint arXiv:2106.09685*.

Hu, Shengding, Yuge Tu, Xu Han, et al. 2024. "MiniCPM: Unveiling the Potential of Small Language Models with Scalable Training Strategies." *ArXiv:2404.06395*. https://arxiv.org/abs/2404.06395.

Huffman, David A. 1952. "A Method for the Construction of Minimum-Redundancy Codes." *Proceedings of the IRE* 40 (9): 1098–101. https://doi.org/10.1109/JRPROC.1952.273898.

Hutchinson, Michael F. 1989. "A Stochastic Estimator of the Trace of the Influence Matrix for Laplacian Smoothing Splines." *Communications in Statistics—Simulation and Computation* 18 (3): 1059–76.

Hyvärinen, Aapo. 2005. "Estimation of Non-Normalized Statistical Models by Score Matching." *Journal of Machine Learning Research* 6: 695–709.

IEEE. 2019. *IEEE Standard for Floating-Point Arithmetic*. IEEE Std 754-2019.

Itô, Kiyosi. 1944. "Stochastic Integral." *Proceedings of the Imperial Academy* 20 (8): 519–24.

Jaggi, Martin. 2013. "Revisiting Frank-Wolfe: Projection-Free Sparse Convex Optimization." *Proceedings of the 30th International Conference on Machine Learning*, Proceedings of machine learning research, vol. 28: 427–35.

James, W., and Charles Stein. 1961. "Estimation with Quadratic Loss." *Proceedings of the Fourth Berkeley Symposium on Mathematical Statistics and Probability* 1: 361–79.

Jang, Eric, Shixiang Gu, and Ben Poole. 2017. "Categorical Reparameterization with Gumbel-Softmax." *International Conference on Learning Representations*.

Jaynes, Edwin T. 1957. "Information Theory and Statistical Mechanics." *Physical Review* 106 (4): 620–30. https://doi.org/10.1103/PhysRev.106.620.

Jaynes, Edwin T. 1968. "Prior Probabilities." *IEEE Transactions on Systems Science and Cybernetics* 4 (3): 227–41. https://doi.org/10.1109/TSSC.1968.300117.

Jelinek, Frederick, Robert L. Mercer, Lalit R. Bahl, and James K. Baker. 1977. "Perplexity—a Measure of the Difficulty of Speech Recognition Tasks." *Journal of the Acoustical Society of America* 62 (S1): S63.

Johari, Ramesh, Pete Koomen, Leonid Pekelis, and David Walsh. 2017. "Peeking at A/B Tests: Why It Matters, and What to Do about It." *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 1517–25. https://doi.org/10.1145/3097983.3097992.

Johnson, Rie, and Tong Zhang. 2013. "Accelerating Stochastic Gradient Descent Using Predictive Variance Reduction." *Advances in Neural Information Processing Systems* 26: 315–23.

Johnson, William B., and Joram Lindenstrauss. 1984. "Extensions of Lipschitz Mappings into a Hilbert Space." *Contemporary Mathematics* 26: 189–206.

Jordan, Keller, Yuchen Jin, Vlado Boza, et al. 2024. *Muon: An Optimizer for Hidden Layers in Neural Networks*. Https://kellerjordan.github.io/posts/muon/.

Jordan, Richard, David Kinderlehrer, and Felix Otto. 1998. "The Variational Formulation of the Fokker–Planck Equation." *SIAM Journal on Mathematical Analysis* 29 (1): 1–17.

Kahan, William. 1965. "Further Remarks on Reducing Truncation Errors." *Communications of the ACM* 8 (1): 40.

Kaplan, Jared, Sam McCandlish, Tom Henighan, et al. 2020. "Scaling Laws for Neural Language Models." *ArXiv:2001.08361*. https://arxiv.org/abs/2001.08361.

Karimi, Hamed, Julie Nutini, and Mark Schmidt. 2016. "Linear Convergence of Gradient and Proximal-Gradient Methods Under the Polyak–Łojasiewicz Condition." *Joint European Conference on Machine Learning and Knowledge Discovery in Databases (ECML PKDD)*, 795–811.

Karras, Tero, Miika Aittala, Timo Aila, and Samuli Laine. 2022. "Elucidating the Design Space of Diffusion-Based Generative Models." *Advances in Neural Information Processing Systems* 35.

Karush, William. 1939. "Minima of Functions of Several Variables with Inequalities as Side Constraints." Master's thesis, Department of Mathematics, University of Chicago.

Kidger, Patrick. 2022. "On Neural Differential Equations." *arXiv Preprint arXiv:2202.02435*.

Kidger, Patrick, James Morrill, James Foster, and Terry Lyons. 2020. "Neural Controlled Differential Equations for Irregular Time Series." *Advances in Neural Information Processing Systems* 33: 6696–707.

Kingma, Diederik P, and Jimmy Ba. 2015. "Adam: A Method for Stochastic Optimization." *International Conference on Learning Representations*. https://arxiv.org/abs/1412.6980.

Kingma, Diederik P., Tim Salimans, Ben Poole, and Jonathan Ho. 2021. "Variational Diffusion Models." *Advances in Neural Information Processing Systems* 34.

Kingma, Diederik P., and Max Welling. 2014. "Auto-Encoding Variational Bayes." *International Conference on Learning Representations (ICLR)*. https://arxiv.org/abs/1312.6114.

Kloeden, Peter E., and Eckhard Platen. 1992. *Numerical Solution of Stochastic Differential Equations*. Springer.

Kolda, Tamara G., and Brett W. Bader. 2009. "Tensor Decompositions and Applications."

SIAM Review 51 (3): 455–500.

Koller, Daphne, and Nir Friedman. 2009. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press. https://doi.org/10.7551/mitpress/7432.001.0001.

Kolmogorov, Andrey N. 1931. "Über Die Analytischen Methoden in Der Wahrscheinlichkeitsrechnung." *Mathematische Annalen* 104: 415–58.

Koopman, Bernard O. 1936. "On Distributions Admitting a Sufficient Statistic." *Transactions of the American Mathematical Society* 39 (3): 399–409.

Kraskov, Alexander, Harald Stögbauer, and Peter Grassberger. 2004. "Estimating Mutual Information." *Physical Review E* 69 (6): 066138.

Kuhn, Harold W., and Albert W. Tucker. 1951. "Nonlinear Programming." *Proceedings of the Second Berkeley Symposium on Mathematical Statistics and Probability*, 481–92.

Kullback, Solomon, and Richard A. Leibler. 1951. "On Information and Sufficiency." *Annals of Mathematical Statistics* 22 (1): 79–86.

Kynkäänniemi, Tuomas, Miika Aittala, Tero Karras, Samuli Laine, Timo Aila, and Jaakko Lehtinen. 2024. "Applying Guidance in a Limited Interval Improves Sample and Distribution Quality in Diffusion Models." *Advances in Neural Information Processing Systems* 37. https://arxiv.org/abs/2404.07724.

Lakshminarayanan, Balaji, Alexander Pritzel, and Charles Blundell. 2017. "Simple and Scalable Predictive Uncertainty Estimation Using Deep Ensembles." *Advances in Neural Information Processing Systems* 30.

Laplace, Pierre-Simon. 1814. *Essai Philosophique Sur Les Probabilités*. Courcier.

Läuchli, Peter. 1961. "Jordan-Elimination Und Ausgleichung Nach Kleinsten Quadraten." *Numerische Mathematik* 3 (1): 226–40.

LeCun, Yann, Leon Bottou, G Orr, and Klaus-Robert Muller. 1998. "Efficient Backprop." In *Neural Networks: Tricks of the Trade*. Springer. https://doi.org/10.1007/3-540-49430-8_2.

LeCun, Yann, Sumit Chopra, Raia Hadsell, Marc'Aurelio Ranzato, and Fu Jie Huang. 2006. "A Tutorial on Energy-Based Learning." In *Predicting Structured Data*. MIT Press.

Lessard, Laurent, Benjamin Recht, and Andrew Packard. 2016. "Analysis and Design of Optimization Algorithms via Integral Quadratic Constraints." *SIAM Journal on Optimization* 26 (1): 57–95.

Levin, David A., and Yuval Peres. 2017. *Markov Chains and Mixing Times*. 2nd ed. American Mathematical Society.

Levy, Omer, and Yoav Goldberg. 2014. "Neural Word Embedding as Implicit Matrix Factorization." *Advances in Neural Information Processing Systems* 27.

Li, Ming, and Paul Vitányi. 2019. *An Introduction to Kolmogorov Complexity and Its Applications*. 4th ed. Texts in Computer Science. Springer. https://doi.org/10.1007/978-3-030-11298-1.

Li, Qianxiao, Cheng Tai, and Weinan E. 2017. "Stochastic Modified Equations and Adaptive Stochastic Gradient Algorithms." *Proceedings of the 34th International Conference on Machine Learning*, Proceedings of machine learning research, vol. 70: 2101–10.

Li, Xuechen, Ting-Kam Leonard Wong, Ricky T. Q. Chen, and David Duvenaud. 2020. "Scalable Gradients for Stochastic Differential Equations." *Proceedings of the 23rd International Conference on Artificial Intelligence and Statistics*, Proceedings of machine learning research, vol. 108: 3870–82.

Li, Yujia, Kevin Swersky, and Richard Zemel. 2015. "Generative Moment Matching Networks." *Proceedings of the 32nd International Conference on Machine Learning*, 1718–27.

Lin, Jianhua. 1991. "Divergence Measures Based on the Shannon Entropy." *IEEE Transactions on Information Theory* 37 (1): 145–51.

Linnainmaa, Seppo. 1970. "The Representation of the Cumulative Rounding Error of an Algorithm as a Taylor Expansion of the Local Rounding Errors." Master's thesis, University of Helsinki.

Lipman, Yaron, Ricky T. Q. Chen, Heli Ben-Hamu, Maximilian Nickel, and Matt Le. 2023. "Flow Matching for Generative Modeling." *International Conference on Learning Representations*.

Liu, Chaoyue, Libin Zhu, and Mikhail Belkin. 2022. "Loss Landscapes and Optimization in over-Parameterized Non-Linear Systems and Neural Networks." *Applied and Computational Harmonic Analysis* 59: 85–116.

Liu, Dong C, and Jorge Nocedal. 1989. "On the Limited Memory BFGS Method for Large Scale Optimization." *Mathematical Programming* 45 (1): 503–28. https://doi.org/10.1007/bf01589116.

Liu, Qiang. 2022. "Rectified Flow: A Marginal Preserving Approach to Optimal Transport." *arXiv Preprint arXiv:2209.14577*.

Liu, Qiang, Jason D. Lee, and Michael I. Jordan. 2016. "A Kernelized Stein Discrepancy for Goodness-of-Fit Tests." *Proceedings of the 33rd International Conference on Machine Learning*, 276–84.

Liu, Qiang, and Dilin Wang. 2016. "Stein Variational Gradient Descent: A General Purpose Bayesian Inference Algorithm." *Advances in Neural Information Processing Systems* 29.

Liu, Shih-Yang, Chien-Yi Wang, Hongxu Yin, et al. 2024. "DoRA: Weight-Decomposed Low-Rank Adaptation." *International Conference on Machine Learning*.

Liu, Xingchao, Chengyue Gong, and Qiang Liu. 2023. "Flow Straight and Fast: Learning to Generate and Transfer Data with Rectified Flow." *International Conference on Learning*

Representations.

Łojasiewicz, Stanisław. 1963. "Une Propriété Topologique Des Sous-Ensembles Analytiques réels." In *Les Équations Aux dérivées Partielles*. Éditions du CNRS.

Loshchilov, Ilya, and Frank Hutter. 2016. "SGDR: Stochastic Gradient Descent with Warm Restarts." *ArXiv:1608.03983*. https://arxiv.org/abs/1608.03983.

Loshchilov, Ilya, and Frank Hutter. 2019. "Decoupled Weight Decay Regularization." *International Conference on Learning Representations*. https://arxiv.org/abs/1711.05101.

Lu, Cheng, Kaiwen Zheng, Fan Bao, Jianfei Chen, Chongxuan Li, and Jun Zhu. 2022. "Maximum Likelihood Training for Score-Based Diffusion ODEs by High-Order Denoising Score Matching." *International Conference on Machine Learning*.

Lu, Cheng, Yuhao Zhou, Fan Bao, Jianfei Chen, Chongxuan Li, and Jun Zhu. 2022. "DPM-Solver: A Fast ODE Solver for Diffusion Probabilistic Model Sampling in Around 10 Steps." *Advances in Neural Information Processing Systems* 35.

Ma, Siyuan, Raef Bassily, and Mikhail Belkin. 2018. "The Power of Interpolation: Understanding the Effectiveness of SGD in Modern over-Parametrized Learning." *Proceedings of the 35th International Conference on Machine Learning*, Proceedings of machine learning research, vol. 80: 3325–34.

MacKay, David J. C. 1992. "The Evidence Framework Applied to Classification Networks." *Neural Computation* 4 (5): 720–36. https://doi.org/10.1162/neco.1992.4.5.720.

Maddison, Chris J., Andriy Mnih, and Yee Whye Teh. 2017. "The Concrete Distribution: A Continuous Relaxation of Discrete Random Variables." *International Conference on Learning Representations*.

Madry, Aleksander, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. 2018. "Towards Deep Learning Models Resistant to Adversarial Attacks." *International Conference on Learning Representations*. https://arxiv.org/abs/1706.06083.

Mahoney, Michael W., and Petros Drineas. 2009. "CUR Matrix Decompositions for Improved Data Analysis." *Proceedings of the National Academy of Sciences* 106 (3): 697–702.

Mandt, Stephan, Matthew D. Hoffman, and David M. Blei. 2017. "Stochastic Gradient Descent as Approximate Bayesian Inference." *Journal of Machine Learning Research* 18 (134): 1–35.

Manning, Christopher D., Prabhakar Raghavan, and Hinrich Schütze. 2008. *Introduction to Information Retrieval*. Cambridge University Press. https://doi.org/10.1017/CBO9780511809071.

Marchenko, Vladimir A., and Leonid A. Pastur. 1967. "Distribution of Eigenvalues for Some Sets of Random Matrices." *Mathematics of the USSR-Sbornik* 1 (4): 457–83.

Maron, Melvin E. 1961. "Automatic Indexing: An Experimental Inquiry." *Journal of the ACM*

8 (3): 404–17.

Martens, James. 2010. "Deep Learning via Hessian-Free Optimization." *Proceedings of the 27th International Conference on Machine Learning*, 735–42.

Martens, James. 2020. "New Insights and Perspectives on the Natural Gradient Method." *Journal of Machine Learning Research* 21 (146).

Martens, James, and Roger Grosse. 2015. "Optimizing Neural Networks with Kronecker-Factored Approximate Curvature." *Proceedings of the 32nd International Conference on Machine Learning (ICML)*, 2408–17.

Martin, Charles H., and Michael W. Mahoney. 2021. "Implicit Self-Regularization in Deep Neural Networks: Evidence from Random Matrix Theory and Implications for Learning." *Journal of Machine Learning Research* 22 (165): 1–73.

Martins, André F. T., and Ramón F. Astudillo. 2016. "From Softmax to Sparsemax: A Sparse Model of Attention and Multi-Label Classification." *Proceedings of the 33rd International Conference on Machine Learning (ICML)*, 1614–23.

Maruyama, Gisiro. 1955. "Continuous Markov Processes and Stochastic Equations." *Rendiconti Del Circolo Matematico Di Palermo* 4: 48–90.

Mazumder, Rahul, Trevor Hastie, and Robert Tibshirani. 2010. "Spectral Regularization Algorithms for Learning Large Incomplete Matrices." *Journal of Machine Learning Research* 11 (80): 2287–322.

McAllester, David A. 1999. "Some PAC-Bayesian Theorems." *Machine Learning* 37 (3): 355–63. https://doi.org/10.1023/A:1007618624809.

McAllester, David, and Karl Stratos. 2020. "Formal Limitations on the Measurement of Mutual Information." *International Conference on Artificial Intelligence and Statistics*, 875–84.

McCandlish, Sam, Jared Kaplan, Dario Amodei, and OpenAI Dota Team. 2018. "An Empirical Model of Large-Batch Training." *arXiv Preprint arXiv:1812.06162*.

McCann, Robert J. 1997. "A Convexity Principle for Interacting Gases." *Advances in Mathematics* 128 (1): 153–79.

McDiarmid, Colin. 1989. "On the Method of Bounded Differences." In *Surveys in Combinatorics*. Cambridge University Press.

McFadden, Daniel. 1974. "Conditional Logit Analysis of Qualitative Choice Behavior." In *Frontiers in Econometrics*, edited by Paul Zarembka. Academic Press.

Meng, Fanxu, Zhaohui Wang, and Muhan Zhang. 2024. "PiSSA: Principal Singular Values and Singular Vectors Adaptation of Large Language Models." *Advances in Neural Information Processing Systems*.

Metropolis, Nicholas, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller,

and Edward Teller. 1953. “Equation of State Calculations by Fast Computing Machines.” *The Journal of Chemical Physics* 21 (6): 1087–92.

Metropolis, Nicholas, and Stanislaw Ulam. 1949. “The Monte Carlo Method.” *Journal of the American Statistical Association* 44 (247): 335–41.

Miller, Evan. 2024. “Adding Error Bars to Evals: A Statistical Approach to Language Model Evaluations.” *arXiv Preprint arXiv:2411.00640*.

Milstein, Grigori N. 1975. “Approximate Integration of Stochastic Differential Equations.” *Theory of Probability and Its Applications* 19 (3): 557–62.

Mironov, Ilya. 2017. “Rényi Differential Privacy.” *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*, 263–75.

Mirsky, Leon. 1960. “Symmetric Gauge Functions and Unitarily Invariant Norms.” *The Quarterly Journal of Mathematics* 11 (1): 50–59.

Miyato, Takeru, Toshiki Kataoka, Masanori Koyama, and Yuichi Yoshida. 2018. “Spectral Normalization for Generative Adversarial Networks.” *International Conference on Learning Representations*.

Mohamed, Shakir, Mihaela Rosca, Michael Figurnov, and Andriy Mnih. 2020. “Monte Carlo Gradient Estimation in Machine Learning.” *Journal of Machine Learning Research* 21 (132): 1–62.

Moler, Cleve B. 1967. “Iterative Refinement in Floating Point.” *Journal of the ACM* 14 (2): 316–21.

Müller, Alfred. 1997. “Integral Probability Metrics and Their Generating Classes of Functions.” *Advances in Applied Probability* 29 (2): 429–43.

Müller, Rafael, Simon Kornblith, and Geoffrey Hinton. 2019. “When Does Label Smoothing Help?” *Advances in Neural Information Processing Systems* 32. https://arxiv.org/abs/1906.02629.

Naumann, Uwe. 2008. “Optimal Jacobian Accumulation Is NP-Complete.” *Mathematical Programming* 112 (2): 427–41.

Neal, Radford M. 2011. “MCMC Using Hamiltonian Dynamics.” In *Handbook of Markov Chain Monte Carlo*. Chapman; Hall/CRC. https://doi.org/10.1201/b10905-6.

Nemirovski, Arkadi. 2004. “Prox-Method with Rate of Convergence $O(1/t)$ for Variational Inequalities with Lipschitz Continuous Monotone Operators and Smooth Convex-Concave Saddle Point Problems.” *SIAM Journal on Optimization* 15 (1): 229–51. https://doi.org/10.1137/S1052623403425629.

Nemirovski, Arkadi, and David Yudin. 1983. *Problem Complexity and Method Efficiency in Optimization*. Wiley.

Nesterov, Yu. 2018. *Lectures on Convex Optimization*. Springer. https://doi.org/10.1007/978-3-319-91578-4.

Nesterov, Yurii. 1983. "A Method of Solving a Convex Programming Problem with Convergence Rate $O(1/k^2)$." *Soviet Mathematics Doklady* 27 (2): 372–76.

Nesterov, Yurii. 2005. "Smooth Minimization of Non-Smooth Functions." *Mathematical Programming* 103 (1): 127–52. https://doi.org/10.1007/s10107-004-0552-5.

Nesterov, Yurii. 2012. "Efficiency of Coordinate Descent Methods on Huge-Scale Optimization Problems." *SIAM Journal on Optimization* 22 (2): 341–62.

Neyman, Jerzy. 1937. "Outline of a Theory of Statistical Estimation Based on the Classical Theory of Probability." *Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences* 236 (767): 333–80. https://doi.org/10.2307/jj.8501421.24.

Ng, Andrew Y., and Michael I. Jordan. 2002. "On Discriminative Vs. Generative Classifiers: A Comparison of Logistic Regression and Naive Bayes." *Advances in Neural Information Processing Systems* 14: 841–48.

Nguyen, XuanLong, Martin J. Wainwright, and Michael I. Jordan. 2010. "Estimating Divergence Functionals and the Likelihood Ratio by Convex Risk Minimization." *IEEE Transactions on Information Theory* 56 (11): 5847–61.

Nigam, Kamal, Andrew Kachites McCallum, Sebastian Thrun, and Tom Mitchell. 2000. "Text Classification from Labeled and Unlabeled Documents Using EM." *Machine Learning* 39 (2–3): 103–34. https://doi.org/10.1023/A:1007692713085.

Nocedal, Jorge, and Stephen J. Wright. 2006. *Numerical Optimization*. 2nd ed. Springer.

Nowozin, Sebastian, Botond Cseke, and Ryota Tomioka. 2016. "F-GAN: Training Generative Neural Samplers Using Variational Divergence Minimization." *Advances in Neural Information Processing Systems* 29.

Øksendal, Bernt. 2003. *Stochastic Differential Equations: An Introduction with Applications*. 6th ed. Springer.

Oord, Aaron van den, Yazhe Li, and Oriol Vinyals. 2018. "Representation Learning with Contrastive Predictive Coding." *arXiv Preprint arXiv:1807.03748*.

Oseledets, Ivan V. 2011. "Tensor-Train Decomposition." *SIAM Journal on Scientific Computing* 33 (5): 2295–317.

Owen, Art B. 2013. *Monte Carlo Theory, Methods and Examples*.

Paley, Raymond E. A. C., Norbert Wiener, and Antoni Zygmund. 1933. "Notes on Random Functions." *Mathematische Zeitschrift* 37: 647–68.

Papamakarios, George, Eric Nalisnick, Danilo Jimenez Rezende, Shakir Mohamed, and Bal-

aji Lakshminarayanan. 2021. “Normalizing Flows for Probabilistic Modeling and Inference.” *Journal of Machine Learning Research* 22 (57): 1–64.

Parikh, Neal, and Stephen Boyd. 2014. “Proximal Algorithms.” *Foundations and Trends in Optimization* 1 (3): 127–239. https://doi.org/10.1561/2400000003.

Pascanu, Razvan, Tomas Mikolov, and Yoshua Bengio. 2013. “On the Difficulty of Training Recurrent Neural Networks.” *International Conference on Machine Learning*, 1310–18.

Pearlmutter, Barak A. 1994. “Fast Exact Multiplication by the Hessian.” *Neural Computation* 6 (1): 147–60. https://doi.org/10.1162/neco.1994.6.1.147.

Pennington, Jeffrey, Samuel Schoenholz, and Surya Ganguli. 2017. “Resurrecting the Sigmoid in Deep Learning Through Dynamical Isometry: Theory and Practice.” *Advances in Neural Information Processing Systems*, 4785–95. https://proceedings.neurips.cc/paper/2017/hash/a9fc2d3b0c721b5b3f0b3f11bb24a75c-Abstract.html.

Peyré, Gabriel, and Marco Cuturi. 2019. “Computational Optimal Transport.” *Foundations and Trends in Machine Learning* 11 (5–6): 355–607.

Pinsker, Mark S. 1964. *Information and Information Stability of Random Variables and Processes*. Holden-Day.

Pitman, Edwin J. G. 1936. “Sufficient Statistics and Intrinsic Accuracy.” *Mathematical Proceedings of the Cambridge Philosophical Society* 32 (4): 567–79.

Planck, Max. 1917. “Über Einen Satz Der Statistischen Dynamik Und Seine Erweiterung in Der Quantentheorie.” *Sitzungsberichte Der Preussischen Akademie Der Wissenschaften*, 324–41.

Platt, John C. 1999. “Probabilistic Outputs for Support Vector Machines and Comparisons to Regularized Likelihood Methods.” In *Advances in Large Margin Classifiers*, edited by Alexander J. Smola, Peter Bartlett, Bernhard Schölkopf, and Dale Schuurmans. MIT Press.

Polyak, Boris T. 1964. “Some Methods of Speeding up the Convergence of Iteration Methods.” *USSR Computational Mathematics and Mathematical Physics* 4 (5): 1–17. https://doi.org/10.1016/0041-5553(64)90137-5.

Polyak, Boris T. 1963. “Gradient Methods for the Minimisation of Functionals.” *USSR Computational Mathematics and Mathematical Physics* 3 (4): 864–78.

Polyak, Boris T., and Anatoli B. Juditsky. 1992. “Acceleration of Stochastic Approximation by Averaging.” *SIAM Journal on Control and Optimization* 30 (4): 838–55.

Pontryagin, Lev S., Vladimir G. Boltyanskii, Revaz V. Gamkrelidze, and Evgenii F. Mishchenko. 1962. *The Mathematical Theory of Optimal Processes*. Interscience.

Pooladian, Aram-Alexandre, Heli Ben-Hamu, Carles Domingo-Enrich, Brandon Amos, Yaron Lipman, and Ricky T. Q. Chen. 2023. “Multisample Flow Matching: Straighten-

ing Flows with Minibatch Couplings." *Proceedings of the 40th International Conference on Machine Learning*.

Poole, Ben, Sherjil Ozair, Aaron van den Oord, Alexander A. Alemi, and George Tucker. 2019. "On Variational Bounds of Mutual Information." *Proceedings of the 36th International Conference on Machine Learning*, 5171–80.

Rademacher, Hans. 1919. "Über Partielle Und Totale Differenzierbarkeit von Funktionen Mehrerer Variabeln Und über Die Transformation Der Doppelintegrale." *Mathematische Annalen* 79 (4): 340–59.

Radford, Alec, Jong Wook Kim, Chris Hallacy, et al. 2021. "Learning Transferable Visual Models from Natural Language Supervision." *International Conference on Machine Learning*, 8748–63. https://proceedings.mlr.press/v139/radford21a.html.

Radhakrishna Rao, C. 1945. "Information and Accuracy Attainable in the Estimation of Statistical Parameters." *Bulletin of the Calcutta Mathematical Society* 37 (3): 81–91. https://doi.org/10.1007/BF02908227.

Rahimi, Ali, and Benjamin Recht. 2007. "Random Features for Large-Scale Kernel Machines." *Advances in Neural Information Processing Systems* 20.

Ramdas, Aaditya, Sashank J. Reddi, Barnabás Póczos, Aarti Singh, and Larry Wasserman. 2015. "On the Decreasing Power of Kernel and Distance Based Nonparametric Hypothesis Tests in High Dimensions." *Proceedings of the AAAI Conference on Artificial Intelligence* 29.

Reddi, Sashank J, Satyen Kale, and Sanjiv Kumar. 2018. "On the Convergence of Adam and Beyond." *International Conference on Learning Representations*. https://arxiv.org/abs/1904.09237.

Rényi, Alfréd. 1961. "On Measures of Entropy and Information." *Proceedings of the Fourth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Contributions to the Theory of Statistics*, 547–61.

Rezende, Danilo Jimenez, Shakir Mohamed, and Daan Wierstra. 2014. "Stochastic Backpropagation and Approximate Inference in Deep Generative Models." *International Conference on Machine Learning*, 1278–86. https://proceedings.mlr.press/v32/rezende14.html.

Rezende, Danilo, and Shakir Mohamed. 2015. "Variational Inference with Normalizing Flows." *International Conference on Machine Learning*, 1530–38. https://proceedings.mlr.press/v37/rezende15.html.

Richardson, Tom, and Rüdiger Urbanke. 2008. *Modern Coding Theory*. Cambridge University Press. https://doi.org/10.1017/CBO9780511791338.

Risken, Hannes. 1996. *The Fokker–Planck Equation: Methods of Solution and Applications*. 2nd ed. Springer.

Rissanen, Jorma J. 1976. "Generalized Kraft Inequality and Arithmetic Coding." *IBM Journal of Research and Development* 20 (3): 198–203.

Robbins, Herbert. 1956. "An Empirical Bayes Approach to Statistics." *Proceedings of the Third Berkeley Symposium on Mathematical Statistics and Probability, Volume 1*, 157–63.

Robbins, Herbert, and Sutton Monro. 1951. "A Stochastic Approximation Method." *The Annals of Mathematical Statistics* 22 (3): 400–407.

Roberts, Gareth O., Andrew Gelman, and Walter R. Gilks. 1997. "Weak Convergence and Optimal Scaling of Random Walk Metropolis Algorithms." *The Annals of Applied Probability* 7 (1): 110–20. https://doi.org/10.1214/aoap/1034625254.

Roberts, Gareth O., and Richard L. Tweedie. 1996. "Exponential Convergence of Langevin Distributions and Their Discrete Approximations." *Bernoulli* 2 (4): 341–63.

Rockafellar, R. T. 1970. *Convex Analysis*. Princeton University Press. https://doi.org/10.1515/9781400873173.

Rombach, Robin, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. 2022. "High-Resolution Image Synthesis with Latent Diffusion Models." *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 10684–95.

Rubanova, Yulia, Ricky T. Q. Chen, and David Duvenaud. 2019. "Latent Ordinary Differential Equations for Irregularly-Sampled Time Series." *Advances in Neural Information Processing Systems* 32.

Rudin, Walter. 1976. *Principles of Mathematical Analysis*. 3rd ed. McGraw-Hill.

Rumelhart, David E, Geoffrey E Hinton, and Ronald J Williams. 1986. "Learning Representations by Back-Propagating Errors." *Nature* 323 (6088): 533–36. https://doi.org/10.1038/323533a0.

Sahami, Mehran, Susan Dumais, David Heckerman, and Eric Horvitz. 1998. "A Bayesian Approach to Filtering Junk e-Mail." *AAAI Workshop on Learning for Text Categorization*.

Samuelson, Paul A. 1965. "Rational Theory of Warrant Pricing." *Industrial Management Review* 6 (2): 13–31.

Särkkä, Simo, and Arno Solin. 2019. *Applied Stochastic Differential Equations*. Cambridge University Press.

Saxe, Andrew M., Yamini Bansal, Joel Dapello, et al. 2018. "On the Information Bottleneck Theory of Deep Learning." *International Conference on Learning Representations*.

Saxe, Andrew M., James L. McClelland, and Surya Ganguli. 2014. "Exact Solutions to the Nonlinear Dynamics of Learning in Deep Linear Neural Networks." *International Conference on Learning Representations*. https://arxiv.org/abs/1312.6120.

Schölkopf, Bernhard, and Alexander J Smola. 2002. *Learning with Kernels: Support Vector*

Machines, Regularization, Optimization, and Beyond. MIT Press. https://doi.org/10.7551/mitpress/4175.001.0001.

Seitzer, Maximilian, Arash Tavakoli, Dimitrije Antic, and Georg Martius. 2022. "On the Pitfalls of Heteroscedastic Uncertainty Estimation with Probabilistic Neural Networks." *International Conference on Learning Representations*. https://arxiv.org/abs/2203.09168.

Shannon, Claude E. 1959. "Coding Theorems for a Discrete Source with a Fidelity Criterion." *IRE National Convention Record* 7 (4): 142–63.

Shannon, Claude Elwood. 1948. "A Mathematical Theory of Communication." *The Bell System Technical Journal* 27 (3): 379–423. https://doi.org/10.1002/j.1538-7305.1948.tb01338.x.

Shannon, Claude Elwood. 1951. "Prediction and Entropy of Printed English." *The Bell System Technical Journal* 30 (1): 50–64. https://doi.org/10.1002/j.1538-7305.1951.tb01366.x.

Shwartz-Ziv, Ravid, and Naftali Tishby. 2017. "Opening the Black Box of Deep Neural Networks via Information." *arXiv Preprint arXiv:1703.00810*.

Simsekli, Umut, Levent Sagun, and Mert Gurbuzbalaban. 2019. "A Tail-Index Analysis of Stochastic Gradient Noise in Deep Neural Networks." *Proceedings of the 36th International Conference on Machine Learning*, Proceedings of machine learning research, vol. 97: 5827–37. https://proceedings.mlr.press/v97/simsekli19a.html.

Slater, Morton. 1950. *Lagrange Multipliers Revisited*. Discussion Paper Mathematics 403. Cowles Commission for Research in Economics.

Smith, Daniel G. A., and Johnnie Gray. 2018. "Opt_einsum: A Python Package for Optimizing Contraction Order for Einsum-Like Expressions." *Journal of Open Source Software* 3 (26): 753.

Sohl-Dickstein, Jascha, Eric Weiss, Niru Maheswaranathan, and Surya Ganguli. 2015. "Deep Unsupervised Learning Using Nonequilibrium Thermodynamics." *International Conference on Machine Learning*, 2256–65. https://proceedings.mlr.press/v37/sohl-dickstein15.html.

Song, Jiaming, and Stefano Ermon. 2020. "Understanding the Limitations of Variational Mutual Information Estimators." *International Conference on Learning Representations*.

Song, Jiaming, Chenlin Meng, and Stefano Ermon. 2021. "Denoising Diffusion Implicit Models." *International Conference on Learning Representations*.

Song, Yang, Conor Durkan, Iain Murray, and Stefano Ermon. 2021. "Maximum Likelihood Training of Score-Based Diffusion Models." *Advances in Neural Information Processing Systems* 34.

Song, Yang, and Stefano Ermon. 2019. "Generative Modeling by Estimating Gradients of the Data Distribution." *Advances in Neural Information Processing Systems* 32. https://arxiv.org/abs/1907.05600.

Song, Yang, Sahaj Garg, Jiaxin Shi, and Stefano Ermon. 2019. "Sliced Score Matching: A Scalable Approach to Density and Score Estimation." *Proceedings of the 35th Conference on Uncertainty in Artificial Intelligence*, Proceedings of machine learning research, vol. 115: 574–84. https://arxiv.org/abs/1905.07088.

Song, Yang, Jascha Sohl-Dickstein, Diederik P Kingma, Abhishek Kumar, Stefano Ermon, and Ben Poole. 2021. "Score-Based Generative Modeling Through Stochastic Differential Equations." *International Conference on Learning Representations*. https://arxiv.org/abs/2011.13456.

Soudry, Daniel, Elad Hoffer, Mor Shpigel Nacson, Suriya Gunasekar, and Nathan Srebro. 2018. "The Implicit Bias of Gradient Descent on Separable Data." *Journal of Machine Learning Research* 19 (70): 1–57.

Squire, William, and George Trapp. 1998. "Using Complex Variables to Estimate Derivatives of Real Functions." *SIAM Review* 40 (1): 110–12. https://doi.org/10.1137/S003614459631241X.

Stein, Charles. 1986. *Approximate Computation of Expectations*. Vol. 7. Institute of Mathematical Statistics Lecture Notes–Monograph Series. Institute of Mathematical Statistics.

Stein, Charles M. 1981. "Estimation of the Mean of a Multivariate Normal Distribution." *Annals of Statistics* 9 (6): 1135–51.

Sterbenz, Pat H. 1974. *Floating-Point Computation*. Prentice-Hall.

Stratonovich, Ruslan L. 1966. "A New Representation for Stochastic Integrals and Equations." *SIAM Journal on Control* 4 (2): 362–71.

Student (Gosset, William S.). 1908. "The Probable Error of a Mean." *Biometrika* 6 (1): 1–25.

Sutskever, Ilya, James Martens, George Dahl, and Geoffrey Hinton. 2013. "On the Importance of Initialization and Momentum in Deep Learning." *International Conference on Machine Learning*, 1139–47. https://proceedings.mlr.press/v28/sutskever13.html.

Szegedy, Christian, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. "Rethinking the Inception Architecture for Computer Vision." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2818–26. https://doi.org/10.1109/cvpr.2016.308.

Székely, Gábor J., and Maria L. Rizzo. 2013. "Energy Statistics: A Class of Statistics Based on Distances." *Journal of Statistical Planning and Inference* 143 (8): 1249–72.

Tao, Terence, and Van Vu. 2010. "Random Matrices: Universality of ESDs and the Circular Law." *Annals of Probability* 38 (5): 2023–65.

Taylor, Adrien B., Julien M. Hendrickx, and François Glineur. 2018. "Exact Worst-Case Convergence Rates of the Proximal Gradient Method for Composite Convex Minimization." *Journal of Optimization Theory and Applications* 178 (2): 455–76.

Tibshirani, Robert. 1996. "Regression Shrinkage and Selection via the Lasso." *Journal of the Royal Statistical Society: Series B* 58 (1): 267–88.

Tieleman, Tijmen, and Geoffrey Hinton. 2012. "Divide the Gradient by a Running Average of Its Recent Magnitude." In *COURSERA: Neural Networks for Machine Learning, Lecture 6.5-Rmsprop*. https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

Tishby, Naftali, Fernando C. Pereira, and William Bialek. 1999. "The Information Bottleneck Method." *Proceedings of the 37th Annual Allerton Conference on Communication, Control, and Computing*, 368–77.

Tishby, Naftali, and Noga Zaslavsky. 2015. "Deep Learning and the Information Bottleneck Principle." *IEEE Information Theory Workshop (ITW)*, 1–5. https://doi.org/10.1109/ITW.2015.7133169.

Tong, Alexander, Kilian Fatras, Nikolay Malkin, et al. 2024. "Improving and Generalizing Flow-Based Generative Models with Minibatch Optimal Transport." *Transactions on Machine Learning Research*.

Townsend, James, Thomas Bird, and David Barber. 2019. "Practical Lossless Compression with Latent Variables Using Bits Back Coding." *International Conference on Learning Representations*. https://arxiv.org/abs/1901.04866.

Trefethen, Lloyd N., and David Bau. 1997. *Numerical Linear Algebra*. SIAM.

Trefethen, Lloyd N., and Mark Embree. 2005. *Spectra and Pseudospectra: The Behavior of Nonnormal Matrices and Operators*. Princeton University Press.

Tschannen, Michael, Josip Djolonga, Paul K. Rubenstein, Sylvain Gelly, and Mario Lucic. 2020. "On Mutual Information Maximization for Representation Learning." *International Conference on Learning Representations*.

Tseng, Paul. 2001. "Convergence of a Block Coordinate Descent Method for Nondifferentiable Minimization." *Journal of Optimization Theory and Applications* 109 (3): 475–94.

Tucker, George, Andriy Mnih, Chris J. Maddison, John Lawson, and Jascha Sohl-Dickstein. 2017. "REBAR: Low-Variance, Unbiased Gradient Estimates for Discrete Latent Variable Models." *Advances in Neural Information Processing Systems* 30.

Uhlenbeck, George E., and Leonard S. Ornstein. 1930. "On the Theory of the Brownian Motion." *Physical Review* 36 (5): 823–41.

Vapnik, V., and A. Chervonenkis. 1971. "On the Uniform Convergence of Relative Frequencies of Events to Their Probabilities." *Theory Probab. Appl.* 16 (2): 264–81. https://doi.org/10.1137/1116025.

Vaswani, Ashish, Noam Shazeer, Niki Parmar, et al. 2017. "Attention Is All You Need." *Advances in Neural Information Processing Systems*, 5998–6008. https://doi.org/10.5555/3295222.3295349.

Vaswani, Sharan, Francis Bach, and Mark Schmidt. 2019. "Fast and Faster Convergence of SGD for over-Parameterized Models and an Accelerated Perceptron." *Proceedings of the Twenty-Second International Conference on Artificial Intelligence and Statistics*, Proceedings of machine learning research, vol. 89: 1195–204.

Veit, Andreas, Michael J Wilber, and Serge Belongie. 2016. "Residual Networks Behave Like Ensembles of Relatively Shallow Networks." *Advances in Neural Information Processing Systems*. https://arxiv.org/abs/1605.06431.

Vershynin, Roman. 2018. *High-Dimensional Probability: An Introduction with Applications in Data Science*. Cambridge University Press.

Vincent, Pascal. 2011. "A Connection Between Score Matching and Denoising Autoencoders." *Neural Computation* 23 (7): 1661–74.

Vovk, Vladimir, Alexander Gammerman, and Glenn Shafer. 2022. *Algorithmic Learning in a Random World*. 2nd ed. Springer. https://doi.org/10.1007/978-3-031-06649-8.

Wallace, Chris S. 1990. "Classification by Minimum-Message-Length Inference." *Advances in Computing and Information – ICCI '90*, Lecture notes in computer science, vol. 468: 72–81.

Wang, Feng, and Huaping Liu. 2021. "Understanding the Behaviour of Contrastive Loss." *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2495–504.

Wang, Sinong, Belinda Z. Li, Madian Khabsa, Han Fang, and Hao Ma. 2020. "Linformer: Self-Attention with Linear Complexity." *arXiv Preprint arXiv:2006.04768*.

Wasserman, Larry. 2013. *All of Statistics: A Concise Course in Statistical Inference*. Springer. https://link.springer.com/book/10.1007/978-0-387-21736-9.

Wasserstein, Ronald L, and Nicole A Lazar. 2016. "The ASA Statement on p-Values: Context, Process, and Purpose." *The American Statistician* 70 (2): 129–33. https://doi.org/10.1080/00031305.2016.1154108.

Weed, Jonathan, and Francis Bach. 2019. "Sharp Asymptotic and Finite-Sample Rates of Convergence of Empirical Measures in Wasserstein Distance." *Bernoulli* 25 (4A): 2620–48.

Welch, Lloyd R. 1974. "Lower Bounds on the Maximum Cross Correlation of Signals." *IEEE Transactions on Information Theory* 20 (3): 397–99. https://doi.org/10.1109/TIT.1974.1055219.

Welford, B. P. 1962. "Note on a Method for Calculating Corrected Sums of Squares and Products." *Technometrics* 4 (3): 419–20.

Welling, Max, and Yee W Teh. 2011. "Bayesian Learning via Stochastic Gradient Langevin Dynamics." *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, 681–88. https://dl.acm.org/doi/10.5555/3104482.3104568.

Wengert, Robert Edwin. 1964. "A Simple Automatic Derivative Evaluation Program." *Communications of the ACM* 7 (8): 463–64. https://doi.org/10.1145/355588.365726.

Wenliang, Li K., and Heishiro Kanagawa. 2020. "Blindness of Score-Based Methods to Isolated Components and Mixing Proportions." *arXiv Preprint arXiv:2008.10087*. https://arxiv.org/abs/2008.10087.

White, Halbert. 1982. "Maximum Likelihood Estimation of Misspecified Models." *Econometrica* 50 (1): 1–25.

White, Tom. 2016. "Sampling Generative Networks." *arXiv Preprint arXiv:1609.04468*.

Wiener, Norbert. 1923. "Differential-Space." *Journal of Mathematics and Physics* 2: 131–74.

Williams, Ronald J. 1992. "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning." *Machine Learning* 8: 229–56.

Witten, Ian H., Radford M. Neal, and John G. Cleary. 1987. "Arithmetic Coding for Data Compression." *Communications of the ACM* 30 (6): 520–40.

Wong, Eugene, and Moshe Zakai. 1965. "On the Convergence of Ordinary Integrals to Stochastic Integrals." *The Annals of Mathematical Statistics* 36 (5): 1560–64.

Woodruff, David P. 2014. "Sketching as a Tool for Numerical Linear Algebra." *Foundations and Trends in Theoretical Computer Science* 10 (1–2): 1–157.

Wu, C. F. Jeff. 1983. "On the Convergence Properties of the EM Algorithm." *Annals of Statistics* 11 (1): 95–103.

Wu Fung, Samy, Howard Heaton, Qiuwei Li, Daniel McKenzie, Stanley Osher, and Wotao Yin. 2022. "JFB: Jacobian-Free Backpropagation for Implicit Networks." *Proceedings of the AAAI Conference on Artificial Intelligence* 36: 6648–56. https://doi.org/10.1609/aaai.v36i6.20619.

Xiao, Han, Kashif Rasul, and Roland Vollgraf. 2017. "Fashion-MNIST: A Novel Image Dataset for Benchmarking Machine Learning Algorithms." *ArXiv:1708.07747*. https://arxiv.org/abs/1708.07747.

Xie, Shuo, and Zhiyuan Li. 2024. "Implicit Bias of AdamW: ℓ_∞ Norm Constrained Optimization." *arXiv Preprint arXiv:2404.04454*.

Xu, Yilun, Shengjia Zhao, Jiaming Song, Russell Stewart, and Stefano Ermon. 2020. "A Theory of Usable Information Under Computational Constraints." *International Conference on Learning Representations*.

Yin, Y. Q., Z. D. Bai, and P. R. Krishnaiah. 1988. "On the Limit of the Largest Eigenvalue of the Large Dimensional Sample Covariance Matrix." *Probability Theory and Related Fields* 78 (4): 509–21.

Zadrozny, Bianca, and Charles Elkan. 2002. "Transforming Classifier Scores into Accurate

Multiclass Probability Estimates." *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 694–99. https://doi.org/10.1145/775047.775151.

Zhang, Chiyuan, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. 2021. "Understanding Deep Learning (Still) Requires Rethinking Generalization." *Communications of the ACM* 64 (3): 107–15. https://doi.org/10.1145/3446776.

Zhang, Jingzhao, Tianxing He, Suvrit Sra, and Ali Jadbabaie. 2020. "Why Gradient Clipping Accelerates Training: A Theoretical Justification for Adaptivity." *International Conference on Learning Representations*.

Zhang, Qinsheng, and Yongxin Chen. 2023. "Fast Sampling of Diffusion Models with Exponential Integrator." *International Conference on Learning Representations*. https://arxiv.org/abs/2204.13902.

Zinkevich, Martin. 2003. "Online Convex Programming and Generalized Infinitesimal Gradient Ascent." *Proceedings of the 20th International Conference on Machine Learning*, 928–36.

Ziv, Jacob, and Abraham Lempel. 1977. "A Universal Algorithm for Sequential Data Compression." *IEEE Transactions on Information Theory* 23 (3): 337–43. https://doi.org/10.1109/TIT.1977.1055714.

Ziv, Jacob, and Abraham Lempel. 1978. "Compression of Individual Sequences via Variable-Rate Coding." *IEEE Transactions on Information Theory* 24 (5): 530–36. https://doi.org/10.1109/TIT.1978.1055934.