Dive into Deep Learning

BASICS

Alexander J Smola

Aston Zhang · Zachary C Lipton · Mu Li · Alexander J Smola (1st edition)

Table of contents

Basics **1**

1 Preliminaries **3**
1.1 Data Manipulation . . . 3
1.1.1 Tensor Creation . . . 4
1.1.2 Reshaping and Indexing . . . 6
1.1.3 Operations . . . 11
1.1.4 Memory and Conversion . . . 14
1.1.5 Summary . . . 16
1.1.6 Exercises . . . 16
1.2 Data Preprocessing . . . 17
1.2.1 Loading and Inspecting . . . 18
1.2.2 Cleaning and Encoding . . . 20
1.2.3 Scaling and Converting . . . 23
1.2.4 Summary . . . 25
1.2.5 Exercises . . . 25
1.3 Linear Algebra . . . 27
1.3.1 Scalars, Vectors, Matrices, and Tensors . . . 28
1.3.2 Elementwise Operations and Reductions . . . 30
1.3.3 Products . . . 31
1.3.4 Norms . . . 34
1.3.5 Eigenvectors and Eigenvalues . . . 37
1.3.6 Summary . . . 39
1.3.7 Exercises . . . 39
1.4 Calculus . . . 41
1.4.1 Derivatives . . . 42
1.4.2 Partial Derivatives and Gradients . . . 45
1.4.3 The Chain Rule . . . 49
1.4.4 Summary . . . 50
1.4.5 Exercises . . . 50
1.5 Automatic Differentiation . . . 52
1.5.1 Computing Gradients . . . 52
1.5.2 Controlling the Graph . . . 55
1.5.3 Differentiating Programs . . . 56
1.5.4 Forward and Reverse Mode . . . 57
1.5.5 Summary . . . 58
1.5.6 Exercises . . . 58

1.6 Probability and Statistics . . . 59
1.6.1 Sampling . . . 60
1.6.2 Events and Random Variables . . . 61
1.6.3 Expectation and Variance . . . 64
1.6.4 Estimation Error . . . 65
1.6.5 Conditional Probability . . . 67
1.6.6 Summary . . . 69
1.6.7 Exercises . . . 70

2 Linear Regression **73**
2.1 Linear Regression . . . 73
2.1.1 The Model and the Loss . . . 74
2.1.2 Minimizing the Loss . . . 77
2.1.3 Squared Loss and Gaussian Noise . . . 82
2.1.4 Linear Regression as a Neural Network . . . 85
2.1.5 Summary . . . 85
2.1.6 Exercises . . . 86
2.2 Synthetic Data . . . 88
2.2.1 Generating the Data . . . 88
2.2.2 Inspecting the Data . . . 89
2.2.3 Recovering the Parameters . . . 90
2.2.4 Summary . . . 90
2.2.5 Exercises . . . 90
2.3 Linear Regression Implementation from Scratch . . . 91
2.3.1 Model, Loss, and Gradient . . . 92
2.3.2 Training from Scratch . . . 93
2.3.3 Checking the Implementation . . . 94
2.3.4 Summary . . . 98
2.3.5 Exercises . . . 98
2.4 Object-Oriented Design for Implementation . . . 99
2.4.1 Utilities . . . 100
2.4.2 Models . . . 101
2.4.3 Data . . . 103
2.4.4 Training . . . 105
2.4.5 Training Linear Regression Again . . . 108
2.4.6 Summary . . . 109
2.4.7 Exercises . . . 110
2.5 Concise Implementation of Linear Regression . . . 111
2.5.1 The Fully Connected Layer . . . 111
2.5.2 The Loss Function . . . 111
2.5.3 The Optimizer . . . 112
2.5.4 Training with Framework Components . . . 112
2.5.5 Summary . . . 113
2.5.6 Exercises . . . 113
2.6 Generalization . . . 114
2.6.1 Training Error and Generalization Error . . . 115
2.6.2 Underfitting and Overfitting . . . 115
2.6.3 Model Selection . . . 122
2.6.4 Summary . . . 125
2.6.5 Exercises . . . 125

2.7 Weight Decay . . . 127
2.7.1 Norm Penalties . . . 128
2.7.2 Weight Decay from Scratch . . . 130
2.7.3 Shrinkage Along Eigenvectors . . . 133
2.7.4 Concise Implementation . . . 136
2.7.5 Summary . . . 138
2.7.6 Exercises . . . 138

3 Linear Classification **141**
3.1 Softmax Regression . . . 142
3.1.1 The Model . . . 142
3.1.2 The Cross-Entropy Loss . . . 145
3.1.3 Accuracy and Confidence . . . 147
3.1.4 Summary . . . 148
3.1.5 Exercises . . . 149
3.2 Image Classification Data . . . 151
3.2.1 Loading the Dataset . . . 152
3.2.2 Reading Minibatches . . . 153
3.2.3 Inspecting the Images . . . 154
3.2.4 Summary . . . 155
3.2.5 Exercises . . . 155
3.3 Softmax Regression Implementation . . . 156
3.3.1 The Softmax . . . 156
3.3.2 Loss and Accuracy . . . 157
3.3.3 The Model and the Loss . . . 158
3.3.4 Training and Evaluation . . . 160
3.3.5 Concise Implementation . . . 164
3.3.6 Summary . . . 166
3.3.7 Exercises . . . 167
3.4 Measuring Classifiers . . . 168
3.4.1 The Limits of Accuracy . . . 169
3.4.2 Calibration . . . 170
3.4.3 Summary . . . 173
3.4.4 Exercises . . . 173
3.5 Generalization in Classification . . . 175
3.5.1 The Test Set . . . 175
3.5.2 Test Set Reuse . . . 177
3.5.3 Statistical Learning Theory . . . 179
3.5.4 Summary . . . 180
3.5.5 Exercises . . . 180
3.6 Environment and Distribution Shift . . . 182
3.6.1 Types of Distribution Shift . . . 182
3.6.2 Covariate Shift Correction . . . 183
3.6.3 Label Shift Correction . . . 190
3.6.4 Concept Shift and Feedback . . . 192
3.6.5 Summary . . . 192
3.6.6 Exercises . . . 193

4 Multilayer Perceptrons **195**
4.1 Multilayer Perceptrons . . . 196

4.1.1 Limits of Linear Models: The XOR Problem . . . 197
4.1.2 Hidden Layers . . . 198
4.1.3 Expressive Power . . . 200
4.1.4 Activation Functions . . . 201
4.1.5 Summary . . . 203
4.1.6 Exercises . . . 203
4.2 Implementation . . . 204
4.2.1 Implementation from Scratch . . . 204
4.2.2 Concise Implementation . . . 207
4.2.3 Comparison with Softmax Regression . . . 208
4.2.4 Summary . . . 211
4.2.5 Exercises . . . 211
4.3 Backpropagation . . . 212
4.3.1 Computational Graphs . . . 212
4.3.2 Backpropagation through an MLP . . . 213
4.3.3 Memory and Computation . . . 217
4.3.4 Summary . . . 217
4.3.5 Exercises . . . 217
4.4 Numerical Stability and Initialization . . . 219
4.4.1 Vanishing and Exploding Gradients . . . 219
4.4.2 Parameter Initialization . . . 221
4.4.3 Symmetry among Hidden Units . . . 224
4.4.4 Summary . . . 225
4.4.5 Exercises . . . 225
4.5 Generalization in Deep Learning . . . 227
4.5.1 Memorization and Capacity . . . 227
4.5.2 Double Descent . . . 230
4.5.3 Implicit Bias of Gradient Descent . . . 232
4.5.4 Early Stopping . . . 233
4.5.5 Summary . . . 235
4.5.6 Exercises . . . 235
4.6 Dropout . . . 237
4.6.1 Noise in Hidden Layers . . . 237
4.6.2 Implementation . . . 238
4.6.3 Effects of Dropout under Label Noise . . . 241
4.6.4 Summary . . . 242
4.6.5 Exercises . . . 242
4.7 Predicting House Prices on Kaggle . . . 244
4.7.1 The Dataset and the Log-RMSE Metric . . . 244
4.7.2 Leakage-Free Preprocessing . . . 245
4.7.3 Model Selection via K-Fold Cross-Validation . . . 246
4.7.4 Submitting Predictions . . . 249
4.7.5 Summary . . . 250
4.7.6 Exercises . . . 250

5 Computation **253**
Resources and Further Reading . . . 254
5.1 Modules and Model Construction . . . 255
5.1.1 The Module Abstraction . . . 255
5.1.2 Module Containers . . . 258

5.1.3 Forward Computation . . . 260
5.1.4 Lazy Initialization . . . 262
5.1.5 Models from a Config . . . 263
5.1.6 Summary . . . 265
5.1.7 Exercises . . . 265
5.2 Parameters, State, and Memory . . . 266
5.2.1 The State Tree . . . 266
5.2.2 Training Memory . . . 269
5.2.3 Shared and Frozen Parameters . . . 271
5.2.4 Summary . . . 274
5.2.5 Exercises . . . 274
5.3 Initialization . . . 275
5.3.1 Applying Initializers . . . 276
5.3.2 Custom Initializers and Direct Edits . . . 277
5.3.3 Initialization of Residual Networks . . . 278
5.3.4 Summary . . . 282
5.3.5 Exercises . . . 282
5.4 Custom Layers and Functions . . . 283
5.4.1 Layers without Parameters . . . 284
5.4.2 Layers with Parameters . . . 284
5.4.3 Layers with Fixed State . . . 287
5.4.4 Functions with Custom Gradients . . . 289
5.4.5 Summary . . . 292
5.4.6 Exercises . . . 292
5.5 Numerical Formats . . . 293
5.5.1 Floating-Point Formats . . . 293
5.5.2 Dtype Promotion and Model Casting . . . 295
5.5.3 Mixed-Precision Training . . . 297
5.5.4 Numerical Failure Modes . . . 302
5.5.5 Summary . . . 304
5.5.6 Exercises . . . 304
5.6 Saving, Loading, and Pretrained Weights . . . 305
5.6.1 Model State . . . 306
5.6.2 Pickle and safetensors . . . 306
5.6.3 Training Checkpoints . . . 309
5.6.4 Pretrained Weights . . . 313
5.6.5 Summary . . . 314
5.6.6 Exercises . . . 314
5.7 GPUs, Devices, and Memory . . . 315
5.7.1 Devices and Placement . . . 316
5.7.2 Asynchronous Execution . . . 319
5.7.3 GPU Memory . . . 321
5.7.4 Device-Aware Trainer . . . 326
5.7.5 Summary . . . 328
5.7.6 Exercises . . . 328
5.8 Reproducibility and Inspection . . . 329
5.8.1 Seeds and Random Number Generators . . . 330
5.8.2 Deterministic Arithmetic . . . 333
5.8.3 Model Inspection with Hooks . . . 334
5.8.4 Summary . . . 338

5.8.5 Exercises . . . 338

6 Convolutional Neural Networks 341
Resources and Further Reading . . . 342
6.1 From Fully Connected Layers to Convolutions . . . 343
6.1.1 Equivariance and Invariance . . . 344
6.1.2 Constraints on the Fully Connected Layer . . . 346
6.1.3 Convolution and Cross-Correlation . . . 348
6.1.4 Channels . . . 349
6.1.5 Summary . . . 350
6.1.6 Exercises . . . 351
6.2 Convolutions for Images . . . 352
6.2.1 Cross-Correlation . . . 352
6.2.2 Convolutional Layers . . . 353
6.2.3 Convolution as Matrix Multiplication . . . 356
6.2.4 Feature Maps and Receptive Fields . . . 357
6.2.5 Summary . . . 358
6.2.6 Exercises . . . 360
6.3 Padding, Stride, and Dilation . . . 361
6.3.1 Padding . . . 361
6.3.2 Stride . . . 364
6.3.3 Dilation . . . 366
6.3.4 Summary . . . 368
6.3.5 Exercises . . . 368
6.4 Multiple Input and Multiple Output Channels . . . 369
6.4.1 Multiple Input Channels . . . 370
6.4.2 Multiple Output Channels . . . 371
6.4.3 The 1×1 Convolution . . . 373
6.4.4 Summary . . . 374
6.4.5 Exercises . . . 374
6.5 Pooling . . . 375
6.5.1 Maximum Pooling and Average Pooling . . . 375
6.5.2 Tolerance to Small Shifts . . . 377
6.5.3 Global Pooling . . . 378
6.5.4 Padding and Stride . . . 379
6.5.5 Multiple Channels . . . 380
6.5.6 Summary . . . 381
6.5.7 Exercises . . . 381
6.6 LeNet . . . 382
6.6.1 The LeNet Architecture . . . 382
6.6.2 Layer Shapes . . . 384
6.6.3 Parameters and Multiply-Adds . . . 386
6.6.4 Training . . . 386
6.6.5 Summary . . . 390
6.6.6 Exercises . . . 391

7 Modern Convolutional Neural Networks 393
7.1 AlexNet and Learned Image Representations . . . 394
7.1.1 Learned Features, Data, and Hardware . . . 394
7.1.2 The AlexNet Architecture . . . 395

7.1.3 Training on Fashion-MNIST . . . 399
7.1.4 Summary . . . 402
7.1.5 Exercises . . . 402
7.2 Blocks, Bottlenecks, and Branches: VGG, NiN, GoogLeNet . . . 403
7.2.1 VGG: Blocks as the Unit of Design . . . 404
7.2.2 NiN: 1×1 Convolutions and Global Average Pooling . . . 408
7.2.3 GoogLeNet: Multi-Branch Blocks . . . 413
7.2.4 Comparing the Networks . . . 416
7.2.5 Summary . . . 417
7.2.6 Exercises . . . 417
7.3 Normalization Layers . . . 418
7.3.1 Batch Normalization . . . 419
7.3.2 Implementation from Scratch . . . 421
7.3.3 LeNet with Batch Normalization . . . 423
7.3.4 Normalization without Batch Statistics . . . 426
7.3.5 Summary . . . 428
7.3.6 Exercises . . . 428
7.4 Residual Networks . . . 429
7.4.1 Function Classes and Depth . . . 430
7.4.2 Residual Blocks . . . 430
7.4.3 Plain and Residual Networks Compared . . . 433
7.4.4 ResNet-18 . . . 436
7.4.5 ResNeXt and DenseNet . . . 439
7.4.6 Summary . . . 443
7.4.7 Exercises . . . 444
7.5 Efficient Convnets . . . 445
7.5.1 Depthwise-Separable Networks . . . 446
7.5.2 Scaling and Searching . . . 456
7.5.3 Structural Re-parameterization . . . 457
7.5.4 Summary . . . 460
7.5.5 Exercises . . . 460
7.6 Modern Training Recipes for Convnets . . . 462
7.6.1 Training Recipes from 2015 to 2022 . . . 462
7.6.2 Implementing the Ingredients . . . 463
7.6.3 Two Recipes on One Network . . . 468
7.6.4 Summary . . . 472
7.6.5 Exercises . . . 472
7.7 ConvNeXt: Modernizing a ResNet Step by Step . . . 473
7.7.1 The Modernization Roadmap . . . 474
7.7.2 Implementation . . . 478
7.7.3 A Small Roadmap on Fashion-MNIST . . . 480
7.7.4 Limits and Open Problems . . . 483
7.7.5 Summary . . . 483
7.7.6 Exercises . . . 483
7.8 Convolutional Network Design Spaces . . . 484
7.8.1 The AnyNet Design Space . . . 485
7.8.2 Comparing Design Spaces . . . 488
7.8.3 RegNet . . . 492
7.8.4 Summary . . . 496
7.8.5 Exercises . . . 497

8 Sequence Models **499**
8.1 Working with Sequences . . . 500
8.1.1 Sequential Data . . . 500
8.1.2 Autoregressive Models . . . 501
8.1.3 Markov Models and Stationarity . . . 503
8.1.4 Forecasting a Noisy Sine Wave . . . 503
8.1.5 Summary . . . 511
8.1.6 Exercises . . . 511
8.2 From Text to Tokens . . . 512
8.2.1 Reading the Dataset . . . 512
8.2.2 Characters, Words, and Bytes . . . 513
8.2.3 Byte Pair Encoding . . . 517
8.2.4 Special Tokens . . . 524
8.2.5 Tokenizers in the Wild . . . 525
8.2.6 Summary . . . 528
8.2.7 Exercises . . . 528
8.3 Language Models . . . 529
8.3.1 n-gram Language Models . . . 530
8.3.2 Perplexity and Smoothing . . . 536
8.3.3 Partitioning Sequences . . . 542
8.3.4 Summary . . . 544
8.3.5 Exercises . . . 544
8.4 Recurrent Neural Networks . . . 546
8.4.1 Recurrent Neural Networks with Hidden States . . . 546
8.4.2 An RNN Language Model from Scratch . . . 550
8.4.3 Gradient Clipping . . . 552
8.4.4 Training and Evaluation . . . 553
8.4.5 Text Generation . . . 559
8.4.6 Concise Implementation . . . 560
8.4.7 Summary . . . 562
8.4.8 Exercises . . . 562
8.5 Backpropagation Through Time . . . 564
8.5.1 Vanishing and Exploding Gradients . . . 565
8.5.2 Truncated Backpropagation Through Time . . . 570
8.5.3 Summary . . . 570
8.5.4 Exercises . . . 571
8.6 Decoding and Generation . . . 572
8.6.1 The Decoding Problem . . . 572
8.6.2 Greedy Decoding . . . 575
8.6.3 Beam Search . . . 576
8.6.4 Sampling . . . 579
8.6.5 Evaluation and Cost . . . 584
8.6.6 Summary . . . 586
8.6.7 Exercises . . . 586

References **589**

Basics

In the Introduction, we saw how machine learning replaces hand-written rules with parameterized programs that learn directly from examples. Suppose you are handed tens of thousands of small grayscale photographs of clothing and asked to build a system that identifies whether each image shows a T-shirt, a sneaker, a coat, or a handbag. No programmer knows how to write explicit `if-else` rules that map raw pixel intensities to clothing categories reliably. Yet by the end of the first three chapters of this part, you will have built and trained a working classifier that does exactly that—and by the end of the part, you will have progressed to deep convolutional networks that recognize complex visual scenes and recurrent networks that learn to generate English prose from scratch.

Our approach throughout *Basics* is guided by a simple principle: to truly understand deep learning, you should build its core mechanisms with your own hands before relying on black-box library calls. Across these eight chapters, we implement nearly every foundational model, layer, loss function, and optimizer twice: first *from scratch*, relying only on basic tensor arithmetic and automatic differentiation so that every mathematical operation is visible in code, and then *concisely*, using the high-level abstractions of modern deep learning frameworks (PyTorch, JAX, TensorFlow, and MXNet). Once you have seen how a component works from the ground up, we use its concise framework implementation as a building block in later chapters.

The eight chapters of this part unfold in three stages:

- **The Toolkit and the Core Recipe (Chapter 1 to Chapter 3).** We begin on models simple enough to analyze completely. Chapter 1 equips you with the essential working tools of deep learning: storing and manipulating data in multidimensional arrays (*tensors*), preprocessing tabular data, applying linear algebra and calculus in code, computing derivatives automatically via *autograd*, and reasoning about uncertainty with probability. Chapter 2 and Chapter 3 then introduce the universal training loop of machine learning on linear models. In Chapter 2, we fit continuous targets with linear regression, squared-error loss, and minibatch stochastic gradient descent, and confront the central statistical challenge of *generalization*—why fitting our training data too closely leads to overfitting, and how regularization methods such as weight decay keep models honest. In Chapter 3, we turn from continuous numbers to discrete categories, developing softmax regression and cross-entropy loss to classify clothing photographs on the Fashion-MNIST benchmark, while examining how deployed classifiers behave when the real-world data distribution shifts away from the training set.
- **Going Deep and Mastering the Software Engine (Chapter 4 to Chapter 5).** Linear models can only draw flat decision boundaries between classes. Chapter 4 breaks through that limitation by stacking hidden layers of units equipped with nonlinear ac-

tivation functions to create *multilayer perceptrons* (MLPs). We trace how signals and gradients flow forward and backward through a deep network via backpropagation, explore why deep networks can suffer from vanishing or exploding gradients if initialized carelessly, and introduce *dropout* as a powerful regularizer for high-capacity networks. Chapter 5 then opens up the software architecture of modern deep learning frameworks: how complex networks are assembled from reusable modular blocks, how parameters are shared, inspected, and initialized, how custom layers and forward passes are defined, how trained weights are saved and loaded, and how computations are moved onto GPUs.

- **Building Domain Structure into the Architecture (Chapter 6 to Chapter 8).** Fully connected networks treat every input feature symmetrically, ignoring whether the inputs are pixels arranged on a two-dimensional image grid or words arranged in a sequential sentence. The final three chapters show how encoding the natural symmetries of the data directly into the network architecture leads to dramatic gains in statistical and computational efficiency. For images, Chapter 6 develops *convolutional neural networks* (CNNs) from the principles of translation invariance and spatial locality, introducing padding, striding, multiple input/output channels, and pooling to build the classic LeNet architecture. Chapter 7 then traces the architectural milestones that carried computer vision into the modern era: AlexNet, repeatable block design (VGG), 1×1 convolutions (NiN), multi-branch parallel blocks (GoogLeNet), *batch normalization*, *residual connections* (ResNet), and DenseNet, culminating in principled architecture design spaces (AnyNet and RegNet). Finally, Chapter 8 turns to sequential data such as text and time series, developing autoregressive language modeling, tokenization, *recurrent neural networks* (RNNs), backpropagation through time, and the classical gated architectures—Gated Recurrent Units (GRUs), Long Short-Term Memory (LSTM) networks, deep stacked RNNs, and bidirectional RNNs—all tested by training a character-level language model to generate new passages in the style of H. G. Wells's *The Time Machine*.

The first five chapters (Chapter 1 through Chapter 5) form a sequential spine and are best read in order. Once you have completed them, the computer vision chapters (Chapter 6 and Chapter 7) and the sequence modeling chapter (Chapter 8) do not depend on one another and can be studied in either order. Together, they establish the conceptual and engineering habits that the *Advanced* part builds upon.

Chapter 1

Preliminaries

To prepare for your dive into deep learning, you will need a few essential survival skills: (i) techniques for storing and manipulating multidimensional data in memory (Section 1.1); (ii) tools for ingesting and cleaning tabular data from raw files, complete with missing entries and categorical text fields (Section 1.2); (iii) fluency in the linear-algebraic operations that transform high-dimensional representations and measure their size (Section 1.3); (iv) differential calculus to determine which direction to adjust each parameter in order to decrease a loss function (Section 1.4); (v) automatic differentiation so that the framework handles the chain rule across millions of parameters for you (Section 1.5); and (vi) a working grasp of probability and statistics, our primary language for reasoning under uncertainty and distinguishing genuine signal from sampling noise (Section 1.6).

What deep learning adds to classical linear algebra, calculus, and probability is *scale*: we apply them to millions or billions of variables where calculation by hand is out of the question. Doing so relies on two software pillars that every subsequent chapter takes for granted: *tensor libraries*, which apply compiled operations to entire arrays of numbers in parallel on CPUs and GPUs, and *automatic differentiation*, which differentiates arbitrary numerical programs mechanically.

This chapter provides a rapid, hands-on crash course in these prerequisites. If you are already comfortable with NumPy, linear algebra, and multivariable calculus, you can skim the familiar parts, though we still recommend reading Section 1.5 on automatic differentiation and Section 1.6 on standard errors and statistical estimation, as later chapters rely on both constantly. For a deeper theoretical foundation with complete proofs and derivations, our *Mathematics for Deep Learning* appendix develops linear algebra (Chapter 33), calculus (Chapter 34), and probability and statistics (Chapter 36) in full detail.

1.1 Data Manipulation

To get anything done in machine learning, we need a way to store and manipulate data efficiently in memory. We start with n-dimensional arrays, which in deep learning are called *tensors*.

To see why specialized array containers are indispensable, consider a single *minibatch* of 256 grayscale photographs of resolution 28×28 pixels, containing $256 \times 28 \times 28 = 200{,}704$

numbers. Image files store each pixel as an integer in $\{0, \dots, 255\}$, which we rescale to $[0, 1]$ by dividing by 255. In plain Python, three nested loops would execute $200{,}704$ separate scalar divisions in the interpreter. In a tensor library, the single expression `images / 255` dispatches to compiled C++ or CUDA kernels that divide all $200{,}704$ numbers in parallel on a CPU or GPU. Throughout deep learning, we operate on whole tensors at once rather than iterating over individual scalar elements in Python loops.

Across all modern deep learning frameworks, the tensor class closely mirrors NumPy's `ndarray` (Harris et al. 2020), with two crucial additions:

1. **Hardware acceleration:** framework tensors execute across GPUs and specialized AI accelerators, whereas NumPy runs on CPUs.
2. **Automatic differentiation:** frameworks track computations on tensors and compute gradients automatically via backpropagation (Section 1.5).

In PyTorch the tensor class is `torch.Tensor`. We import PyTorch (`torch`).

```
import torch
```

1.1.1 Tensor Creation

1.1.1.1 Shape

A tensor arranges numerical values along one or more *axes*. A tensor with one axis is a *vector*, with two axes a *matrix*, and with $k > 2$ axes a k^{th}*-order tensor* (its number of axes is its *order*, exposed in code as `ndim`). Our running minibatch of grayscale images is a 4^{th}-order tensor of shape $(256, 1, 28, 28)$: axis 0 indexes the image in the batch, axis 1 the color channel (1 for grayscale, 3 for RGB), axis 2 the vertical pixel row, and axis 3 the horizontal pixel column.

Calling `arange(n)` creates a 1D vector of evenly spaced values from 0 (included) to n (excluded). Newly created tensors reside in main memory for CPU computation by default; here we request 32-bit floating-point values (`float32`):

```
x = torch.arange(12, dtype=torch.float32)
x
```

```
tensor([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11.])
```

Each value in a tensor is an *element*. We can query the total number of elements in `x` (its *size*):

In PyTorch, the total element count is returned by `numel()`.

```
x.numel()
```

```
12
```

Next, we inspect the tensor's *shape* tuple via `x.shape`, which lists its length along each axis (here a 1-tuple `(12,)` for our 12-element vector):

```
x.shape
```

```
torch.Size([12])
```

Figure 1.1 illustrates tensors of orders 0 through 3. A single number (*scalar*) is a 0^{th}-order tensor with empty shape `()`. For any tensor, the size equals the product of the lengths in its shape tuple ($256 \times 1 \times 28 \times 28 = 200{,}704$ for our image minibatch).

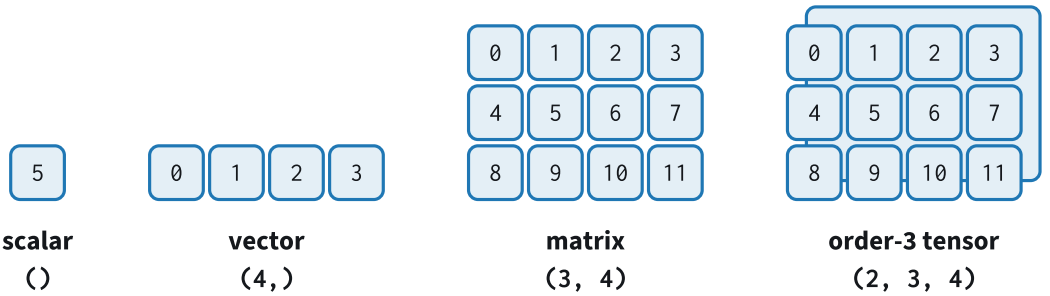


Figure 1.1: Tensors with zero, one, two, and three axes. The shape lists the length of each axis, and the size is the product of these lengths: 1, 4, 12, and 24 elements.

1.1.1.2 Data Types

Every tensor carries a *data type* (`dtype`) specifying the binary representation of each element. Deep learning models perform most arithmetic in 32-bit floating point (`float32`, 4 bytes per element, ≈ 7 decimal digits of precision) or 16-bit formats (`float16` / `bfloat16`, 2 bytes per element; see Section 5.5). Raw image pixels are typically stored as 8-bit unsigned integers (`uint8`, 1 byte per pixel for $\{0, \dots, 255\}$), so our $200{,}704$-pixel minibatch occupies 0.2 MB as `uint8` and 0.8 MB as `float32`.

`torch.arange` defaults to `int64` on integer arguments; cast with `x.to(torch.int32)`.

1.1.1.3 Initial Values

The functions `zeros` and `ones` construct a tensor of any shape filled with 0s or 1s (used for biases or buffers):

```
torch.zeros((2, 3, 4))
```

```
tensor([[[0., 0., 0., 0.],
         [0., 0., 0., 0.],
         [0., 0., 0., 0.]],

        [[0., 0., 0., 0.],
         [0., 0., 0., 0.],
         [0., 0., 0., 0.]]])
```

Neural network weights are typically initialized by sampling elements independently from a probability distribution (Section 4.4). Below, we draw a 3×4 matrix from a standard Gaussian distribution (mean 0, standard deviation 1):

```
torch.randn(3, 4)
```

```
tensor([[ 0.2763, -0.8037, -1.2709,  0.2572],
        [ 0.3519, -1.3422, -0.7240,  0.4446],
        [ 1.1666,  0.8201,  0.8971,  0.9919]])
```

Computer-generated random numbers are *pseudorandom*: a deterministic algorithm computes them from an integer *seed*, so starting again from the same seed reproduces the exact same sequence (`a` and `c` below match, whereas `b` does not):

PyTorch resets its global generator via `torch.manual_seed`.

```
torch.manual_seed(42)
a = torch.randn(3)
b = torch.randn(3)  # The generator has moved on
torch.manual_seed(42)
c = torch.randn(3)  # Same seed: c repeats a
a, b, c
```

```
(tensor([0.3367, 0.1288, 0.2345]),
 tensor([ 0.2303, -1.1229, -0.1863]),
 tensor([0.3367, 0.1288, 0.2345]))
```

When we know the exact entries of a small tensor ahead of time, we can construct it directly from nested Python lists (here creating a 3×4 matrix where the outer list gives the three rows and each inner list supplies four columns):

```
torch.tensor([[2, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
```

```
tensor([[2, 1, 4, 3],
        [1, 2, 3, 4],
        [4, 3, 2, 1]])
```

Finally, we construct our running synthetic minibatch `images` of shape `(256, 1, 28, 28)` and type `uint8`, filled with random integers in $\{0, \dots, 255\}$:

```
# 256 grayscale images of 28 x 28 pixels, each pixel an integer in [0, 255]
images = torch.randint(0, 256, (256, 1, 28, 28), dtype=torch.uint8)
images.shape, images.dtype
```

```
(torch.Size([256, 1, 28, 28]), torch.uint8)
```

1.1.2 Reshaping and Indexing

1.1.2.1 Reshaping

Calling `reshape` reconfigures a tensor's axes while preserving its element count and row-major order in memory (Figure 1.2). Here we reshape our 12-element vector `x` into a 3×4 matrix `X`:

```
X = x.reshape(3, 4)
X
```

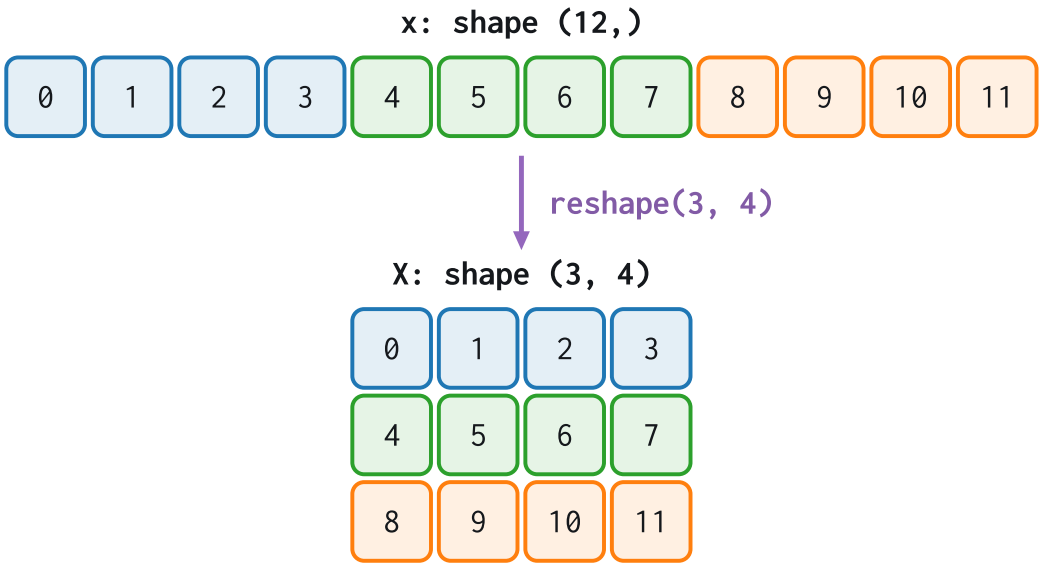


Figure 1.2: Reshaping a vector of 12 elements into a 3×4 matrix. The elements keep their order and fill the matrix one row at a time.

```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.]])
```

Because the total size is fixed, passing `-1` for one dimension lets the framework infer its length automatically; for example, we can flatten our $(256, 1, 28, 28)$ image minibatch into a $(256, 784)$ matrix with one image per row:

```
images.reshape(256, -1).shape
```

```
torch.Size([256, 784])
```

Indexing with `None` inserts a new singleton axis of length 1 at that position, while `squeeze` removes an axis of length 1:

In PyTorch, `image.unsqueeze(0)` also inserts a singleton axis at position `0`.

```
image = images[0]
image[None].shape, image.squeeze(0).shape
```

```
(torch.Size([1, 1, 28, 28]), torch.Size([28, 28]))
```

A reshape keeps the elements in their order and only regroups them, so it can never exchange two axes. Some libraries expect an image minibatch with the channel axis last, as $(256, 28, 28, 1)$. Getting there from our $(256, 1, 28, 28)$ minibatch means moving axis 1 to the end. This is a *transpose*: we list the old axes in their new order, here $(0, 2, 3, 1)$. A matrix has only one such reordering, the exchange of rows and columns, written `X.T`. We compare it with a reshape to the same shape:

In PyTorch, the method that reorders all axes is called `permute`; `transpose(i, j)` exchanges only the two axes `i` and `j`.

```
images.permute(0, 2, 3, 1).shape, X.reshape(4, 3), X.T
```

```
(torch.Size([256, 28, 28, 1]),
 tensor([[ 0.,  1.,  2.],
         [ 3.,  4.,  5.],
         [ 6.,  7.,  8.],
         [ 9., 10., 11.]]),
 tensor([[ 0.,  4.,  8.],
         [ 1.,  5.,  9.],
         [ 2.,  6., 10.],
         [ 3.,  7., 11.]]))
```

Both matrices are 4×3, but the reshape still reads $0, 1, 2, \ldots$ along its rows, whereas the transpose has turned the rows of `X` into columns. Since the shapes agree, a reshape written where a transpose was meant raises no error. It only hands the next layer the right numbers in the wrong places.

1.1.2.2 Indexing and Slicing

Elements and sub-tensors are accessed via 0-based integer indexing (negative indices count backward from the end) and half-open slices `start:stop` (including `start`, excluding `stop`), separated by commas across axes:

```
X[-1], X[1:3]
```

```
(tensor([ 8.,  9., 10., 11.]),
 tensor([[ 4.,  5.,  6.,  7.],
         [ 8.,  9., 10., 11.]]))
```

Notice the shape difference (Figure 1.3): an integer index (`X[-1]`) collapses that axis to return a vector of shape `(4,)`, whereas a slice (`X[1:3]`) preserves the axis to return a matrix of shape `(2, 4)`.

On our image minibatch of shape $(256, 1, 28, 28)$, `images[0]` extracts the first image $(1, 28, 28)$, `images[:32]` slices the first 32 images $(32, 1, 28, 28)$, and `images[:, 0, 14, 14]` extracts the center pixel across all 256 images $(256,)$:

```
images[0].shape, images[:32].shape, images[:, 0, 14, 14].shape
```

```
(torch.Size([1, 28, 28]), torch.Size([32, 1, 28, 28]), torch.Size([256]))
```

We can also index a tensor using *integer index tensors* (*gather*): passing two 1D integer tensors `rows = (0, 1, 2)` and `cols = (3, 0, 2)` selects entries `(0, 3)`, `(1, 0)`, and `(2, 2)`:

```
rows, cols = torch.arange(3), torch.tensor([3, 0, 2])
X[rows, cols]
```

```
tensor([ 3.,  4., 10.])
```

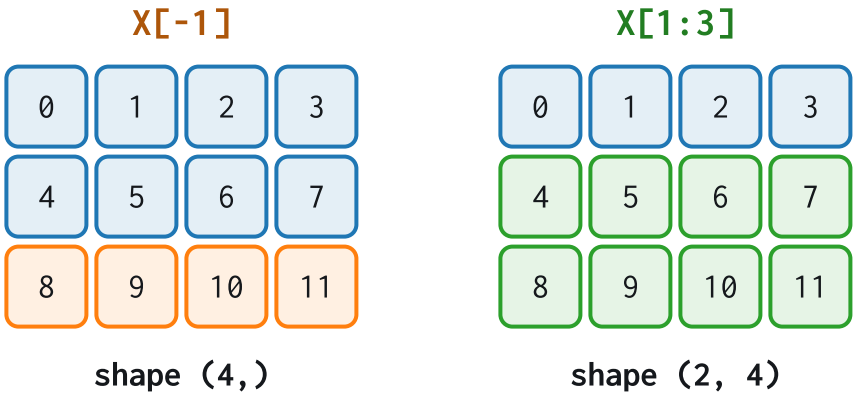


Figure 1.3: Indexing a 3×4 matrix. The integer index `X[-1]` selects the last row and removes axis 0, leaving shape $(4,)$. The slice `X[1:3]` selects rows 1 and 2 and keeps axis 0, leaving shape $(2, 4)$.

Similarly, an elementwise comparison produces a *Boolean mask* of `True`/`False` values (combinable with `&`, `|`, `~`). Indexing `X[mask]` extracts all entries where `mask` is `True` into a flattened 1D vector:

```
mask = (X > 2) & (X < 8)
X[mask]
```

```
tensor([3., 4., 5., 6., 7.])
```

When we instead want to preserve the 3×4 shape of `X`, `where(mask, X, 0)` selects from `X` wherever `mask` is `True` and fills all remaining positions with `0`:

```
torch.where(mask, X, 0)
```

```
tensor([[0., 0., 0., 3.],
        [4., 5., 6., 7.],
        [0., 0., 0., 0.]])
```

Figure 1.4 sets the three selections side by side. They differ in the shape they return: index tensors give one entry per row, a mask gives a vector whose length depends on the data, and `where` gives a tensor shaped like `X`.

1.1.2.3 Writing Elements

Whether a tensor can be overwritten in place depends on the framework's design:

PyTorch tensors are *mutable*: `X[i, j] = val` overwrites memory in place. Moreover, `reshape` returned a zero-copy *view* sharing the original buffer, so mutating `X` also mutates `x`. We assign `17` to `X[1, 2]` (flat index 6) and inspect both `X` and `x`:

```
X[1, 2] = 17
X, x
```

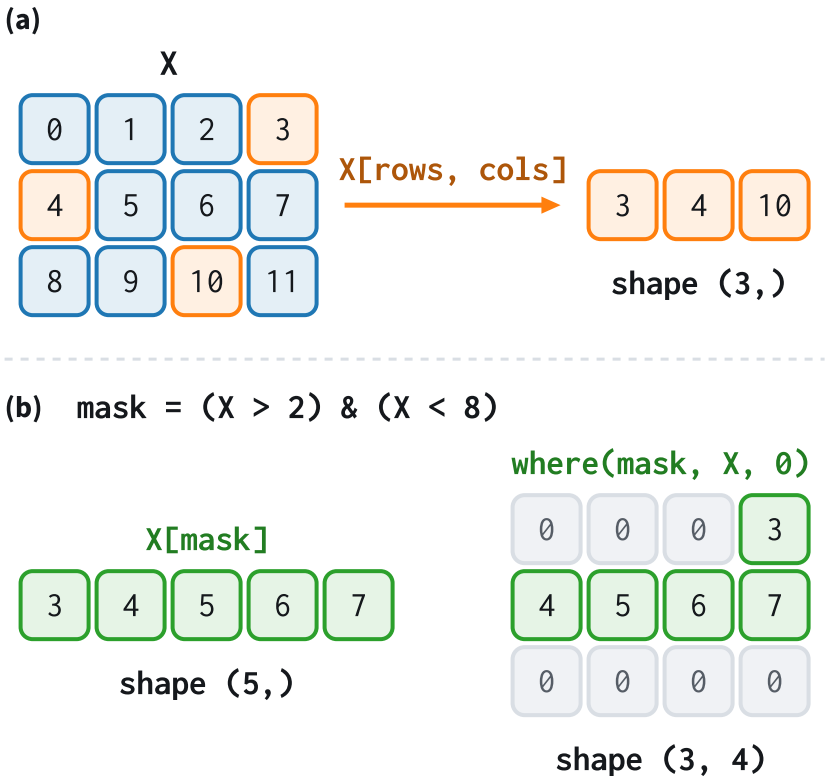


Figure 1.4: Selecting entries of the 3×4 matrix `X`. (a) The index tensors `rows = (0, 1, 2)` and `cols = (3, 0, 2)` select one entry from each row. (b) The mask `(X > 2) & (X < 8)` extracts the five entries between 2 and 8 as a vector, whereas `where` keeps the shape of `X` and writes 0 everywhere else.

```
(tensor([[ 0.,  1.,  2.,  3.],
         [ 4.,  5., 17.,  7.],
         [ 8.,  9., 10., 11.]]),
 tensor([ 0.,  1.,  2.,  3.,  4.,  5., 17.,  7.,  8.,  9., 10., 11.]))
```

In PyTorch, `x[6]` changed to `17.` because `reshape` returned a view sharing storage with `x`.

Slicing on the left-hand side assigns to an entire block at once:

```
X[:2, :] = 12
X
```

```
tensor([[12., 12., 12., 12.],
        [12., 12., 12., 12.],
        [ 8.,  9., 10., 11.]])
```

1.1.3 Operations

1.1.3.1 Elementwise Operations

Standard binary arithmetic operators (`+`, `-`, `*`, `/`, `**`) lift scalar arithmetic to act independently on every pair of corresponding elements of two identically shaped tensors `x` and `y`:

```
x = torch.tensor([1.0, 2, 4, 8])
y = torch.tensor([2.0, 2, 2, 2])
x + y, x - y, x * y, x / y, x ** y
```

```
(tensor([ 3.,  4.,  6., 10.]),
 tensor([-1.,  0.,  2.,  6.]),
 tensor([ 2.,  4.,  8., 16.]),
 tensor([0.5000, 1.0000, 2.0000, 4.0000]),
 tensor([ 1.,  4., 16., 64.]))
```

Unary mathematical functions such as `exp` likewise apply elementwise, computing e^{x_i} for each entry of `x`:

```
torch.exp(x)
```

```
tensor([2.7183e+00, 7.3891e+00, 5.4598e+01, 2.9810e+03])
```

Dividing our `uint8` minibatch `images` by `255` rescales all $200{,}704$ pixels at once and promotes the result to `float32`:

```
scaled = images / 255
print(scaled.dtype, images[0, 0, 0, :4], scaled[0, 0, 0, :4], sep='\n')
```

```
torch.float32
tensor([102, 121, 210, 214], dtype=torch.uint8)
tensor([0.4000, 0.4745, 0.8235, 0.8392])
```

Relational operators such as `x == y` also act elementwise, returning a Boolean tensor with `True` at positions where `x` and `y` agree (here only index 1, where both equal `2.0`):

```
x == y
```

```
tensor([False,  True, False, False])
```

Summing a tensor via `sum()` collapses all its entries into a single scalar ($1+2+4+8=15$ for `x`); because `True` counts as 1 and `False` as 0, summing the Boolean mask `(x == y).sum()` counts how many entries match:

```
x.sum(), (x == y).sum()
```

```
(tensor(15.), tensor(1))
```

1.1.3.2 Broadcasting

Often we combine tensors of *different* shapes, such as adding a 4-element bias vector `v` to every row of a 3×4 matrix `X`:

```
X = torch.arange(12, dtype=torch.float32).reshape(3, 4)
v = torch.tensor([100.0, 200, 300, 400])
X + v
```

```
tensor([[100., 201., 302., 403.],
        [104., 205., 306., 407.],
        [108., 209., 310., 411.]])
```

Rather than materializing physical copies, tensor libraries apply a two-step *broadcasting* rule:

1. **Align shapes from the right:** Prepend 1s on the left of the lower-order tensor's shape tuple until both shapes have the same length. Two aligned dimensions are compatible if they are equal or if one of them is 1.
2. **Stretch singleton axes:** Virtually replicate any length-1 axis to match the other tensor's length $k > 1$ without copying memory, then apply the elementwise operation.

To watch both tensors stretch simultaneously (Figure 1.5), let us construct a 3×1 column vector `a` and a 1×2 row vector `b`:

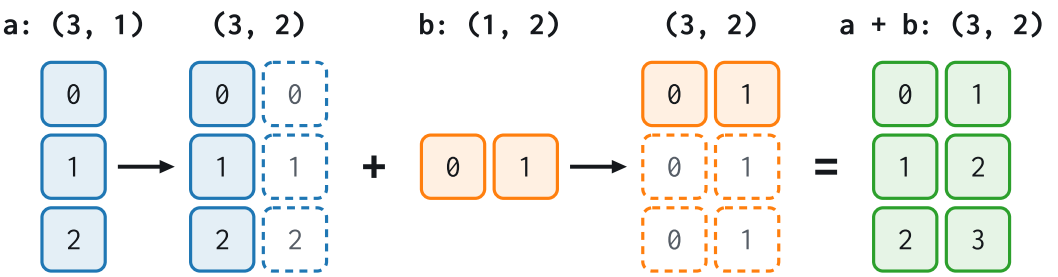


Figure 1.5: Broadcasting stretches axes of length 1. The column `a` of shape $(3, 1)$ and the row `b` of shape $(1, 2)$ are both expanded to shape $(3, 2)$ (dashed: virtual copies) before the elementwise addition.

```
a = torch.arange(3).reshape(3, 1)
b = torch.arange(2).reshape(1, 2)
a, b
```

```
(tensor([[0],
         [1],
         [2]]),
 tensor([[0, 1]]))
```

Adding `a + b` virtually replicates `a` across two columns and `b` across three rows to produce the 3×2 matrix of all pairwise sums $a_{i0} + b_{0j}$:

```
a + b
```

```
tensor([[0, 1],
        [1, 2],
        [2, 3]])
```

If any pair of aligned dimensions differs and neither is 1 (such as `(3, 2)` and `(2, 3)`), broadcasting raises an error:

```
try:
    torch.ones((3, 2)) + torch.ones((2, 3))
except Exception as e:
    print(e)
```

```
The size of tensor a (2) must match the size of tensor b (3) at non-singleton dimension 1
```

Thus, subtracting a $(28, 28)$ mean image from `images` of shape $(256, 1, 28, 28)$ broadcasts cleanly as $(1, 1, 28, 28)$, whereas adding a per-image vector of shape $(256,)$ to flattened images $(256, 784)$ fails unless reshaped to a column $(256, 1)$.

An error message is the lucky outcome. Subtracting labels of shape $(3,)$ from predictions of shape $(3, 1)$ is legal under the same rules, and it returns a 3×3 matrix of all pairwise differences where we wanted three errors (Figure 1.6). A loss averaged over that matrix is still a single number, so nothing looks wrong until the model fails to learn.

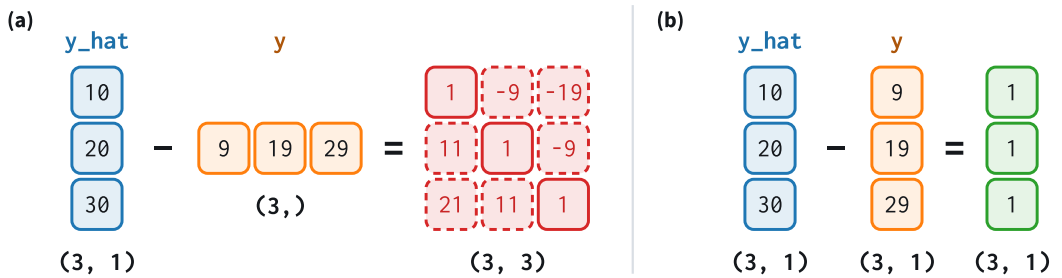


Figure 1.6: Broadcasting without an error message. (a) Predictions `y_hat` of shape $(3, 1)$ minus labels `y` of shape $(3,)$ give all nine pairwise differences; only the diagonal (solid) holds the three errors we wanted. (b) Once `y` is reshaped to $(3, 1)$, the same subtraction returns those three errors and nothing else.

1.1.3.3 Concatenation

`concatenate` (or `cat` / `concat`) joins tensors end to end along an *existing* axis (all other axes must have matching lengths): joining two 3×4 matrices `X` and `Y` along `axis=0` stacks their rows into shape `(6, 4)`, whereas `axis=1` joins their columns into shape `(3, 8)`:

```
X = torch.arange(12, dtype=torch.float32).reshape(3, 4)
Y = torch.tensor([[2.0, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
torch.cat((X, Y), dim=0), torch.cat((X, Y), dim=1)
```

```
(tensor([[ 0.,  1.,  2.,  3.],
         [ 4.,  5.,  6.,  7.],
         [ 8.,  9., 10., 11.],
         [ 2.,  1.,  4.,  3.],
         [ 1.,  2.,  3.,  4.],
         [ 4.,  3.,  2.,  1.]]),
 tensor([[ 0.,  1.,  2.,  3.,  2.,  1.,  4.,  3.],
         [ 4.,  5.,  6.,  7.,  1.,  2.,  3.,  4.],
         [ 8.,  9., 10., 11.,  4.,  3.,  2.,  1.]]))
```

By contrast, `stack` joins identically shaped tensors along a *new* axis, bundling `X` and `Y` into a 3^{rd}-order tensor of shape `(2, 3, 4)` by default (`axis=0`) or `(3, 4, 2)` when stacked along `axis=2`:

```
torch.stack((X, Y)).shape, torch.stack((X, Y), dim=2).shape
```

```
(torch.Size([2, 3, 4]), torch.Size([3, 4, 2]))
```

1.1.4 Memory and Conversion

1.1.4.1 Allocation and In-Place Updates

Standard expressions like `Y = Y + X` allocate a new buffer and rebind `Y` to the new address, as verified with Python's `id()`:

```
before = id(Y)
Y = Y + X
id(Y) == before
```

```
False
```

For models with gigabytes of parameters, unnecessary allocations waste memory bandwidth and can leave other references pointing to stale buffers (though inside forward passes, you must not overwrite activations still needed by automatic differentiation; see Section 1.5).

In MXNet and PyTorch, slice assignment `Z[:] = X + Y` writes the result into an existing preallocated buffer `Z` so `id(Z)` never changes:

```
Z = torch.zeros_like(Y)
before = id(Z)
```

```
Z[:] = X + Y
id(Z) == before
```

```
True
```

When the old value of `X` is no longer needed, augmented assignment `X += Y` asks for the sum to be written into `X` itself; `id(X)` shows whether the framework could do so:

```
before = id(X)
X += Y
id(X) == before
```

```
True
```

`id(X) == before` is `True`: `X` was updated in place.

1.1.4.2 Conversion to Other Python Objects

Converting between framework tensors and NumPy arrays is routine at the boundary with plotting or data libraries, but frameworks differ in whether conversion shares memory or copies:

On the CPU, `X.numpy()` and `torch.from_numpy(A)` **share the exact same underlying memory buffer**, so mutating one in place immediately modifies the other: after the write below, both `X[0, 0]` and `B[0, 0]` change to `100.0`.

```
A = X.numpy()
B = torch.from_numpy(A)
type(A), type(B)
```

```
(numpy.ndarray, torch.Tensor)
```

We can verify whether `A`, `B`, and `X` share memory by overwriting `A[0, 0] = 100` in the NumPy array and inspecting `X[0, 0]` and `B[0, 0]`:

```
A[0, 0] = 100  # Write to the NumPy array
X[0, 0], B[0, 0]
```

```
(tensor(100.), tensor(100.))
```

To extract a single-element (0^{th}-order) tensor into a Python `float` or `int` (for example, when logging the scalar loss), call `.item()`, `float()`, or `int()`, noting that doing so forces the CPU to synchronize with any pending GPU kernels:

```
a = torch.tensor(3.5)
a, a.item(), float(a), int(a)
```

```
(tensor(3.5000), 3.5, 3.5, 3)
```

> **Tip: Exploring Framework APIs.** Whenever you encounter an unfamiliar module or function, use `dir(obj)` (or Tab completion) to list available attributes,

`help(fn)` or `fn?` in Jupyter to inspect its signature and docstring (and `fn??` to view its Python source), and verify your understanding with a tiny test input whose expected output you can predict by hand.

1.1.5 Summary

Tensors store inputs, parameters, activations, and gradients as n-dimensional arrays characterized by a `shape` and a `dtype`. Deep learning code expresses computations over whole tensors at once, rescaling all $200{,}704$ pixels of a minibatch in a single compiled operation (`images / 255`). We reconfigure axes with `reshape`, extract or modify subsets with integer indices, slices, and Boolean masks (`where`), join tensors with `concatenate` and `stack`, and combine differently shaped tensors via *broadcasting* (aligning shapes from the right and stretching singleton axes). Finally, where tensors are mutable, in-place updates (`Z[:] = ...`, `X += Y`) reuse memory; otherwise mutable containers or the compiler take over that role.

1.1.6 Exercises

1. code **Inferred reshape axes.** Suppose a minibatch of n grayscale images of resolution 28×28 arrives as a tensor of shape $(n, 1, 28, 28)$, and each image must be flattened into a row vector of 784 pixel values. Verify each prediction below in code:
    1. Predict the output shape of reshaping the minibatch to `(-1, 784)` versus `(256, -1)`, first for a full minibatch of $n = 256$ images and then for the final, smaller minibatch of an epoch with $n = 100$ images. Which of the two target shapes works reliably for *any* batch size n?
    2. Suppose the images were accidentally loaded in 3-channel RGB format of shape $(n, 3, 28, 28)$. What happens when you reshape that tensor to `(-1, 784)`? Does `reshape(-1, 784)` catch the channel-count mistake, or does it silently scramble the batch dimension?
2. code **Broadcasting shapes and silent bugs.** Broadcasting aligns two shape tuples from the right.
    1. Construct two 3^{rd}-order tensors `a` and `b` whose shapes differ along every single axis yet are broadcast-compatible. Predict the shape of `a + b` by hand, then run the code to verify your prediction.
    2. A regression model outputs a column of predictions `y_hat` of shape $(5, 1)$ for five examples whose ground-truth targets `y` are stored as a 1D vector of shape $(5,)$. Predict the shape of the residual `y_hat - y`. Next, let `y` hold `[0.0, 1.0, 2.0, 3.0, 4.0]` and set `y_hat = y.reshape(5, 1)` so that every prediction equals its true label identically. Compute the mean of the squared entries of `y_hat - y`, explain why this "mean squared error" is nonzero even though the predictions are perfect, and show how to reshape `y` (or `y_hat`) so that each prediction is compared only against its own label.
3. code **Arithmetic limits of data types.** Raw pixels in `images` are stored as 8-bit unsigned integers (`uint8`), whereas neural network layers compute in floating point (`float32`).
    1. Predict the numerical result of adding `100` to a `uint8` tensor entry holding the value `200`, and verify your prediction in code. Explain why this wrap-around occurs, and show how to brighten the first image of `images` by 100 intensity levels (clamped or promoted) without integer overflow.

2. An IEEE 754 `float32` number allocates 24 bits of significand (mantissa) precision. Explain why `float32` can represent every consecutive integer up to $2^{24} = 16{,}777{,}216$ exactly, and verify in code that evaluating `float32(2**24) + float32(1)` leaves the number unchanged at $16{,}777{,}216$. If a training loop accumulates the total number of examples or tokens processed in a `float32` scalar counter (incrementing by 1 each time), after how many examples will the counter freeze completely, and what data type should be used instead?

4. code **In-place composition.** ● Evaluate `(A + B) * (-A / 2)` for two identically shaped `float32` tensors `A` and `B` *without allocating any intermediate tensor*, using only PyTorch's in-place methods (`add_`, `div_`, `neg_`, `mul_`) or `out=` keyword arguments. Confirm that `A.data_ptr()` and `B.data_ptr()` remain unchanged and that the final result matches the out-of-place expression `(A + B) * (-A / 2)`. Explain why writing `A[:] = A + B` would *not* satisfy the zero-temporary-allocation requirement.

   Adapted from Nicolas P. Rougier, 100 numpy exercises, problem 35.

5. code **Python loops versus broadcasting.** ● Given a 1D vector `a` of 200 elements and a 1D vector `b` of 300 elements, we want to compute the 200×300 pairwise absolute-difference matrix D with entries $d_{ij} = |a_i - b_j|$.

   1. Compute D first using two nested Python `for` loops, and then using a single broadcasted subtraction after reshaping `a` to `(200, 1)` and `b` to `(1, 300)`. Verify that both matrices agree numerically, measure the wall-clock execution time of each method, and report the speedup factor.
   2. Even though broadcasting avoids copying the inputs `a` and `b`, the output matrix D must still be materialized in memory. How many megabytes does D occupy in `float32` if `a` and `b` each contain $10{,}000$ elements? What lesson does this teach about the peak memory footprint of broadcasted outer operations?

1.2 Data Preprocessing

So far, we have been working with synthetic data that arrived in ready-made tensors of whatever shape and data type we asked for. To apply deep learning in the wild, however, we must extract messy data stored in arbitrary file formats and preprocess it into clean numerical tensors. Turning a raw table of records into model-ready tensors involves more statistical judgment than the humble word *preprocessing* suggests.

Consider our running example for this section: a small table of ten residential home sales recording the number of rooms (`NumRooms`), roof material (`RoofType`), floor area in square feet (`Area`), and sale price in dollars (`Price`). Small as it is, this table exhibits the three obstacles that prevent feeding raw tabular data straight into a neural network:

1. **Missing values:** three of the ten homes are missing a room count, and four are missing a roof type.
2. **Categorical text fields:** the roof type is recorded as a word (`Slate` or `Tile`), whereas neural networks compute exclusively with numbers.
3. **Incommensurate scales:** floor area in square feet is roughly 500 times larger numerically than the room count.

Each obstacle demands a statistical choice: discarding incomplete rows would throw away most of the table, encoding unordered roof types as `0, 1, 2` would invent a false ordering, and leaving square footage unscaled would let it dominate distances and gradients. The

pandas library (McKinney 2010) is Python's standard tool for tabular data; below we walk through the preprocessing pipeline in Figure 1.7, taking a raw CSV file to `float32` feature and target tensors.

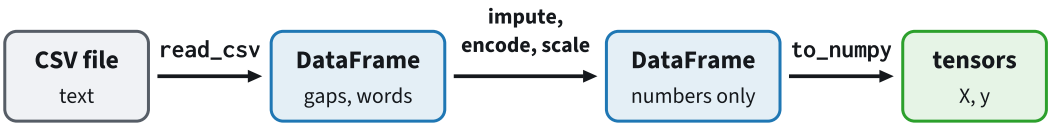


Figure 1.7: From a file to tensors. `read_csv` loads the text file into a DataFrame with gaps and text columns; imputation, encoding, and scaling make every entry a number; `to_numpy` hands the result to the framework as the tensors `X` and `y`.

We begin by importing `pandas` under its customary alias `pd`, Python's `os` module for file-path handling, and our deep learning framework (needed at the final step when we convert the cleaned table into tensors).

```
import os
import pandas as pd
import torch
```

1.2.1 Loading and Inspecting

1.2.1.1 CSV Files

Comma-separated values (CSV) files store tabular data in plain text, with column headers on the first line and one comma-delimited record per subsequent line. We write our ten-home dataset to `../data/house_tiny.csv`, where empty fields between consecutive commas (such as `,Tile,2100,290000`) denote unrecorded values:

```
os.makedirs(os.path.join('..', 'data'), exist_ok=True)
data_file = os.path.join('..', 'data', 'house_tiny.csv')
with open(data_file, 'w') as f:
    f.write('''NumRooms,RoofType,Area,Price
3,Slate,1500,210000
,Tile,2100,290000
2,,850,127500
4,Slate,1940,258000
,,1200,168000
3,Tile,1650,225000
5,Slate,2600,375000
2,,900,142000
,Tile,1750,240000
4,,2050,295000''')
```

Calling `pd.read_csv` parses the file into a pandas *DataFrame*, a two-dimensional table of named, typed columns with a row index (0 through 9) along the left margin:

```
data = pd.read_csv(data_file)
data
```

```
   NumRooms RoofType  Area   Price
0       3.0    Slate  1500  210000
1       NaN     Tile  2100  290000
2       2.0      NaN   850  127500
3       4.0    Slate  1940  258000
4       NaN      NaN  1200  168000
5       3.0     Tile  1650  225000
6       5.0    Slate  2600  375000
7       2.0      NaN   900  142000
8       NaN     Tile  1750  240000
9       4.0      NaN  2050  295000
```

Notice that `pandas` automatically replaced every empty field (as well as standard tokens like `NA`) with `NaN` (*not a number*), its sentinel for a *missing value*.

1.2.1.2 Data Types and Summaries

Before modifying a newly loaded dataset, inspect its row and column counts, non-missing entries, and inferred types with `data.info()`:

```
data.info()
```

```
<class 'pandas.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 4 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   NumRooms  7 non-null      float64
 1   RoofType  6 non-null      str
 2   Area      10 non-null     int64
 3   Price     10 non-null     int64
dtypes: float64(1), int64(2), str(1)
memory usage: 481.0 bytes
```

Of the 10 rows, `Area` and `Price` are complete (`int64`), `NumRooms` has 7 observed entries (`float64`), and `RoofType` has 6 (`str`, or `object` in older pandas releases). Notice that `NumRooms` counts whole rooms yet was promoted to `float64`: under IEEE 754, `NaN` is a floating-point bit pattern, so standard integer arrays cannot represent missing entries.

For the numerical columns, `data.describe()` reports the count, mean, sample standard deviation (`std`), minimum, quartiles, and maximum across observed entries:

```
data.describe()
```

```
       NumRooms         Area          Price
count  7.000000    10.000000      10.000000
mean   3.285714  1654.000000  233050.000000
std    1.112697   556.421104   76128.710455
min    2.000000   850.000000  127500.000000
25%    2.500000  1275.000000  178500.000000
50%    3.000000  1700.000000  232500.000000
75%    4.000000  2022.500000  282000.000000
max    5.000000  2600.000000  375000.000000
```

NumRooms ranges from 2 to 5 (mean 3.29), whereas Area ranges from 850 to 2,600 square feet (mean 1,654) and Price from \$127,500 to \$375,000.

1.2.1.3 Separating Inputs and Targets

In supervised learning, we predict a designated *target* (or *label*) column, here Price, from the remaining *input* (*feature*) columns (NumRooms, RoofType, Area). Separating features from the target up front prevents subsequent feature transformations from touching or leaking the target. Using 0-based positional slicing via .iloc, data.iloc[:, :3] selects the first three columns and data.iloc[:, 3] selects the fourth:

```
inputs, targets = data.iloc[:, :3], data.iloc[:, 3]
list(inputs.columns), targets.name
```

```
(['NumRooms', 'RoofType', 'Area'], 'Price')
```

Positional slicing is compact, but in production pipelines an inserted or reordered CSV column would cause data.iloc[:, 3] to grab the wrong target silently; selecting by name (targets = data['Price'] and inputs = data.drop(columns='Price')) avoids this risk (Exercise 1).

1.2.2 Cleaning and Encoding

1.2.2.1 Missing Values

Missing values arise routinely when sensors drop out, survey respondents skip questions, or historical records go unrecorded. Chaining .isna() with .sum() counts the missing entries in each column (True counts as 1):

```
inputs.isna().sum()
```

```
NumRooms    3
RoofType    4
Area        0
dtype: int64
```

Three strategies are standard, each with a distinct statistical trade-off (Figure 1.8):

- **Deletion** discards every row (or column) containing a missing value. It is effortless to implement, but it sacrifices data and biases the surviving sample whenever values are not missing completely at random (for instance, if studio apartments disproportionately omit room counts).
- **Imputation** replaces each NaN with an estimate such as the column mean or median, preserving every row at the cost of substituting approximations that reduce the column's apparent spread.
- **Missingness indicators** pair imputation with an extra binary $(0/1)$ column recording *whether* each entry was originally missing, allowing the model to exploit informative missingness (for example, if unreported room counts occur mainly on luxury estates) at the cost of one extra feature per imputed column.

On our ten-home table, row deletion (inputs.dropna()) leaves very little data:

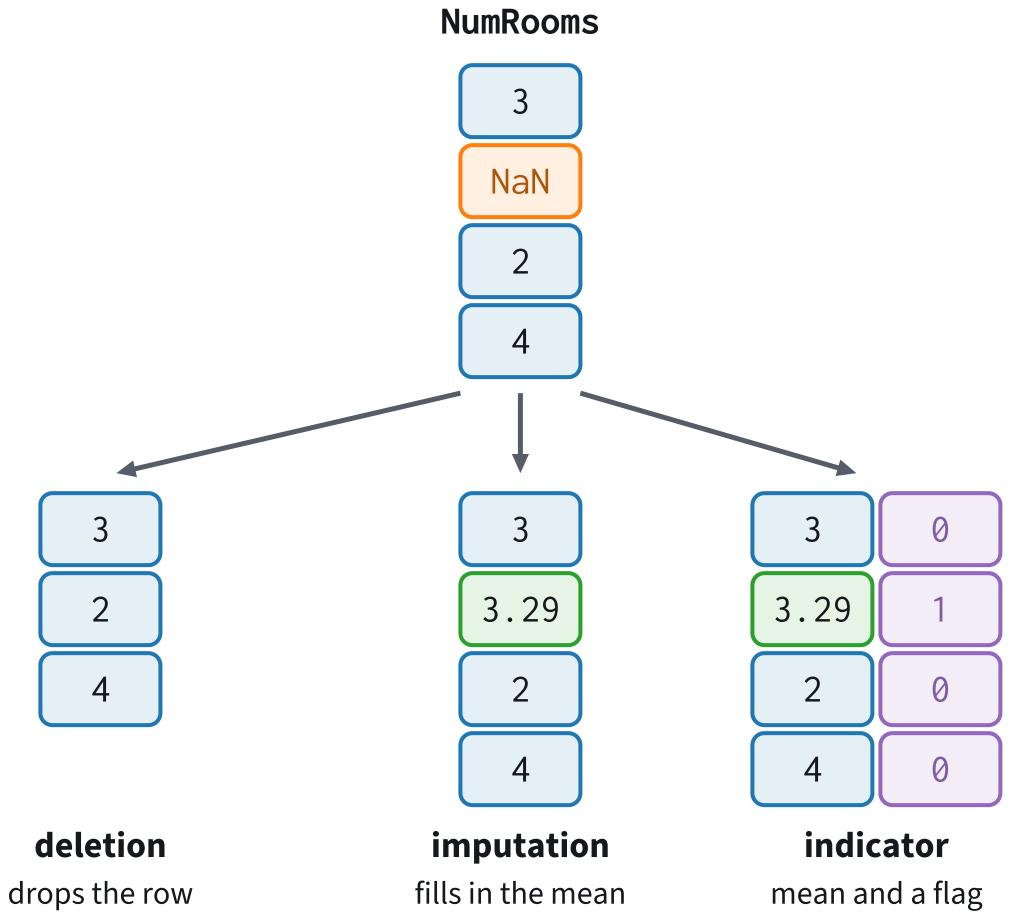


Figure 1.8: Three treatments of a missing room count, on the first four rows of `NumRooms`. Deletion drops the row. Imputation fills in the mean of the seven observed counts, $23/7 \approx 3.29$. A missingness indicator fills in the same mean and adds a column that records where the gap was.

```
len(inputs.dropna()), len(inputs)
```

```
(4, 10)
```

Dropping incomplete rows discards 6 of our 10 homes (60% of the dataset), so we impute instead. Calling `inputs.fillna(inputs.mean(numeric_only=True))` replaces each numerical `NaN` with the mean of that column's observed entries (the median is preferable when outliers skew the mean), while leaving the text column `RoofType` for the categorical step next:

```
inputs = inputs.fillna(inputs.mean(numeric_only=True))
inputs
```

```
   NumRooms RoofType  Area
0  3.000000    Slate  1500
1  3.285714     Tile  2100
2  2.000000      NaN   850
3  4.000000    Slate  1940
4  3.285714      NaN  1200
5  3.000000     Tile  1650
6  5.000000    Slate  2600
7  2.000000      NaN   900
8  3.285714     Tile  1750
9  4.000000      NaN  2050
```

Rows 1, 4, and 8 now hold $23/7 \approx 3.285714$, the average of the seven observed room counts. No physical house has 3.29 rooms; the imputed mean serves as a neutral statistical placeholder.

1.2.2.2 Categorical Features

Including missing entries as a distinct state, `RoofType` falls into three unordered categories: `Slate`, `Tile`, or `NaN`. Assigning single integer codes such as Slate $= 0$, Tile $= 1$, and missing $= 2$ would force any linear combination of features to treat `Tile` as halfway between `Slate` and an unrecorded roof, imposing an artificial metric and ordering.

One-hot encoding (*dummy variable encoding*) avoids this trap by replacing a k-category column with k binary $(0/1)$ indicator columns, setting the active category's entry to 1.0 and the other $k - 1$ entries to 0.0. In `pandas`, `pd.get_dummies` encodes all string/categorical columns while leaving numerical columns intact; `dummy_na=True` adds an explicit `RoofType_nan` missingness indicator, and `dtype=float` stores $0.0/1.0$ floats rather than Booleans:

```
inputs = pd.get_dummies(inputs, dummy_na=True, dtype=float)
inputs
```

```
   NumRooms  Area  RoofType_Slate  RoofType_Tile  RoofType_nan
0  3.000000  1500             1.0            0.0           0.0
1  3.285714  2100             0.0            1.0           0.0
2  2.000000   850             0.0            0.0           1.0
3  4.000000  1940             1.0            0.0           0.0
4  3.285714  1200             0.0            0.0           1.0
5  3.000000  1650             0.0            1.0           0.0
6  5.000000  2600             1.0            0.0           0.0
```

```
7  2.000000   900            0.0           0.0           1.0
8  3.285714  1750            0.0           1.0           0.0
9  4.000000  2050            0.0           0.0           1.0
```

In every row, `RoofType_Slate`, `RoofType_Tile`, and `RoofType_nan` sum to 1.0 (making one column redundant alongside an intercept bias; likewise, a two-state column with no missing values needs only one $0/1$ indicator). One-hot encoding suits modest category counts, whereas columns with tens of thousands of categories (such as postal codes, user IDs, or vocabulary words) are mapped instead to compact learnable *embeddings* (Section 8.4).

1.2.3 Scaling and Converting

1.2.3.1 Standardization

Although every column in `inputs` is now numerical, `NumRooms` and `Area` live on incompatible measurement scales: in raw units, a difference of 2 rooms is numerically smaller than 3 square feet. Unscaled square footage would dominate Euclidean distances and weight penalties, and as we will see in Section 2.3 and Chapter 9, multi-order scale gaps stretch the loss surface into narrow valleys that slow gradient descent.

Standardization (*z-score normalization*) puts continuous features on a common scale by mapping each entry x to

$$\frac{x - \mu}{\sigma}, \tag{1.1}$$

where μ is the column's sample mean and $\sigma = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \mu)^2}$ is its sample standard deviation (Bessel's $n - 1$ denominator is derived in Section 1.6). Every standardized column has mean 0 and standard deviation 1, measuring observations in unitless standard deviations regardless of whether area was recorded in square feet or square meters. Because our one-hot indicators already lie in $\{0, 1\}$, we standardize only `['NumRooms', 'Area']`:

```
numerical = ['NumRooms', 'Area']
mu, sigma = inputs[numerical].mean(), inputs[numerical].std()
inputs[numerical] = (inputs[numerical] - mu) / sigma
inputs
```

```
   NumRooms      Area  RoofType_Slate  RoofType_Tile  RoofType_nan
0 -0.314485 -0.276769             1.0            0.0           0.0
1  0.000000  0.801551             0.0            1.0           0.0
2 -1.415185 -1.444949             0.0            0.0           1.0
3  0.786214  0.513999             1.0            0.0           0.0
4  0.000000 -0.815929             0.0            0.0           1.0
5 -0.314485 -0.007189             0.0            1.0           0.0
6  1.886913  1.700151             1.0            0.0           0.0
7 -1.415185 -1.355089             0.0            0.0           1.0
8  0.000000  0.172531             0.0            1.0           0.0
9  0.786214  0.711691             0.0            0.0           1.0
```

Row 6 (5 rooms, $2{,}600$ square feet) now reads $(1.89, 1.70)$, about 1.7 to 1.9 standard deviations above average on both dimensions, while the three mean-imputed room counts in rows 1, 4, and 8 sit at 0.0. Notice the side effect shown in Figure 1.9: placing three imputed

rows at μ adds zero to the numerator of σ while raising n from 7 to 10, shrinking the room-count standard deviation from 1.11 to 0.91 and inflating the observed z-scores (from 1.54 to 1.89 on row 6).

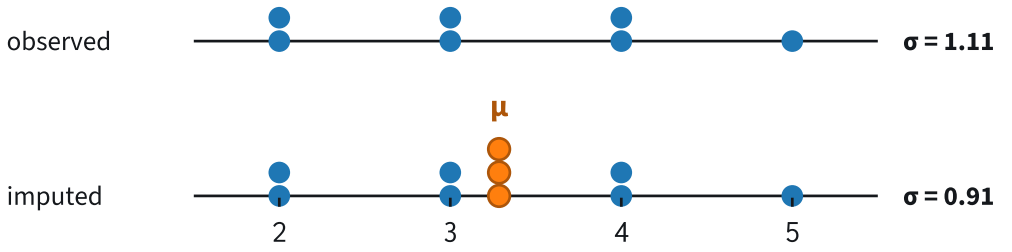


Figure 1.9: The room counts before and after mean imputation. The three imputed counts (orange) all land on the mean μ of the seven observed ones, so the column is less spread out than the data it came from: its standard deviation falls from 1.11 to 0.91.

One methodological rule is paramount whenever a dataset is split into a *training set* and a *test set*: **all preprocessing statistics, including imputation means or medians, category vocabularies, and standardization (μ, σ), must be fitted exclusively on the training set** and applied unchanged to the test set (Figure 1.10). Fitting statistics on the combined dataset leaks information from the held-out test split into training (*data leakage*) and inflates test scores; we enforce this rule on a full real-estate benchmark in Section 4.7 (see also Exercise 4).

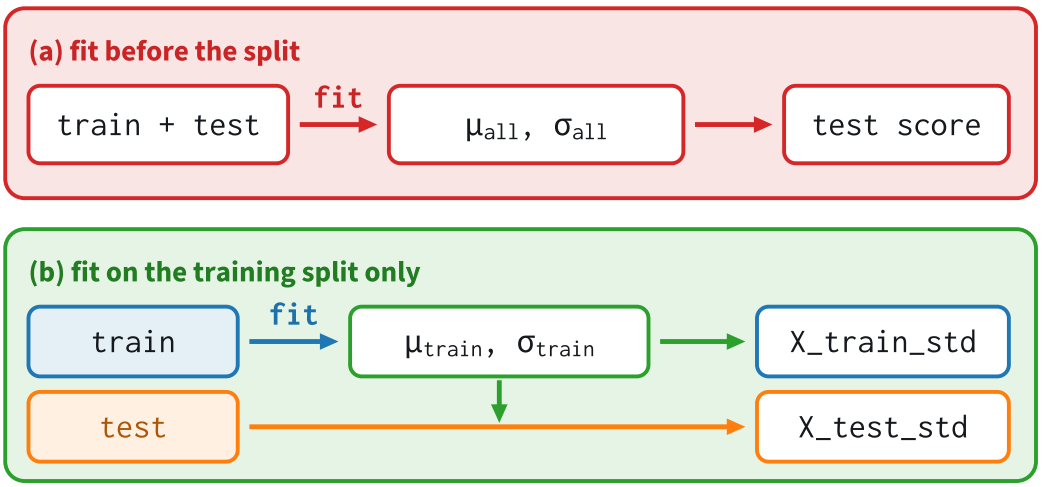


Figure 1.10: The order of fitting and splitting. (a) Statistics fitted on all rows have already seen the test set, so the test score comes out too optimistic. (b) Statistics fitted on the training split alone are applied unchanged to both splits, and the test set stays unseen.

1.2.3.2 Conversion to Tensors

Finally, `.to_numpy(dtype='float32')` extracts the cleaned columns as contiguous 32-bit floating-point NumPy arrays (downcasting from pandas' default `float64`), ready for our framework's tensor constructor (Section 1.1):

```
X = torch.tensor(inputs.to_numpy(dtype='float32'))
y = torch.tensor(targets.to_numpy(dtype='float32'))
X.shape, X.dtype, y.shape, y.dtype
```

```
(torch.Size([10, 5]), torch.float32, torch.Size([10]), torch.float32)
```

The feature matrix `X` has shape `(10, 5)` (10 homes by 2 standardized numerical features plus 3 roof-type indicators) and the target vector `y` has shape `(10,)`, both in `float32`. Production pipelines often join multiple relational tables and screen for corrupted records or sentinel codes (such as `-9999` for an unknown coordinate) before reaching this stage; once assembled into a design matrix $\mathbf{X}$, data flow into the linear-algebraic operations of the next section.

1.2.4 Summary

Real-world tabular datasets almost always arrive with missing values, categorical text fields, and numerical columns measured on wildly different scales, whereas neural networks require clean `float32` tensors. Using `pandas`, we load CSV files into a DataFrame with `read_csv`, inspect column types, missing-value counts, and value distributions with `info()` and `describe()`, and separate the target column from the input features right at the outset. Missing values can be handled via deletion, imputation (such as filling numerical gaps with the training-set mean or median), or missingness indicator columns. Unordered categorical columns are converted into binary $(0/1)$ indicator columns via one-hot encoding (`get_dummies`) to avoid fabricating a numerical order, and continuous numerical columns are standardized to zero mean and unit variance ($x \mapsto (x - \mu)/\sigma$) using statistics computed solely on the training split. Finally, `.to_numpy(dtype='float32')` bridges the cleaned DataFrame directly to deep learning framework tensors.

1.2.5 Exercises

1. code **Selecting columns by name versus position.** In this section, we split `data` into features and targets by integer position using `data.iloc[:, :3]` and `data.iloc[:, 3]`. Suppose an updated version of `house_tiny.csv` inserts a new column `YearBuilt` between `Area` and `Price`.
    1. What will the positional split `data.iloc[:, :3]` and `data.iloc[:, 3]` place into `inputs` and `targets` on the updated file? Will any subsequent line of code in this section raise an exception, or will the pipeline train a model to predict the wrong target silently?
    2. Rewrite the feature/target split using column names (for example, via `data['Price']` and `data.drop(...)`) so that `targets` always extracts `Price` and `inputs` retains all other columns, regardless of column order or newly added features. Consult the pandas user guide on indexing and selecting data for details.
2. code **Comparing imputation strategies.** Impute the missing entries in `NumRooms` under three different schemes: (i) mean imputation, (ii) median imputation, and (iii) mean imputation augmented with a binary `NumRooms_isna` indicator column.

1. For both mean imputation and median imputation, compute the sample mean and sample standard deviation of the filled `NumRooms` column, compare them against the mean and standard deviation of the 7 originally observed values, and explain why the standard deviation changes in the direction it does.
2. Suppose that in a larger real-estate dataset, `NumRooms` goes unreported primarily for luxury mansions. What room count does a model see for those mansions under mean imputation alone? Explain how a linear model (which predicts price as a weighted sum of the input columns plus a bias) can use the extra `NumRooms_isna` indicator column to compensate for that systematic understatement.
3. State clearly what statistical assumption each of the three treatments (deletion, mean imputation alone, and imputation with an indicator) makes about the mechanism causing values to go missing.

Adapted from Kaggle Learn, Intermediate Machine Learning, Exercise 2.

3. **High-cardinality categorical features.** One-hot encoding expands a categorical column with k distinct categories into k binary columns (or $k+1$ with `dummy_na=True`).

   1. Suppose a table of $n = 10^6$ homes contains a `ZipCode` column with $k = 10^4$ distinct postal codes and no missing values. Calculate how many gigabytes the resulting one-hot encoded matrix occupies if stored densely in 64-bit floating point (`float64`), and state what fraction of its entries are nonzero.
   2. Suppose the table also includes a `ListingID` column containing a unique identifier string for every single row. Compare three possible ways of handling `ListingID`: (i) one-hot encoding, (ii) mapping each ID to a learnable embedding vector, and (iii) dropping the column entirely. Under each treatment, what can a model learn from `ListingID` that generalizes to a newly listed house whose `ListingID` never appeared in the training set? Which treatment should you choose?

   Adapted from Kaggle Learn, Intermediate Machine Learning, Exercise 3.

4. code **Quantifying data leakage.** In the main text, we imputed missing `NumRooms` entries and standardized `NumRooms` and `Area` using the sample mean and standard deviation computed across all 10 rows. Now partition `data` into a training set consisting of the first 8 rows (`data.iloc[:8]`) and a test set consisting of the final 2 rows (`data.iloc[8:]`).

   1. Re-run the preprocessing pipeline so that the imputation mean and the standardization parameters (μ and σ) are computed *exclusively on the* 8 *training rows*, and then use those training parameters unchanged to impute and standardize both the training split and the test split. Compare the resulting standardized test rows against the values obtained when μ and σ were computed from all 10 rows.
   2. Explain precisely what information about the held-out test homes leaked into the training features under the whole-table recipe.
   3. Suppose a categorical or numerical feature happens to take a constant value across all training rows (so its training standard deviation is $\sigma = 0$), but varies on the test rows. What does the training-only standardization formula $(x-\mu)/\sigma$ produce on the training and test splits, and how should a robust preprocessing pipeline handle zero-variance features?

1.3 Linear Algebra

Dividing an image minibatch by 255 rescales each pixel in isolation, whereas a predictive model must combine hundreds or thousands of input features into a decision. *Linear algebra* is the mathematical language of such weighted combinations, allowing us to express the forward pass of an entire layer across a minibatch in one compact equation and read off its output shape and floating-point cost ahead of time.

Consider a linear image classifier sorting 28×28 grayscale clothing images (Chapter 3) into 10 categories. Flattening each photograph yields $28 \times 28 = 784$ pixels. Stacking a minibatch of 256 flattened images as the rows of a data matrix $\mathbf{X} \in \mathbb{R}^{256 \times 784}$, arranging the 784 weights for each of the 10 categories as the columns of $\mathbf{W} \in \mathbb{R}^{784 \times 10}$, and collecting the 10 category offsets into $\mathbf{b} \in \mathbb{R}^{10}$ ($7{,}850$ parameters in all), all $256 \times 10 = 2{,}560$ category scores across the minibatch are given by

$$\mathbf{O} = \mathbf{X}\mathbf{W} + \mathbf{b}, \tag{1.2}$$

where $\mathbf{b}$ is added to every row via broadcasting (Figure 1.11).

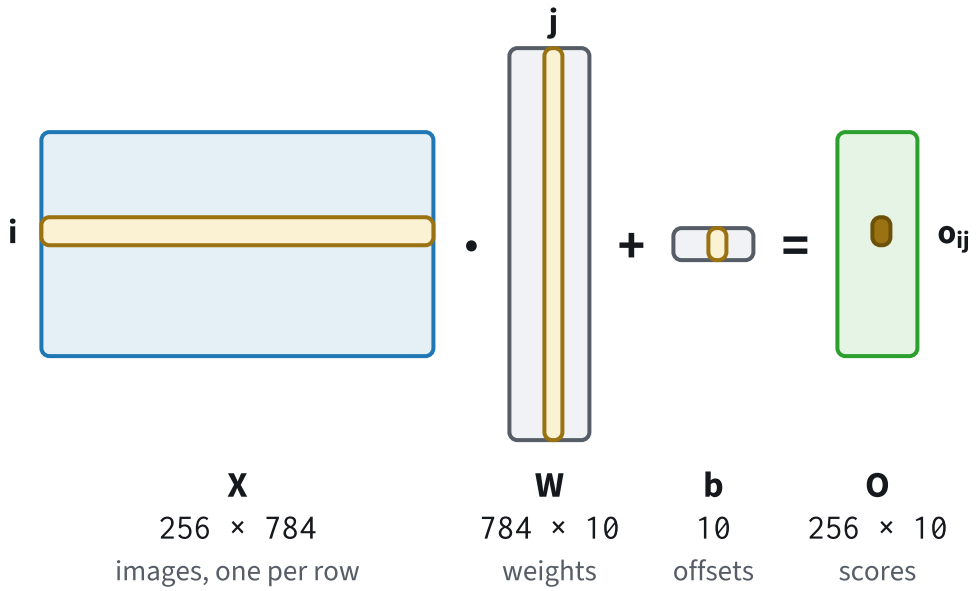


Figure 1.11: The scores of a linear classifier for a minibatch of 256 images and 10 categories. Row i of $\mathbf{X}$ holds image i, column j of $\mathbf{W}$ holds the weights of category j, and the two meet in the score o_{ij}.

In this section, we build up to $\mathbf{O} = \mathbf{X}\mathbf{W} + \mathbf{b}$ starting from scalars, vectors, matrices, and tensors, through reductions, dot products, and matrix products, and conclude with *norms* and *eigenvalues* (notational conventions are summarized in Notation).

```
import torch
```

1.3.1 Scalars, Vectors, Matrices, and Tensors

We already created and reshaped 0^{th}-, 1^{st}-, 2^{nd}-, and 4^{th}-order arrays in Section 1.1; here we pair each order (`ndim` in code) with its standard mathematical name and notation.

1.3.1.1 Scalars

Single numbers are called *scalars*, denoted by ordinary lowercase italic letters (x, y, z). Writing $x \in \mathbb{R}$ states that x is a real number. In code, a scalar is a 0^{th}-order tensor with empty shape `()`:

```
x = torch.tensor(3.0)
x, x.shape
```

```
(tensor(3.), torch.Size([]))
```

1.3.1.2 Vectors

A *vector* is a fixed-length 1D array of scalars (its *elements* or *components*), denoted by bold lowercase letters ($\mathbf{x}, \mathbf{y}, \mathbf{w}$). We write $\mathbf{x} \in \mathbb{R}^n$ for an n-dimensional real vector, treated by default as a *column vector*:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}, \tag{1.3}$$

or written inline as $\mathbf{x} = (x_1, \dots, x_n)$. In code, a vector is a 1^{st}-order tensor:

```
x = torch.arange(3, dtype=torch.float32)
x
```

```
tensor([0., 1., 2.])
```

Whereas mathematical notation uses 1-based indexing ($x_1, \dots, x_n$), Python uses 0-based indexing, so the third component x_3 is accessed via `x[2]`:

```
x[2]
```

```
tensor(2.)
```

A vector's *dimensionality* n (its number of components) is returned by Python's built-in `len(x)` or as the sole entry in its 1-tuple shape `x.shape`:

```
len(x), x.shape
```

```
(3, torch.Size([3]))
```

To prevent confusion when "dimension" is used informally, we reserve *order* (`ndim`) for the number of axes of a tensor and *dimensionality* for the number of components of a vector.

1.3.1.3 Matrices

Matrices are 2^{nd}-order tensors, denoted by bold uppercase letters ($\mathbf{A}, \mathbf{X}, \mathbf{W}$). Writing $\mathbf{A} \in \mathbb{R}^{m \times n}$ indicates m rows and n columns (when $m = n$, $\mathbf{A}$ is *square*), with entry a_{ij} in row i and column j:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}. \tag{1.4}$$

Tabular and flattened minibatch matrices $\mathbf{X} \in \mathbb{R}^{n \times d}$ store one example per row and one feature per column. Below, we construct a 2×3 matrix `A`:

```
A = torch.arange(6, dtype=torch.float32).reshape(2, 3)
A
```

```
tensor([[0., 1., 2.],
        [3., 4., 5.]])
```

Exchanging a matrix's rows and columns yields its *transpose* $\mathbf{A}^\top \in \mathbb{R}^{n \times m}$, with $[\mathbf{A}^\top]_{ij} = a_{ji}$, accessed in code via `.T`:

```
A.T
```

```
tensor([[0., 3.],
        [1., 4.],
        [2., 5.]])
```

A square matrix is *symmetric* if $\mathbf{S} = \mathbf{S}^\top$ ($s_{ij} = s_{ji}$ for all i, j):

```
S = torch.tensor([[2.0, 1.0], [1.0, 2.0]])
(S == S.T).all()
```

```
tensor(True)
```

1.3.1.4 Higher-Order Tensors

Tensors with three or more axes are denoted by sans-serif uppercase letters (`X`, `Y`, `Z`). Below, we generate a synthetic 4D image minibatch `images` of shape $(256, 1, 28, 28)$ with entries in $[0, 1]$ and flatten each image to form the 2D matrix $\mathbf{X} \in \mathbb{R}^{256 \times 784}$:

```
images = torch.rand(256, 1, 28, 28)
X = images.reshape(256, -1)  # One row of 784 pixels per image
images.shape, X.shape
```

```
(torch.Size([256, 1, 28, 28]), torch.Size([256, 784]))
```

1.3.2 Elementwise Operations and Reductions

1.3.2.1 Elementwise Operations and the Hadamard Product

The elementwise product of two identically shaped matrices $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{m \times n}$ is their *Hadamard product* $\mathbf{A} \odot \mathbf{B}$, with entries $[\mathbf{A} \odot \mathbf{B}]_{ij} = a_{ij} b_{ij}$, computed in code with `*`:

```
A * A
```

```
tensor([[ 0.,  1.,  4.],
        [ 9., 16., 25.]])
```

1.3.2.2 Reductions Along Axes

Passing `axis=0` to `.sum()` sums over rows to produce a vector of column sums ($\sum_{i=1}^{m} a_{ij}$), whereas `axis=1` sums over columns to produce a vector of row sums ($\sum_{j=1}^{n} a_{ij}$); omitting `axis` sums all entries (Figure 1.12):

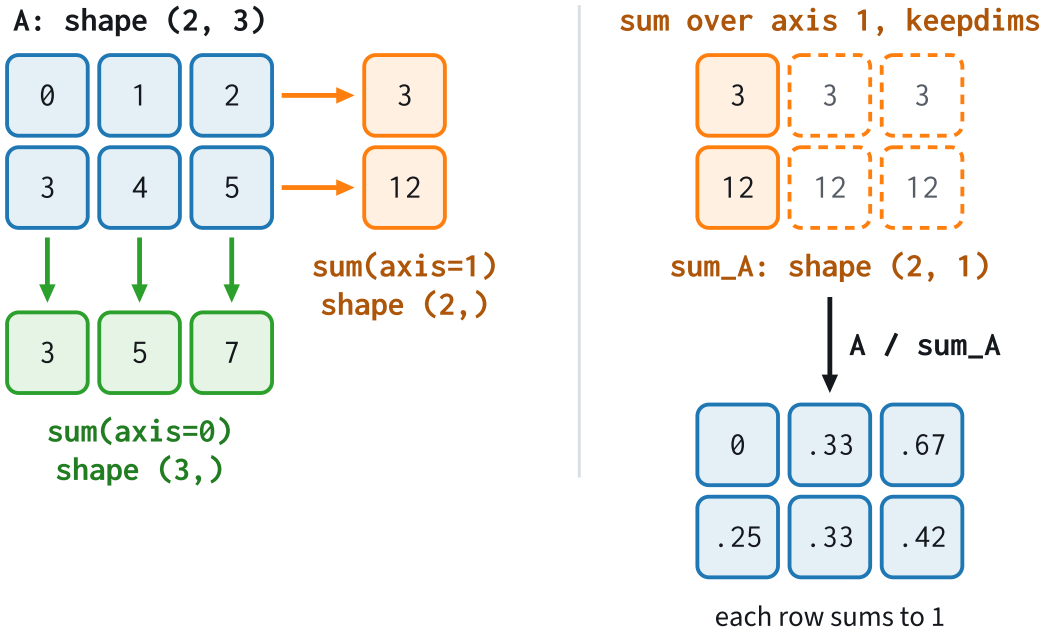


Figure 1.12: Reductions of a 2×3 matrix. Summing over axis 0 leaves one sum per column, and summing over axis 1 leaves one sum per row; the reduced axis disappears from the shape. With `keepdims=True`, the row sums keep the shape $(2, 1)$, so broadcasting stretches them across the three columns and the division `A / sum_A` scales every row to sum to 1.

```
A.sum(), A.sum(axis=0), A.sum(axis=1)
```

```
(tensor(15.), tensor([3., 5., 7.]), tensor([ 3., 12.]))
```

1.3.2.3 Keeping the Reduced Axis for Broadcasting

To normalize each row of `A` (shape `(2, 3)`) by its row sum, `A.sum(axis=1)` (shape `(2,)`) cannot broadcast against `(2, 3)` because trailing dimensions 2 and 3 clash. Passing `keepdims=True` retains the reduced axis with length 1 so `sum_A` has shape `(2, 1)`:

```
sum_A = A.sum(axis=1, keepdims=True)
sum_A, sum_A.shape
```

```
(tensor([[ 3.],
         [12.]]),
 torch.Size([2, 1]))
```

PyTorch accepts both `dim`/`keepdim` and the NumPy aliases `axis`/`keepdims`.

Because `sum_A` has shape `(2, 1)`, dividing `A / sum_A` broadcasts each row sum across all three columns so that every row of the resulting 2×3 matrix sums to 1:

```
A / sum_A
```

```
tensor([[0.0000, 0.3333, 0.6667],
        [0.2500, 0.3333, 0.4167]])
```

On our 256×784 minibatch $\mathbf{X}$, subtracting per-pixel column means (`axis=0`, shape `(784,)`) broadcasts directly, whereas subtracting per-image row means (`axis=1`) requires `keepdims=True` (shape `(256, 1)`):

```
col_means = X.mean(axis=0)
row_means = X.mean(axis=1, keepdims=True)
col_means.shape, row_means.shape, (X - row_means).shape
```

```
(torch.Size([784]), torch.Size([256, 1]), torch.Size([256, 784]))
```

Beware: if $\mathbf{X}$ happens to be square (784×784), omitting `keepdims=True` in `X - X.mean(axis=1)` raises no error and silently subtracts image j's mean from column j of every image (Exercise 2).

1.3.3 Products

1.3.3.1 Dot Products

Given two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$, their *dot product* (or *inner product*) $\mathbf{x}^\top \mathbf{y}$ (also written $\langle \mathbf{x}, \mathbf{y} \rangle$) sums the products of corresponding entries:

$$\mathbf{x}^\top \mathbf{y} = \sum_{i=1}^{d} x_i y_i. \tag{1.5}$$

In code, we compute the dot product of two 1D vectors `x` and `y` directly with the framework's `dot` function:

```
y = torch.ones(3)
x, y, torch.dot(x, y)
```

```
(tensor([0., 1., 2.]), tensor([1., 1., 1.]), tensor(3.))
```

Equivalently, we can multiply `x * y` elementwise and sum the resulting products with `sum` (returning $\sum_i x_i = 0 + 1 + 2 = 3$ when $\mathbf{y} = \mathbf{1}$):

```
torch.sum(x * y)
```

```
tensor(3.)
```

1.3.3.2 Matrix–Vector Products

Partitioning $\mathbf{A} \in \mathbb{R}^{m \times n}$ into its m row vectors $\mathbf{a}_1^\top, \ldots, \mathbf{a}_m^\top \in \mathbb{R}^n$, the *matrix–vector product* $\mathbf{Ax}$ with $\mathbf{x} \in \mathbb{R}^n$ is the m-vector of row dot products (Figure 1.13):

$$\mathbf{Ax} = \begin{bmatrix} \mathbf{a}_1^\top \\ \vdots \\ \mathbf{a}_m^\top \end{bmatrix} \mathbf{x} = \begin{bmatrix} \mathbf{a}_1^\top \mathbf{x} \\ \vdots \\ \mathbf{a}_m^\top \mathbf{x} \end{bmatrix} \in \mathbb{R}^m.$$

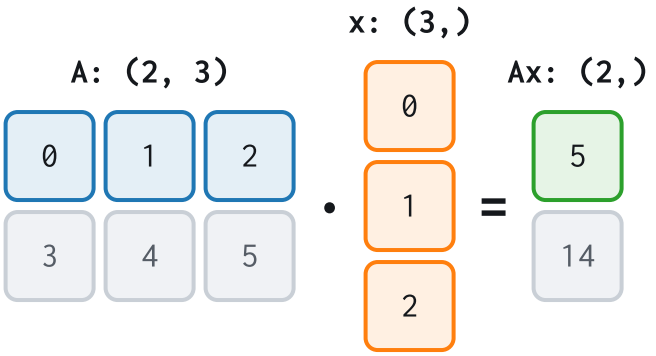


Figure 1.13: The matrix–vector product $\mathbf{Ax}$ takes one dot product per row of $\mathbf{A}$, so a 2×3 matrix maps a vector of length 3 to a vector of length 2. The first row gives $0 \cdot 0 + 1 \cdot 1 + 2 \cdot 2 = 5$.

In Python, the infix operator `@` computes both matrix–vector and matrix–matrix products:

```
A.shape, x.shape, A @ x
```

```
(torch.Size([2, 3]), torch.Size([3]), tensor([ 5., 14.]))
```

Viewed column by column, $\mathbf{Ax} = \sum_{j=1}^{n} x_j (\mathbf{A}\mathbf{e}_j)$ is a linear combination of the columns of $\mathbf{A}$, where the j^{th} column $\mathbf{A}\mathbf{e}_j$ is the image of the j^{th} standard basis vector $\mathbf{e}_j$. In $\mathbb{R}^2$, a 2×2 matrix maps the unit square spanned by $\mathbf{e}_1, \mathbf{e}_2$ onto the parallelogram spanned by its columns (Figure 1.14).

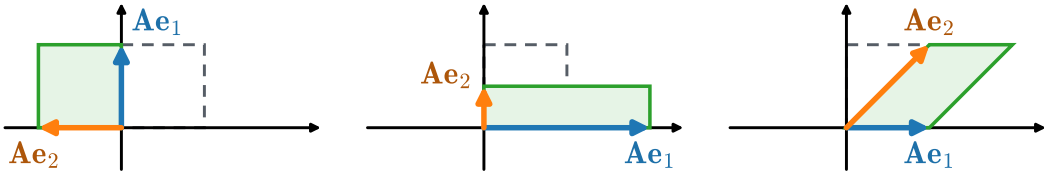


Figure 1.14: A 2×2 matrix maps the unit square (dashed) to the parallelogram spanned by its columns $\mathbf{Ae}_1$ and $\mathbf{Ae}_2$. From left to right: a rotation by $90°$, a scaling by 2 along $\mathbf{e}_1$ and by $1/2$ along $\mathbf{e}_2$, and a shear.

1.3.3.3 Matrix–Matrix Products

Given $\mathbf{A} \in \mathbb{R}^{m \times k}$ and $\mathbf{B} \in \mathbb{R}^{k \times n}$, their matrix product $\mathbf{C} = \mathbf{AB} \in \mathbb{R}^{m \times n}$ has (i, j) entry equal to the dot product of row i of $\mathbf{A}$ with column j of $\mathbf{B}$:

$$c_{ij} = \sum_{l=1}^{k} a_{il}\, b_{lj}. \tag{1.6}$$

```
B = torch.ones(3, 4)
A @ B
```

```
tensor([[ 3.,  3.,  3.,  3.],
        [12., 12., 12., 12.]])
```

We can now evaluate our 10-class linear classifier $\mathbf{O} = \mathbf{XW} + \mathbf{b}$ on the 256×784 minibatch `X` and pick each image's highest-scoring category via `argmax` along axis 1:

```
W = 0.01 * torch.randn(784, 10)  # Column j: category j
b = torch.zeros(10)              # Entry j: category j
O = X @ W + b                    # Row i: image i
O.shape, O.argmax(dim=1).shape
```

```
(torch.Size([256, 10]), torch.Size([256]))
```

Multiplying an $m \times k$ matrix by a $k \times n$ matrix computes mn dot products of length k, costing $\approx 2mkn$ FLOPs ($2 \times 256 \times 784 \times 10 \approx 4 \times 10^6$ FLOPs for `X @ W`, or $2n^3$ FLOPs against only $3n^2$ memory entries for two $n \times n$ matrices; see Chapter 13). Two special square matrices recur throughout linear algebra: the *identity matrix* $\mathbf{I} \in \mathbb{R}^{n \times n}$ (1s on the main diagonal and 0s elsewhere), which leaves every vector unchanged ($\mathbf{Ix} = \mathbf{x}$), and the *inverse* $\mathbf{A}^{-1}$ of an invertible matrix $\mathbf{A}$, defined by $\mathbf{A}^{-1}\mathbf{A} = \mathbf{AA}^{-1} = \mathbf{I}$ (Section 33.2.2.3).

Later models multiply a whole stack of matrices at once, one for each example in the minibatch. The `@` operator does this without a loop: on tensors with more than two axes it multiplies along the last two and broadcasts over the leading ones (Section 1.1.3.2). Where `@` does not fit, `einsum` lets us write the sum in Equation 1.6 out index by index. The string `'bil,lj->bij'` names the axes of each factor and of the result, and an index that is missing from the result, here `l`, is summed over. We stack three multiples of `A` and multiply them all by `B`:

```
As = torch.stack([A, 2 * A, 3 * A])  # Three 2 x 3 matrices, shape (3, 2, 3)
C = As @ B                           # B is broadcast across the batch
C.shape, (torch.einsum('bil,lj->bij', As, B) == C).all()
```

```
(torch.Size([3, 2, 4]), tensor(True))
```

The result holds three 2×4 products, one per matrix in the stack, and `einsum` agrees with `@` in every entry.

1.3.4 Norms

Norms measure the magnitude of a vector or matrix, serving in machine learning both as *loss functions* (measuring prediction error $\hat{\mathbf{y}} - \mathbf{y}$) and as *regularization penalties* (constraining weight magnitude).

1.3.4.1 Euclidean Length and Angle

The ℓ_2 *norm* (or *Euclidean norm*) of $\mathbf{x} \in \mathbb{R}^n$ is

$$\|\mathbf{x}\|_2 = \sqrt{\mathbf{x}^\top \mathbf{x}} = \sqrt{\sum_{i=1}^{n} x_i^2}, \tag{1.7}$$

often abbreviated $\|\mathbf{x}\|$. For two planar vectors $\mathbf{x}, \mathbf{y}$ separated by angle θ, equating the law of cosines $\|\mathbf{x}-\mathbf{y}\|^2 = \|\mathbf{x}\|^2 + \|\mathbf{y}\|^2 - 2\|\mathbf{x}\|\|\mathbf{y}\| \cos\theta$ with the bilinear expansion $(\mathbf{x}-\mathbf{y})^\top(\mathbf{x}-\mathbf{y}) = \|\mathbf{x}\|^2 - 2\mathbf{x}^\top\mathbf{y} + \|\mathbf{y}\|^2$ yields (Figure 1.15)

$$\mathbf{x}^\top \mathbf{y} = \|\mathbf{x}\|\|\mathbf{y}\| \cos\theta. \tag{1.8}$$

In arbitrary dimension d, we read Equation 1.8 in reverse to define the *cosine similarity* between nonzero vectors:

$$\cos\theta = \frac{\mathbf{x}^\top \mathbf{y}}{\|\mathbf{x}\|\|\mathbf{y}\|}, \tag{1.9}$$

which depends only on direction and ranges from $+1$ (aligned) through 0 (orthogonal) to -1 (opposed) (Figure 1.16).

The *Cauchy–Schwarz inequality* guarantees that $|\mathbf{x}^\top \mathbf{y}| \le \|\mathbf{x}\|\|\mathbf{y}\|$ (so $\cos\theta \in [-1, 1]$): for unit vectors $\mathbf{u} = \mathbf{x}/\|\mathbf{x}\|$ and $\mathbf{v} = \mathbf{y}/\|\mathbf{y}\|$, expanding $0 \le \|\mathbf{u} \mp \mathbf{v}\|^2 = 2 \mp 2\mathbf{u}^\top\mathbf{v}$ gives $|\mathbf{u}^\top\mathbf{v}| \le 1$.

```
def cosine(x, y):
    return torch.dot(x, y) / (torch.linalg.vector_norm(x) *
                              torch.linalg.vector_norm(y))

x, y = torch.tensor([1.0, 1, 1, 1]), torch.tensor([1.0, 1, 1, -1])
cosine(x, y), cosine(2 * x, y), torch.acos(cosine(x, y))
```

```
(tensor(0.5000), tensor(0.5000), tensor(1.0472))
```

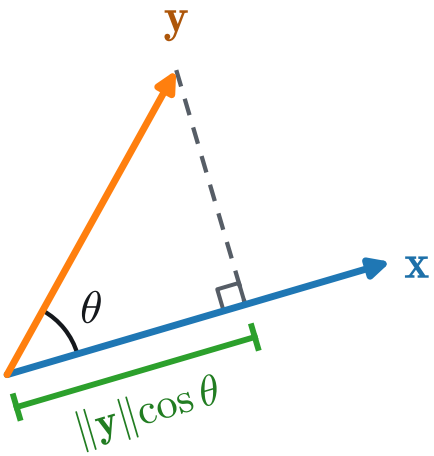


Figure 1.15: The dot product $\mathbf{x}^\top \mathbf{y} = \|\mathbf{x}\|\|\mathbf{y}\| \cos\theta$ is the length of $\mathbf{x}$ times the signed length $\|\mathbf{y}\| \cos\theta$ of the projection of $\mathbf{y}$ onto the line through $\mathbf{x}$.

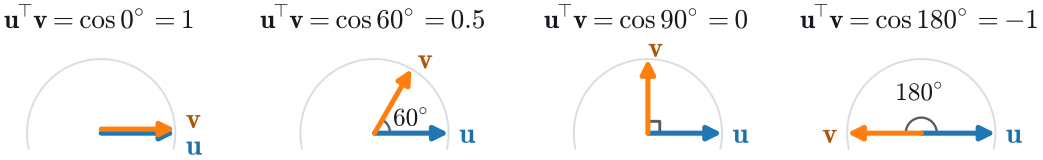


Figure 1.16: Two unit vectors at angle θ: their dot product equals $\cos\theta$, ranging from 1 (aligned) through 0 (perpendicular) to -1 (opposed).

1.3.4.2 The ℓ_1, ℓ_p, and Frobenius Norms

The ℓ_1 *norm* (or *Manhattan distance*) sums the absolute values of the components (Figure 1.17):

$$\|\mathbf{x}\|_1 = \sum_{i=1}^{n} |x_i|. \tag{1.10}$$

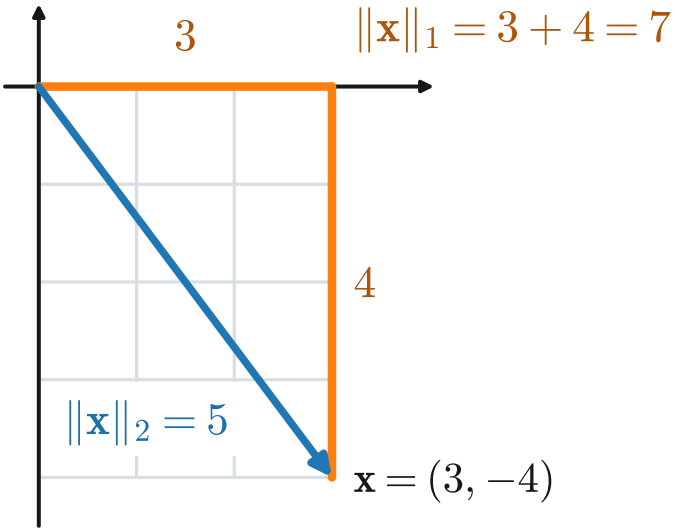


Figure 1.17: The ℓ_2 norm of $\mathbf{x}$ is the length of the straight path to $\mathbf{x}$; the ℓ_1 norm is the length of the path along the grid.

```
x = torch.tensor([3.0, -4.0])
torch.linalg.vector_norm(x), torch.abs(x).sum()
```

```
(tensor(5.), tensor(7.))
```

Because ℓ_2 squares components while ℓ_1 does not, ℓ_1 loss is more robust to large outlier errors, and ℓ_1 weight penalties drive weights to exact zero (sparsity). Both belong to the family of ℓ_p *norms* $\|\mathbf{x}\|_p = (\sum_{i=1}^{n} |x_i|^p)^{1/p}$ ($p \geq 1$), with $\|\mathbf{x}\|_\infty = \max_i |x_i|$ as $p \to \infty$. Any valid norm satisfies (i) absolute homogeneity $\|\alpha\mathbf{x}\| = |\alpha|\|\mathbf{x}\|$, (ii) the triangle inequality $\|\mathbf{x} + \mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|$, and (iii) positive definiteness ($\|\mathbf{x}\| > 0$ for $\mathbf{x} \neq \mathbf{0}$).

For an $m \times n$ matrix $\mathbf{A}$, the *Frobenius norm* computes the ℓ_2 norm of its mn entries:

$$\|\mathbf{A}\|_\mathrm{F} = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij}^2}. \tag{1.11}$$

```
torch.linalg.matrix_norm(torch.ones(4, 9))
```

```
tensor(6.)
```

PyTorch separates `torch.linalg.vector_norm` from `torch.linalg.matrix_norm`; the latter treats a 3D tensor as a batch of 2D matrices.

1.3.5 Eigenvectors and Eigenvalues

What happens when a 50-layer network multiplies an input vector $\mathbf{x}$ by matrices repeatedly? A nonzero vector $\mathbf{v} \neq \mathbf{0}$ is an *eigenvector* of $\mathbf{A} \in \mathbb{R}^{n \times n}$ with scalar *eigenvalue* λ if multiplying by $\mathbf{A}$ simply scales $\mathbf{v}$ without rotating it:

$$\mathbf{A}\mathbf{v} = \lambda \mathbf{v} \quad \Longrightarrow \quad \mathbf{A}^k \mathbf{v} = \lambda^k \mathbf{v}. \tag{1.12}$$

For our symmetric matrix $\mathbf{S} = \left[\begin{smallmatrix} 2 & 1 \\ 1 & 2 \end{smallmatrix}\right]$, $(1, 1)$ is an eigenvector with $\lambda = 3$ and $(-1, 1)$ is an eigenvector with $\lambda = 1$ (Figure 1.18).

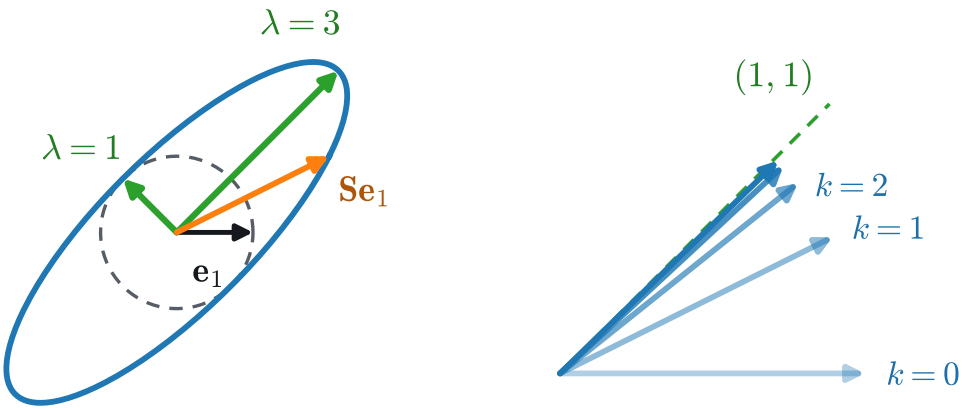


Figure 1.18: Left: $\mathbf{S}$ maps the unit circle (dashed) to an ellipse whose axes are its eigenvectors, stretched by $\lambda = 3$ and $\lambda = 1$, while $\mathbf{e}_1$ is turned to $\mathbf{S}\mathbf{e}_1 = (2, 1)$. Right: the directions of $\mathbf{S}^k \mathbf{e}_1$ for $k = 0, \ldots, 4$ approach the eigenvector $(1, 1)$ of the largest eigenvalue.

We compute eigenvalues and unit eigenvectors of a real symmetric matrix using `linalg.eigh`:

```
S = torch.tensor([[2.0, 1.0], [1.0, 2.0]])
eigenvalues, eigenvectors = torch.linalg.eigh(S)
eigenvalues, eigenvectors
```

```
(tensor([1., 3.]),
 tensor([[-0.7071,  0.7071],
         [ 0.7071,  0.7071]]))
```

By the *spectral theorem* (Section 33.3), every real symmetric $n \times n$ matrix $\mathbf{A}$ has n real eigenvalues and a complete orthonormal basis of eigenvectors $\mathbf{v}_1, \ldots, \mathbf{v}_n$. Expanding any vector $\mathbf{x} = \sum_{i=1}^{n} c_i \mathbf{v}_i$ (with $c_i = \mathbf{v}_i^\top \mathbf{x}$) gives

$$\mathbf{A}^k\mathbf{x} = \sum_{i=1}^{n} c_i \lambda_i^k \mathbf{v}_i. \tag{1.13}$$

As k grows, the term with the largest $|\lambda_i|$ dominates: $\mathbf{A}^k\mathbf{x}$ aligns with the dominant eigenvector and grows by $\max_i |\lambda_i|$ per step (*power iteration*). For a symmetric matrix this growth rate is also the largest factor by which the matrix can stretch any vector, its *spectral norm* $\|\mathbf{A}\|_2 = \max_{\mathbf{x}\neq\mathbf{0}} \|\mathbf{A}\mathbf{x}\|/\|\mathbf{x}\|$.

```
norm = torch.linalg.vector_norm
v = torch.tensor([1.0, 0.0])
for _ in range(10):
    v_prev, v = v, S @ v
growth = norm(v) / norm(v_prev)
growth, v / norm(v)
```

```
(tensor(3.0000), tensor([0.7071, 0.7071]))
```

Returning to our 50-layer thought experiment: for $\mathbf{M} = \left[\begin{smallmatrix} 1 & 0.1 \\ 0.1 & 1 \end{smallmatrix}\right]$, whose eigenvalues $\lambda_1 = 1.1$ and $\lambda_2 = 0.9$ differ from 1 by just 10%, multiplying its two unit eigenvectors by $\mathbf{M}$ 50 times amplifies the first by $1.1^{50} \approx 117.4$ and shrinks the second to $0.9^{50} \approx 0.0052$:

```
M = torch.tensor([[1.0, 0.1], [0.1, 1.0]])
# The eigenvectors (1, 1) and (-1, 1) of M, as columns of length 1
V = torch.tensor([[1.0, -1.0], [1.0, 1.0]]) / 2 ** 0.5
for _ in range(50):  # 50 "layers"
    V = M @ V
torch.linalg.vector_norm(V, dim=0)
```

```
tensor([1.1739e+02, 5.1538e-03])
```

The weight matrices of a network are not symmetric, and most are not even square. Do eigenvalues still tell us how far such a matrix stretches a vector? They do not. Both eigenvalues of $\mathbf{A} = \left[\begin{smallmatrix} 0 & 2 \\ 0 & 0 \end{smallmatrix}\right]$ are 0, and $\mathbf{A}^2 = \mathbf{0}$, yet $\mathbf{A}$ maps $\mathbf{e}_2$ to $2\mathbf{e}_1$ and so doubles its length. Since $\|\mathbf{A}\mathbf{x}\|_2^2 = \mathbf{x}^\top \mathbf{A}^\top \mathbf{A}\mathbf{x}$, stretching is governed by the symmetric matrix $\mathbf{A}^\top\mathbf{A}$. The square roots of its eigenvalues, $\sigma_i = \sqrt{\lambda_i(\mathbf{A}^\top\mathbf{A})}$, are the *singular values* of $\mathbf{A}$, and they exist for every matrix, square or not. We compute them for $\mathbf{S}$ and for $\mathbf{A}$:

```
A = torch.tensor([[0.0, 2.0], [0.0, 0.0]])  # Both eigenvalues are 0
svdvals = torch.linalg.svdvals
(svdvals(S), svdvals(A), torch.linalg.eigvalsh(A.T @ A).sqrt(),
 torch.linalg.matrix_rank(A))
```

```
(tensor([3., 1.]), tensor([2., 0.]), tensor([0., 2.]), tensor(1))
```

For the symmetric $\mathbf{S}$ the singular values are 3 and 1 (up to rounding), the magnitudes of its eigenvalues: for a symmetric matrix the two notions agree. For $\mathbf{A}$ they are 2 and 0, and the eigenvalues of $\mathbf{A}^\top\mathbf{A}$ return the same two numbers in ascending order. The largest singular value is the spectral norm of any matrix. The number of nonzero singular values is its *rank*, the number of independent directions that survive the multiplication, which is one for $\mathbf{A}$. Both facts follow from the singular value decomposition (Section 33.5).

Unless each layer's weight matrix keeps vector lengths close to 1 on average, activations and gradients *explode* or *vanish* exponentially with depth (Section 4.4).

1.3.6 Summary

Linear algebra expresses whole-minibatch models compactly: stacking 256 flattened images as $\mathbf{X} \in \mathbb{R}^{256\times 784}$ computes all $2{,}560$ category scores via $\mathbf{O} = \mathbf{X}\mathbf{W} + \mathbf{b}$. Reductions aggregate along specified axes (`keepdims=True` retains the axis as length 1 for broadcasting). Dot products $\mathbf{x}^\top\mathbf{y} = \sum_i x_i y_i$ build matrix–vector products $\mathbf{A}\mathbf{x}$ and matrix–matrix products $\mathbf{A}\mathbf{B}$ (costing $\approx 2mkn$ FLOPs). Norms (ℓ_2, ℓ_1, ℓ_p, Frobenius, and spectral) measure vector and matrix magnitudes, normalized dot products give cosine similarity $\cos\theta = \mathbf{x}^\top\mathbf{y}/(\|\mathbf{x}\|\|\mathbf{y}\|)$, and eigenvectors ($\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$) explain why repeated matrix multiplication scales components as λ^k. For a matrix that is not symmetric, the singular values measure the stretching, and the largest of them is the spectral norm.

1.3.7 Exercises

1. **Transposes and matrix products.** Prove the following algebraic identities:
    1. Show that the transpose of a matrix product reverses the order of the factors: $(\mathbf{A}\mathbf{B})^\top = \mathbf{B}^\top\mathbf{A}^\top$ whenever $\mathbf{A}\mathbf{B}$ is defined. (Write out the (i,j) entry of both sides using Equation 1.6.)
    2. Show that for *any* rectangular matrix $\mathbf{X} \in \mathbb{R}^{m\times n}$, the product $\mathbf{X}\mathbf{X}^\top$ is always square and symmetric (and likewise for $\mathbf{X}^\top\mathbf{X}$). For our 256×784 image minibatch $\mathbf{X}$, what is the shape of $\mathbf{X}\mathbf{X}^\top$, and what does its (i,j) entry measure?
2. code **Predicting reduction and broadcasting shapes.** Without running any code, predict the shape (or error) of each expression below for `images` of shape $(256, 1, 28, 28)$ and `X` of shape $(256, 784)$, then verify your answers in code:
    1. What does `len(images)` return? What are the shapes of `images.sum(axis=(2, 3))` with `keepdims=False` versus `keepdims=True`? Which of the two can divide `images` directly via broadcasting, and what does `images / images.sum(axis=(2, 3), keepdims=True)` compute?
    2. What happens when you evaluate `X - X.mean(axis=1)` on our 256×784 matrix `X`, and how does `keepdims=True` fix it?
    3. Now suppose your minibatch happens to contain $n = 784$ images, so that `X` is a square 784×784 matrix. Show that `X - X.mean(axis=1)` executes *without raising any error*. Write out what entry (i,j) of `X - X.mean(axis=1)` actually computes in terms of the entries of `X`, and explain why a silent broadcasting bug on a square matrix is far more dangerous than a shape error.
3. **How mutually obtuse can a set of vectors be?** The following questions explore how negative all pairwise dot products among k vectors can simultaneously be.
    1. Can three vectors $\mathbf{u}, \mathbf{v}, \mathbf{w} \in \mathbb{R}^2$ in the plane satisfy $\mathbf{u}^\top\mathbf{v} < 0$, $\mathbf{v}^\top\mathbf{w} < 0$, and $\mathbf{u}^\top\mathbf{w} < 0$ simultaneously? Provide a concrete numerical example, or prove that none exists.
    2. Suppose $k \geq 2$ unit vectors $\mathbf{u}_1, \ldots, \mathbf{u}_k \in \mathbb{R}^d$ ($d \geq k-1$) are arranged symmetrically so that every distinct pair has the exact same dot product $\mathbf{u}_i^\top\mathbf{u}_j = c$ for all $i \neq j$. By expanding $0 \leq \|\mathbf{u}_1 + \cdots + \mathbf{u}_k\|_2^2$, prove that $c \geq -\frac{1}{k-1}$. What does

this bound imply about the pairwise cosine similarity of 10 unit-length class templates in a 10-class classifier when all 10 templates are separated by equal angles?

Adapted from MIT 18.06, Problem Set 1, Section 1.2, Problem 28.

4. code **Norms across multiple parameter tensors.** The vector-norm routines introduced in this section can also be applied to higher-order tensors.
   1. Evaluate the vector norm with default arguments on a 3D tensor of shape $(2, 3, 4)$, state in words what mathematical formula it computes, and confirm your answer by squaring, summing, and taking the square root of the entries manually.
   2. A deep neural network stores its learnable parameters across a list of weight and bias tensors of varying orders and shapes. When monitoring training or clipping gradients (Section 8.4.2), we often need the single global ℓ_2 norm of all parameters concatenated into one giant vector. Express this global ℓ_2 norm in terms of the individual tensor norms without flattening or concatenating the tensors, and verify your formula in code on three tensors of shapes $(3, 4)$, $(4,)$, and $(2, 3, 5)$.

5. **Associativity of matrix multiplication and computational cost.** Although matrix multiplication is associative, meaning $(\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC})$ mathematically, the order in which you parenthesize a chain of matrix products can change its computational cost by orders of magnitude. Given $\mathbf{A} \in \mathbb{R}^{1000\times 50000}$, $\mathbf{B} \in \mathbb{R}^{50000\times 20}$, and $\mathbf{C} \in \mathbb{R}^{20\times 10000}$, calculate the total FLOP count and the number of entries in the intermediate product matrix under both $(\mathbf{AB})\mathbf{C}$ and $\mathbf{A}(\mathbf{BC})$. By what factors do the two evaluation orders differ in arithmetic work and intermediate memory?

6. code **All-pairs Euclidean distance matrix without loops.** ● Given two collections of 50 vectors in $\mathbb{R}^8$ stacked as the rows of $\mathbf{X} \in \mathbb{R}^{50\times 8}$ and $\mathbf{Y} \in \mathbb{R}^{50\times 8}$, we want to compute the 50×50 matrix $\mathbf{D}$ of squared Euclidean distances $D_{ij} = \|\mathbf{x}_i - \mathbf{y}_j\|_2^2$ without Python loops:
   1. Compute $\mathbf{D}$ first via 3D broadcasting: reshape $\mathbf{X}$ to $(50, 1, 8)$ and $\mathbf{Y}$ to $(1, 50, 8)$, subtract, square, and sum along the last axis. Second, compute $\mathbf{D}$ via the quadratic expansion $\|\mathbf{x}_i - \mathbf{y}_j\|_2^2 = \|\mathbf{x}_i\|_2^2 + \|\mathbf{y}_j\|_2^2 - 2\mathbf{x}_i^\top \mathbf{y}_j$, which requires only row-norm reductions, 2D broadcasting, and a single matrix multiplication $\mathbf{XY}^\top$. Verify that both results match to floating-point precision.
   2. When $\mathbf{X}$ and $\mathbf{Y}$ each contain $10{,}000$ vectors in $\mathbb{R}^{128}$, how many intermediate floating-point numbers must the 3D-broadcasting method allocate compared with the matrix-product method? Which method should you use at scale?
   3. Explain why floating-point rounding in $\|\mathbf{x}_i\|_2^2 + \|\mathbf{y}_j\|_2^2 - 2\mathbf{x}_i^\top \mathbf{y}_j$ can occasionally produce tiny negative numbers (such as -10^{-7}) when $\mathbf{x}_i = \mathbf{y}_j$, and show how to clamp the output before taking a square root.

Adapted from Stanford CS231n, Assignment 1 (k-nearest-neighbor distances), and fast.ai, Deep Learning from the Foundations, Lesson 8.

7. code **Power iteration and convergence rate.** ● Implement normalized power iteration, repeatedly updating $\mathbf{v}_{k+1} = \mathbf{A}\mathbf{v}_k / \|\mathbf{A}\mathbf{v}_k\|_2$ from a random unit vector $\mathbf{v}_0$, on the 3×3 symmetric matrix

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 0 & 4 \\ 3 & 4 & 5 \end{bmatrix}. \tag{1.14}$$

1. Record the first iteration k at which the absolute cosine similarity $|\mathbf{v}_k^\top \mathbf{v}_{k-1}|$ between consecutive iterates exceeds 0.999. Compare your converged vector $\mathbf{v}_k$ against the dominant eigenvector returned by `eigh(A)`, and explain why the two vectors may differ by an overall sign ± 1.
2. At each step, measure the sine of the angle between $\mathbf{v}_k$ and the true dominant eigenvector $\mathbf{u}_{\max}$, $\sin\theta_k = \sqrt{1 - (\mathbf{v}_k^\top \mathbf{u}_{\max})^2}$, and compute the per-step shrinkage ratio $\sin\theta_{k+1} / \sin\theta_k$. Express this asymptotic ratio in terms of the eigenvalues of $\mathbf{A}$ using Equation 1.13, and use it to explain why power iteration converged in so few steps.
3. Now run power iteration on the shifted matrix $\mathbf{B} = \mathbf{A} - 9\mathbf{I}$. Which eigenvalue and eigenvector of $\mathbf{A}$ does power iteration converge to now, what happens to the sign of consecutive iterates $\mathbf{v}_k$ and $\mathbf{v}_{k+1}$, and why?

Adapted from MIT 18.06, Problem Set 1, Section 2.1, Problems 29–30.

1.4 Calculus

In Ancient Greece, Archimedes calculated the area of a circle by inscribing regular polygons with increasing numbers of sides $n \to \infty$ (Figure 1.19): as the triangles become thinner, their height approaches r and their combined bases approach the circumference $2\pi r$, giving area $\frac{1}{2} r(2\pi r) = \pi r^2$. Approximating curved quantities by simple linear pieces in the limit of small steps lies at the root of both integral calculus and *differential calculus*.

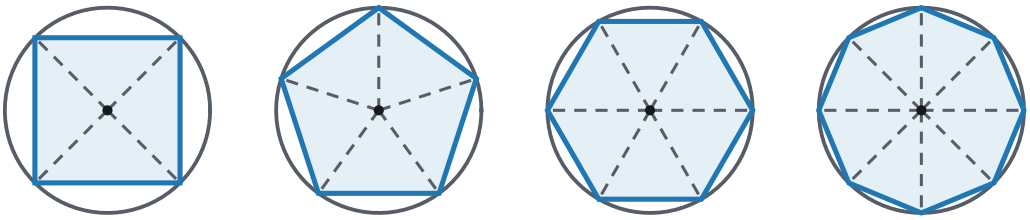


Figure 1.19: Regular polygons with 4, 5, 6, and 8 sides inscribed in a circle. As the number of sides grows, the area of the polygon approaches the area of the circle.

In deep learning, differential calculus is the engine of *optimization*. Training a model means adjusting its parameters to minimize a scalar loss function measuring how poorly the model fits its training examples. With millions or billions of parameters, probing each parameter by trial and error would require millions of forward passes per update. Instead, the *derivative* gives the instantaneous rate at which the loss changes with respect to each parameter; collecting all partial derivatives into the *gradient* vector identifies the direction of steepest ascent, and the *chain rule* assembles gradients across nested layers in a single backward pass.

In this section, we develop derivatives, gradients, and the chain rule on a running linear-regression example small enough that every number can be verified by hand. In Section 1.5, we will see how deep learning frameworks automate these calculations for arbitrary programs.

1.4.1 Derivatives

1.4.1.1 The Derivative as a Limit

Suppose a single-parameter model predicts the constant value w for every example, and we evaluate it on three training examples with labels 1, 1, and 0. Summing the squared errors gives

$$L(w) = (w-1)^2 + (w-1)^2 + (w-0)^2 = 3w^2 - 4w + 2. \tag{1.15}$$

At $w = 1$, the loss is $L(1) = 1$:

```
def loss(w):
    return 3 * w ** 2 - 4 * w + 2
```

To see how $L(w)$ responds to a small perturbation $h \neq 0$, we evaluate the difference quotient $\frac{L(1+h)-L(1)}{h}$ as h shrinks from 10^{-1} to 10^{-5}:

```
for h in 10.0 ** -np.arange(1, 6):
    print(f'h = {h:.0e}   ratio = {(loss(1 + h) - loss(1)) / h:.5f}')
```

```
h = 1e-01   ratio = 2.30000
h = 1e-02   ratio = 2.03000
h = 1e-03   ratio = 2.00300
h = 1e-04   ratio = 2.00030
h = 1e-05   ratio = 2.00003
```

Expanding $L(1+h) = 1 + 2h + 3h^2$ shows that $\frac{L(1+h)-L(1)}{h} = 2 + 3h \to 2$ as $h \to 0$. Formally, for $f : \mathbb{R} \to \mathbb{R}$, the *derivative* of f at x is the limit

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}. \tag{1.16}$$

Geometrically, the difference quotient is the slope of the *secant line* through $P = (x, f(x))$ and $Q = (x+h, f(x+h))$; as $h \to 0$, the secant rotates into the *tangent line* at P with slope $f'(x)$ (Figure 1.20).

Not every function is differentiable everywhere: $|x|$ has a corner at $x = 0$ (slope -1 from the left, $+1$ from the right). Even worse, discrete classification accuracy is flat ($f'(x) = 0$) almost everywhere and jumps discontinuously where a prediction flips, providing no gradient signal. That is why classifiers are trained by minimizing smooth surrogate losses such as cross-entropy (Figure 1.21).

1.4.1.2 Rules of Differentiation

Given $y = f(x)$, we write $f'(x) = y' = \frac{dy}{dx} = \frac{df}{dx} = \frac{d}{dx} f(x) = Df(x)$. Common elementary derivatives are

$$\frac{d}{dx} C = 0, \quad \frac{d}{dx} x^n = n x^{n-1}, \quad \frac{d}{dx} e^x = e^x, \quad \frac{d}{dx} \log x = \frac{1}{x} \quad (x > 0),$$

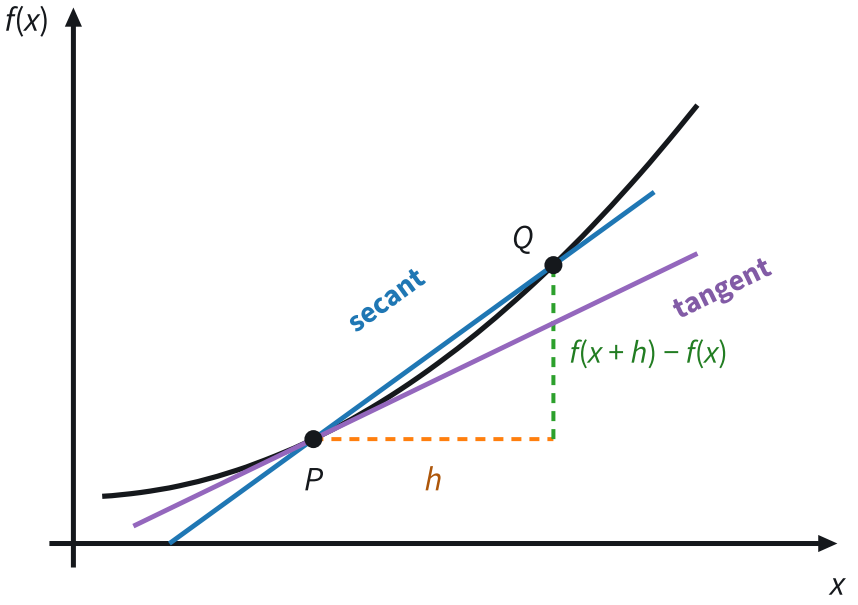


Figure 1.20: The difference quotient $(f(x+h)-f(x))/h$ is the slope of the secant through P and Q. As $h \to 0$, Q slides toward P and the secant turns into the tangent at P, whose slope is $f'(x)$.

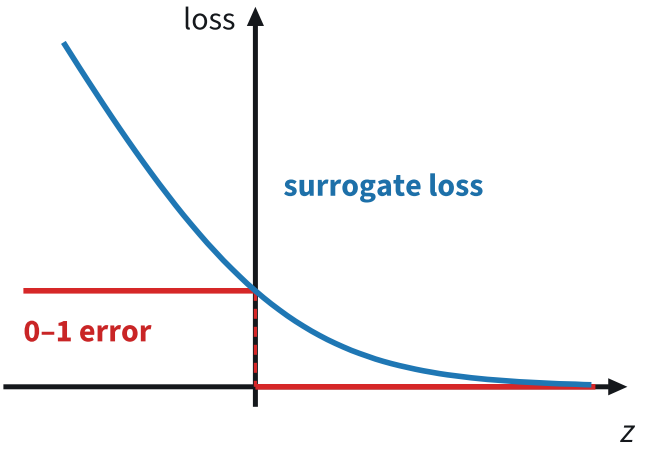


Figure 1.21: The 0–1 error (1 for a wrong prediction, 0 for a correct one) and a smooth surrogate loss, plotted against a score z that is positive when the classifier is right. The 0–1 error is flat on either side of its jump, so its slope says nothing. The surrogate slopes downward everywhere and tells the parameters which way to move.

and combinations of differentiable functions f, g obey

$$\frac{d}{dx}[Cf(x)] = Cf'(x), \qquad \frac{d}{dx}[f(x)+g(x)] = f'(x)+g'(x),$$
$$\frac{d}{dx}[f(x)g(x)] = f'(x)g(x)+f(x)g'(x), \qquad \frac{d}{dx}\frac{f(x)}{g(x)} = \frac{f'(x)g(x)-f(x)g'(x)}{g(x)^2}.$$

Figure 1.22 illustrates the product rule geometrically via the area of a $f(x) \times g(x)$ rectangle (full proofs appear in Section 34.1). Applying these rules to $L(w) = 3w^2 - 4w + 2$ immediately gives $L'(w) = 6w - 4$, so $L'(1) = 2$.

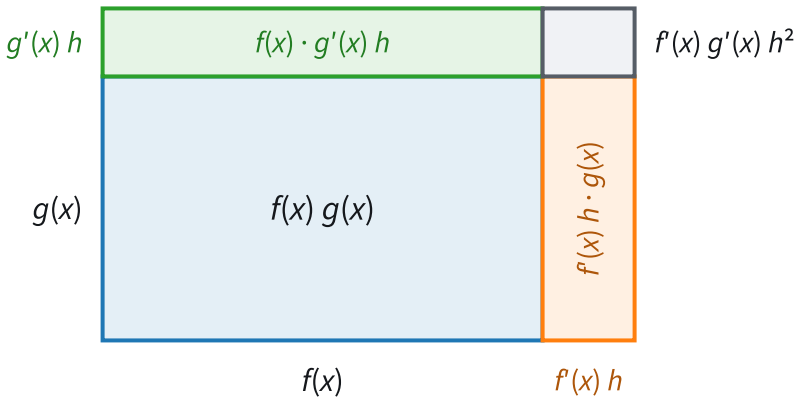


Figure 1.22: The product rule as the area of a rectangle. When x grows by h, the sides $f(x)$ and $g(x)$ grow by about $f'(x)h$ and $g'(x)h$, and the area grows by two thin strips and a corner of area $f'(x)g'(x)h^2$, which is negligible for small h.

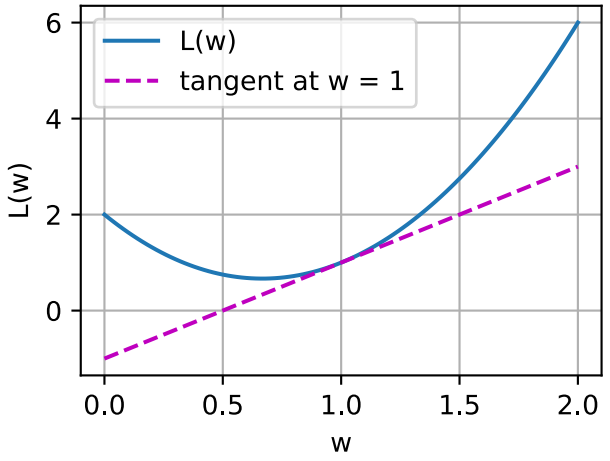
1.4.1.3 First-Order Linear Approximation and Minima

Multiplying $\frac{f(x+h)-f(x)}{h} \approx f'(x)$ by h gives the *first-order linear approximation*

$$f(x+h) \approx f(x) + f'(x)\,h. \tag{1.17}$$

Because $L'(1) = 2 > 0$, decreasing w lowers the loss. Stepping opposite the derivative via $w \leftarrow w - \eta L'(w)$ with step size $\eta = 0.1$ moves w from 1 to $1 - 0.1(2) = 0.8$ and drops L from 1 to $L(0.8) = 0.72$, close to the tangent prediction $1 + 2(-0.2) = 0.60$ (the 0.12 gap is the neglected quadratic curvature $3h^2$). At any smooth local minimum, the tangent must be horizontal ($f'(x) = 0$); setting $L'(w) = 6w - 4 = 0$ gives the unique global minimizer $w^* = 2/3$, the sample mean of the three labels $1, 1, 0$.

```
w = np.linspace(0, 2, 101)
d2l.plot(w, [loss(w), 2 * w - 1], 'w', 'L(w)',
         legend=['L(w)', 'tangent at w = 1'])
```



1.4.2 Partial Derivatives and Gradients

1.4.2.1 Partial Derivatives

Now give our three-example problem scalar inputs 0, 1, and 2 paired with labels 1, 1, and 0, and fit an affine line $w_1 + w_2 \cdot \text{input}$:

$$L(w_1, w_2) = (w_1 - 1)^2 + (w_1 + w_2 - 1)^2 + (w_1 + 2w_2 - 0)^2. \tag{1.18}$$

For a multivariate function $y = f(x_1, \dots, x_n)$, the *partial derivative* with respect to x_i perturbs only coordinate i while holding all other variables fixed:

$$\frac{\partial f}{\partial x_i} = \lim_{h \to 0} \frac{f(x_1, \dots, x_i + h, \dots, x_n) - f(x_1, \dots, x_i, \dots, x_n)}{h}. \tag{1.19}$$

At $(w_1, w_2) = (1, 0)$, holding $w_2 = 0$ fixed gives $L(w_1, 0) = 3w_1^2 - 4w_1 + 2$, so $\frac{\partial L}{\partial w_1}(1, 0) = 2$; holding $w_1 = 1$ fixed gives $L(1, w_2) = 5w_2^2 + 4w_2 + 1$, with derivative $10w_2 + 4$, so $\frac{\partial L}{\partial w_2}(1, 0) = 4$ (Figure 1.23).

In matrix–vector form (Section 1.3), writing each example as a row $(1, \text{input}_i)$ of $\mathbf{X}$ gives

$$\mathbf{X} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 1 & 2 \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}, \quad L(\mathbf{w}) = \|\mathbf{X}\mathbf{w} - \mathbf{y}\|^2. \tag{1.20}$$

We can check these partial derivatives numerically by defining `line_loss(w)` and a finite-difference helper `numerical_gradient(f, w, h=1e-6)` that perturbs one coordinate of `w` at a time by $h = 10^{-6}$ in `float64`:

```
X = np.array([[1.0, 0.0], [1.0, 1.0], [1.0, 2.0]])  # rows [1, input]
y = np.array([1.0, 1.0, 0.0])
```

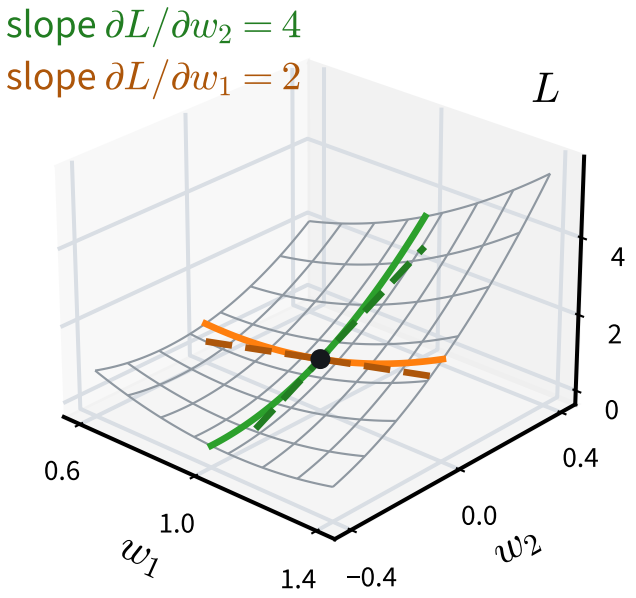


Figure 1.23: The loss $L(w_1, w_2)$ of the line model near $\mathbf{w} = (1, 0)$ (black dot). Holding $w_2 = 0$ cuts out the orange curve, the loss of the constant model; holding $w_1 = 1$ cuts out the green curve. Their tangents at the dot (dashed) have slopes $\partial L/\partial w_1 = 2$ and $\partial L/\partial w_2 = 4$.

```
def line_loss(w):
    return np.sum((X @ w - y) ** 2)

def numerical_gradient(f, w, h=1e-6):
    """Finite differences of f at w, moving one coordinate at a time."""
    grad = np.zeros_like(w)
    for i in range(len(w)):
        step = np.zeros_like(w)
        step[i] = h
        grad[i] = (f(w + step) - f(w)) / h
    return grad
```

Evaluating `numerical_gradient(line_loss, w)` at $\mathbf{w} = (1, 0)$ reproduces our analytical partial derivatives $(2, 4)$ to within the $O(h) \approx 10^{-6}$ truncation error:

```
w = np.array([1.0, 0.0])
numerical_gradient(line_loss, w)
```

```
array([2.000003, 4.000005])
```

Note that `numerical_gradient` requires n extra evaluations of f for n parameters, which becomes prohibitive when $n = 10^8$.

1.4.2.2 Numerical Precision Limits of Finite Differences

Finite differences also suffer from floating-point cancellation when h is too small. In finite precision (machine epsilon $\epsilon \approx 2.2 \times 10^{-16}$ in `float64`, $\epsilon \approx 1.2 \times 10^{-7}$ in `float32`), subtracting nearly equal numbers $f(x+h) - f(x)$ cancels leading digits (*catastrophic cancellation*) and divides the $\sim \epsilon$ rounding residual by h. Thus *truncation error* ($\propto h$) and *rounding error* ($\propto \epsilon/h$) compete:

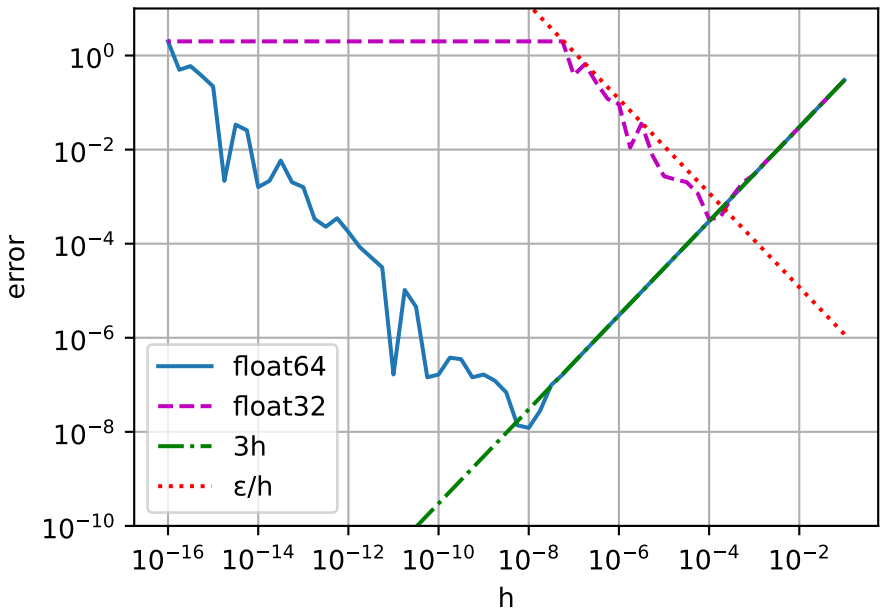
```
def quotient_error(f, x, df, h):
    """Error of (f(x + h) - f(x)) / h as an estimate of f'(x) = df."""
    return abs((f(x + h) - f(x)) / h - df)

h = np.logspace(-1, -16, 61)  # 61 values of h, evenly spaced on a log scale
```

To compare 64-bit NumPy arithmetic against the 32-bit floating-point format used by default in deep learning, we cast the step grid `h` to a `float32` framework tensor `h32`:

```
h32 = torch.tensor(h, dtype=torch.float32)
```

We now evaluate `quotient_error(loss, 1, 2, ...)` at $w = 1$ (where the exact derivative is $L'(1) = 2$) across all 61 step sizes in both `float64` (`h`) and `float32` (`h32`), overlaying the truncation error line $3h$ and the `float32` rounding asymptote ϵ/h:

```
eps32 = np.finfo(np.float32).eps
d2l.plot(h, [quotient_error(loss, 1, 2, h), quotient_error(loss, 1, 2, h32),
             3 * h, eps32 / h], 'h', 'error', xscale='log', yscale='log',
         ylim=(1e-10, 1e1), legend=['float64', 'float32', '3h', 'ε/h'],
         figsize=(5, 3.5))
```



Balancing $3h \approx \epsilon/h$ puts the optimal step at $h^* \approx \sqrt{\epsilon/3}$, losing half the format's digits (best error $\sim 10^{-8}$ at $h \approx 10^{-8}$ in `float64`, and $\sim 10^{-4}$ at $h \approx 10^{-4}$ in `float32`). Analytical differentiation via the chain rule avoids both the $O(n)$ cost and the $\sqrt{\epsilon}$ precision loss, leaving `float64` finite differences for unit-test gradient checking.

1.4.2.3 The Gradient and Steepest Descent

Concatenating all n partial derivatives of $f : \mathbb{R}^n \to \mathbb{R}$ forms the *gradient* vector $\nabla_{\mathbf{x}} f(\mathbf{x}) = (\frac{\partial f}{\partial x_1}, \dots, \frac{\partial f}{\partial x_n})^\top$ (matching the shape of $\mathbf{x}$), which generalizes Equation 1.17 to simultaneous perturbations $\mathbf{h} \in \mathbb{R}^n$:

$$f(\mathbf{x} + \mathbf{h}) \approx f(\mathbf{x}) + \nabla f(\mathbf{x})^\top \mathbf{h}. \tag{1.21}$$

For a step $\mathbf{h} = t\mathbf{d}$ of length $t > 0$ along unit vector $\mathbf{d}$ ($\|\mathbf{d}\|_2 = 1$), Equation 1.8 gives $f(\mathbf{x} + t\mathbf{d}) - f(\mathbf{x}) \approx t\,\|\nabla f(\mathbf{x})\| \cos\theta$. Thus $\nabla f(\mathbf{x})$ points in the direction of *steepest ascent* ($\theta = 0°$), $-\nabla f(\mathbf{x})$ points in the direction of *steepest descent* ($\theta = 180°$), and $\nabla f(\mathbf{x})$ is perpendicular to the level set passing through $\mathbf{x}$ ($\theta = 90°$) (Figure 1.24).

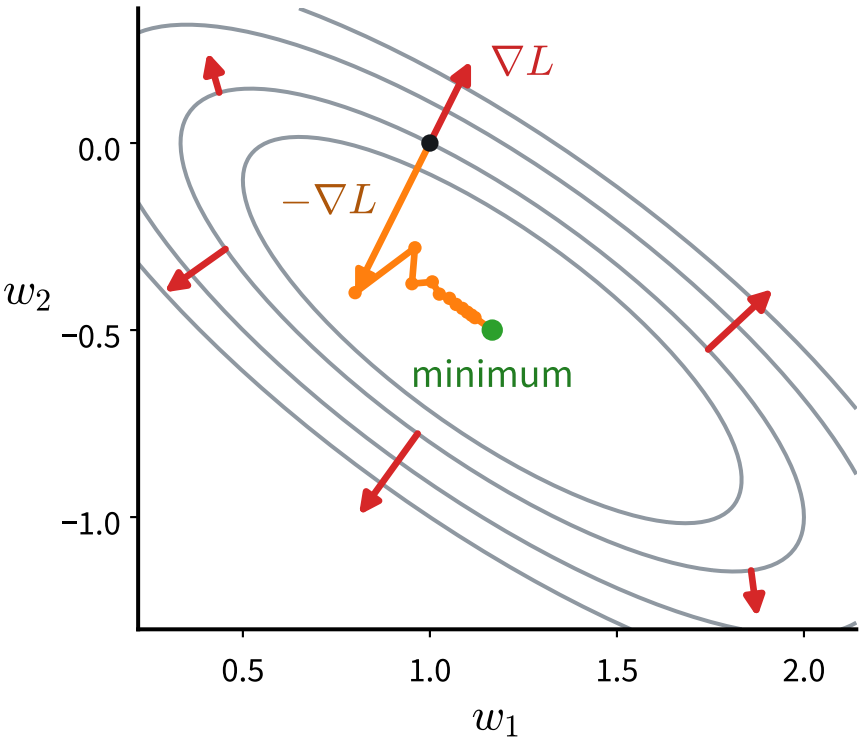


Figure 1.24: Level sets of the loss $L(w_1, w_2)$ of the line model. The gradients (red) are perpendicular to the level set through their base points and point uphill. From $\mathbf{w} = (1, 0)$ (black dot), gradient descent with $\eta = 0.1$ (orange) starts along $-\nabla L$, which does not point at the minimum, and zigzags toward it.

This motivates *gradient descent*, $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla L(\mathbf{w})$: from $\mathbf{w} = (1, 0)$ with learning rate $\eta = 0.1$, one step moves to $(1, 0) - 0.1(2, 4) = (0.8, -0.4)$, cutting the loss from 1.0 to

0.4. Why does steepest descent zigzag across the valley in Figure 1.24 instead of pointing straight at the global minimum $\mathbf{w}^* = (7/6, -1/2)$? Because "steepest" is measured per unit of Euclidean step length $\|\mathbf{h}\|_2$, which depends on the units of each coordinate: stretching one feature axis flattens its contours and tilts $-\nabla L$ away from the minimum, which is why standardizing features (Section 1.2) and preconditioning curvature (Chapter 9) matter so much in practice.

1.4.2.4 Matrix Calculus Identities

Three vector gradient identities (Section 34.4) allow us to differentiate vector expressions without writing out coordinates:

$$\nabla_{\mathbf{x}}(\mathbf{a}^\top \mathbf{x}) = \mathbf{a}, \qquad \nabla_{\mathbf{x}}(\mathbf{x}^\top \mathbf{A}\mathbf{x}) = (\mathbf{A} + \mathbf{A}^\top)\mathbf{x},$$
$$\nabla_{\mathbf{x}}\|\mathbf{x}\|^2 = 2\mathbf{x}.$$

Expanding $L(\mathbf{w}) = \|\mathbf{X}\mathbf{w} - \mathbf{y}\|^2 = \mathbf{w}^\top(\mathbf{X}^\top\mathbf{X})\mathbf{w} - 2(\mathbf{X}^\top\mathbf{y})^\top\mathbf{w} + \mathbf{y}^\top\mathbf{y}$ and applying these rules gives

$$\nabla L(\mathbf{w}) = 2\mathbf{X}^\top\mathbf{X}\mathbf{w} - 2\mathbf{X}^\top\mathbf{y} = 2\mathbf{X}^\top(\mathbf{X}\mathbf{w} - \mathbf{y}). \tag{1.22}$$

In words, each component of the gradient is twice the dot product of the corresponding feature column of $\mathbf{X}$ with the residual vector $\mathbf{r} = \mathbf{X}\mathbf{w} - \mathbf{y}$; at the optimum $\nabla L(\mathbf{w}^*) = \mathbf{0}$, the prediction errors are orthogonal to every feature.

1.4.3 The Chain Rule

Deep networks compose many nested layers $y = f_L(f_{L-1}(\cdots f_1(\mathbf{x})\cdots))$; differentiating compositions without expanding them relies on the *chain rule*.

For scalar functions $z = f(u)$ and $u = g(x)$, rates of change multiply: $\frac{dz}{dx} = \frac{dz}{du}\frac{du}{dx}$. More generally, if $z = f(u_1, \dots, u_m)$ where each $u_j = g_j(x_1, \dots, x_n)$, rates multiply along each path $x_i \to u_j \to z$ and sum across all parallel paths:

$$\frac{\partial z}{\partial x_i} = \sum_{j=1}^{m} \frac{\partial z}{\partial u_j}\frac{\partial u_j}{\partial x_i} \quad \Longrightarrow \quad \nabla_{\mathbf{x}} z = \mathbf{J}^\top \nabla_{\mathbf{u}} z, \tag{1.23}$$

where $\mathbf{J} \in \mathbb{R}^{m \times n}$ is the *Jacobian matrix* of $\mathbf{u}$ with respect to $\mathbf{x}$, with entries $J_{ji} = \frac{\partial u_j}{\partial x_i}$. For an affine map $\mathbf{u} = \mathbf{A}\mathbf{x} - \mathbf{y}$, the Jacobian is simply $\mathbf{J} = \mathbf{A}$.

A small case is worth working by hand. Let $z = u^2 + v$ with $u = x_1 x_2$ and $v = x_2^2$, evaluated at $(x_1, x_2) = (1, 2)$. The input x_2 reaches z along two paths, one through u and one through v (Figure 1.25). Each path contributes the product of its local derivatives, and the two contributions add: $\partial z/\partial x_2 = 4 \cdot 1 + 1 \cdot 4 = 8$.

Decomposing our line loss $L(\mathbf{w}) = \|\mathbf{X}\mathbf{w} - \mathbf{y}\|^2$ into two stages, namely (1) inner residuals $\mathbf{r} = \mathbf{X}\mathbf{w} - \mathbf{y}$ with Jacobian $\mathbf{J} = \mathbf{X}$ and (2) outer squared norm $L = \|\mathbf{r}\|^2$ with gradient $\nabla_{\mathbf{r}} L = 2\mathbf{r}$, the multivariate chain rule recovers $\nabla_{\mathbf{w}} L = \mathbf{X}^\top(2\mathbf{r})$ in two passes:

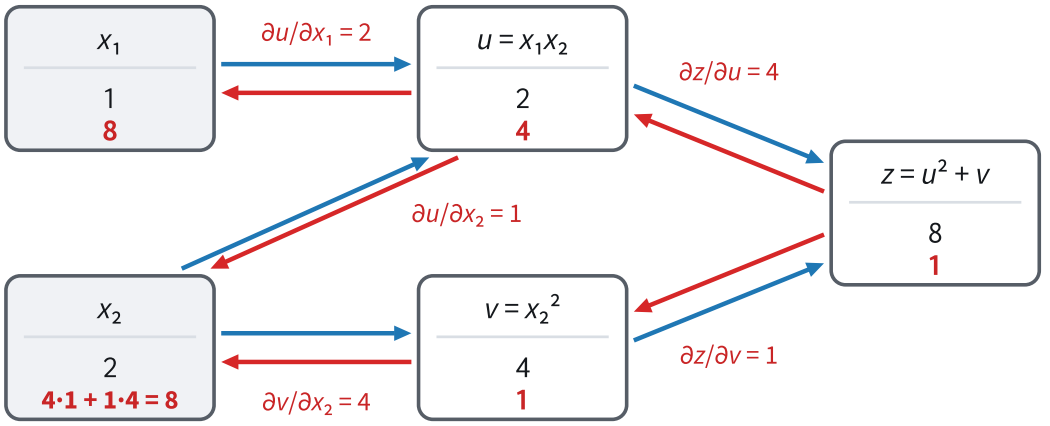


Figure 1.25: The chain rule for $z = u^2 + v$ with $u = x_1 x_2$ and $v = x_2^2$ at $(x_1, x_2) = (1, 2)$. Each box shows the value of a variable (black) and the derivative of z with respect to it (red), and each red arrow carries a local derivative. Derivatives multiply along a path and add where two paths meet, as they do at x_2.

```
r = X @ w - y    # forward: errors, from which the loss is r @ r
grad_r = 2 * r   # backward: gradient of the loss with respect to r
X.T @ grad_r     # backward: gradient with respect to w
```

```
array([2., 4.])
```

The *forward pass* computes $\mathbf{r} = \mathbf{X}\mathbf{w} - \mathbf{y}$ and $L = \mathbf{r}^\top \mathbf{r}$; the *backward pass* (*backpropagation*) multiplies $2\mathbf{r}$ by $\mathbf{X}^\top$, computing the exact gradient with respect to all parameters at roughly the cost of one forward evaluation. In Section 1.5, we hand this mechanical backward pass over to the framework's automatic differentiation engine.

1.4.4 Summary

The derivative $f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$ gives the slope of the tangent line and the local linear approximation $f(x + h) \approx f(x) + f'(x)h$. For $f : \mathbb{R}^n \to \mathbb{R}$, the gradient $\nabla_{\mathbf{x}} f(\mathbf{x})$ collects all partial derivatives, satisfies $f(\mathbf{x} + \mathbf{h}) \approx f(\mathbf{x}) + \nabla f(\mathbf{x})^\top \mathbf{h}$, and points in the direction of steepest ascent (perpendicular to level sets), motivating gradient descent $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla L(\mathbf{w})$. Unlike finite differences, which cost $O(n)$ evaluations and lose precision to catastrophic cancellation ($\sim \sqrt{\epsilon}$), the multivariate chain rule $\nabla_{\mathbf{x}} z = \mathbf{J}^\top \nabla_{\mathbf{u}} z$ propagates exact gradients backward from the loss to all parameters in a single backward pass (*backpropagation*).

1.4.5 Exercises

1. **Differentiating x^x when both base and exponent vary.** For $x > 0$, consider the function $f(x) = x^x$, in which x appears in both the base and the exponent.
    1. Explain why applying the power rule alone (treating the exponent as constant to get $x \cdot x^{x-1} = x^x$) and applying the exponential rule alone (treating the base as

constant to get $x^x \log x$) are both incorrect.
2. Rewrite $x^x = e^{x \log x}$ and apply the chain rule and product rule to find $f'(x)$. Notice that the true derivative is the *sum* of the two naive answers from Part 1; explain why this happens using the two-variable chain rule applied to $g(u, v) = u^v$ along $u = x, v = x$.

2. **Why squared error yields the mean and absolute error yields the median.** Suppose a single-parameter model predicts the constant value $w \in \mathbb{R}$ for n training examples with scalar labels $y_1, \dots, y_n$.
   1. Differentiate the sum of squared errors $L_2(w) = \sum_{i=1}^{n} (w - y_i)^2$ with respect to w, set the derivative to zero, and prove that the unique minimizer is the sample *mean* $w^* = \frac{1}{n} \sum_{i=1}^{n} y_i$.
   2. Now consider the sum of absolute errors $L_1(w) = \sum_{i=1}^{n} |w - y_i|$. Show that at any point w not equal to one of the labels y_i, the derivative $L_1'(w)$ equals the number of labels strictly less than w minus the number of labels strictly greater than w. At which values of w does $L_1'(w)$ fail to exist? Conclude that $L_1(w)$ is minimized by any sample *median* of $\{y_1, \dots, y_n\}$.
   3. For the four labels $\{1, 1, 0, 10\}$, compute the L_2 minimizer (mean) and the L_1 minimizer (median). What happens to each minimizer if the outlier label 10 is corrupted to 100?

3. **Gradient of the Euclidean norm.** Compute the gradient $\nabla_{\mathbf{x}} f(\mathbf{x})$ of the unsquared Euclidean norm $f(\mathbf{x}) = \|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2}$ for any nonzero vector $\mathbf{x} \neq \mathbf{0}$, and verify that $\|\nabla_{\mathbf{x}} f(\mathbf{x})\|_2 = 1$ everywhere it exists. What happens to the gradient at the origin $\mathbf{x} = \mathbf{0}$, and why?

4. **Step-size stability of gradient descent.** Consider minimizing the isotropic quadratic bowl $f(\mathbf{x}) = \|\mathbf{x}\|_2^2$ via the gradient descent update $\mathbf{x}_{k+1} = \mathbf{x}_k - \eta \nabla f(\mathbf{x}_k)$ with learning rate $\eta > 0$.
   1. Starting from $\mathbf{x}_0 = (1, 1)^\top$, compute $\mathbf{x}_1$ and the new objective value $f(\mathbf{x}_1)$ after a single step for three choices of learning rate: $\eta = 0.1$, $\eta = 1.0$, and $\eta = 2.0$.
   2. Write $\mathbf{x}_{k+1}$ in closed form in terms of $\mathbf{x}_k$ and η. For what range of learning rates $\eta > 0$ does a gradient descent step strictly decrease $f(\mathbf{x})$ from any nonzero starting point $\mathbf{x}_k \neq \mathbf{0}$? Which special value of η jumps directly to the global minimum $\mathbf{x}^* = \mathbf{0}$ in a single step, and what happens when $\eta > 1$?
   3. Characterize the stationary points $\mathbf{x}$ at which a gradient descent step leaves the parameter vector unchanged ($\mathbf{x}_{k+1} = \mathbf{x}_k$), both for $f(\mathbf{x}) = \|\mathbf{x}\|_2^2$ and for a general differentiable objective f.

5. code **Central differences versus forward differences.** ● The one-sided *forward difference* $\frac{f(x+h) - f(x)}{h}$ analyzed in this section has truncation error $O(h)$. The symmetric *central difference*

$$\frac{f(x + h) - f(x - h)}{2h} \tag{1.24}$$

evaluates f at $x + h$ and $x - h$ so that the $O(h)$ quadratic terms cancel out. Write a central-difference version of `quotient_error` and evaluate it across our logarithmic grid of h values in `float64`:

1. For the quadratic loss $L(w) = 3w^2 - 4w + 2$ at $w = 1$, plot the forward-difference and central-difference error curves on the same log–log axes. Why does the central-difference error drop straight to machine epsilon ($\sim 10^{-16}$) even at large $h = 10^{-1}$ on a quadratic function?
2. Now repeat the comparison on a non-quadratic function, $f(x) = e^x$ at $x = 0$ (where $f'(0) = 1$). Compare the slope of the large-h truncation branch of the central-difference curve against the slope-1 forward-difference branch, identify the optimal step size h^* and minimum achievable error for each method, and explain both using the Taylor expansion $e^{\pm h} = 1 \pm h + \frac{h^2}{2} \pm \frac{h^3}{6} + \cdots$.

6. code **Gradient checking in `float64` versus `float32`.** ● Although finite differences are far too slow for training, central finite differences are the gold standard for unit-testing hand-derived or custom-kernel gradients.
   1. Write a function `central_gradient(f, x, h=1e-5)` that estimates $\nabla f(\mathbf{x})$ for $f : \mathbb{R}^n \to \mathbb{R}$ via central differences. Use it in `float64` to test the quadratic-form gradient identity $\nabla_{\mathbf{x}}(\mathbf{x}^\top \mathbf{A}\mathbf{x}) = (\mathbf{A} + \mathbf{A}^\top)\mathbf{x}$ on a random *non-symmetric* 5×5 matrix $\mathbf{A}$ at a random vector $\mathbf{x} \in \mathbb{R}^5$, and report the maximum relative error across the five coordinates.
   2. Now run the same check against the *buggy* formula $2\mathbf{A}\mathbf{x}$ (a common mistake made when forgetting that $\mathbf{A}$ is not symmetric). How large is the relative error now?
   3. Finally, repeat the check of the correct formula $(\mathbf{A} + \mathbf{A}^\top)\mathbf{x}$ in 32-bit `float32` using framework tensors across $h \in \{10^{-1}, 10^{-2}, \ldots, 10^{-6}\}$. Which h achieves the smallest relative error in `float32`, why does it differ from the forward-difference optimum $h^* \approx 10^{-4}$, and what relative-error threshold cleanly separates a correct gradient implementation from a buggy one in `float32`?

1.5 Automatic Differentiation

In Section 1.4, deriving the gradient $\nabla_{\mathbf{w}} L = \mathbf{X}^\top(2\mathbf{r}) = (2, 4)$ of our two-parameter line model by hand was straightforward, whereas doing so across dozens of layers and billions of parameters would be hopeless. Modern deep learning frameworks automate this via *automatic differentiation* (*autograd*): as a program evaluates a scalar loss, the framework traces the elementary operations into a *computational graph* and traverses that graph in reverse (*backpropagation*) to apply the chain rule mechanically. Remarkably, whether a model has two parameters or a billion, a single backward sweep computes the exact gradient with respect to *all* of them at only a small constant multiple of the forward-pass cost.

```
import torch
```

1.5.1 Computing Gradients

1.5.1.1 The Forward Pass

We first set up the line model $L(\mathbf{w}) = \|\mathbf{X}\mathbf{w} - \mathbf{y}\|^2$ at $\mathbf{w} = (1, 0)$ from Section 1.4 using framework tensors, writing only the forward pass:

```
X = torch.tensor([[1.0, 0.0], [1.0, 1.0], [1.0, 2.0]])  # [1, input]
y = torch.tensor([1.0, 1.0, 0.0])
w = torch.tensor([1.0, 0.0])

def loss(w):
    return ((X @ w - y) ** 2).sum()
```

Because recording operations and saving intermediate activations consumes memory, the framework tracks only operations that depend on parameters explicitly marked for gradients.

Calling `w.requires_grad_(True)` marks `w` in place so PyTorch tracks operations involving `w` and stores the resulting gradient in `w.grad`:

```
w.requires_grad_(True)
```

```
tensor([1., 0.], requires_grad=True)
```

We now evaluate `L = loss(w)`, and PyTorch automatically records the sequence of operations connecting `w` to `L`:

```
L = loss(w)
L
```

```
tensor(1., grad_fn=<SumBackward0>)
```

Notice `grad_fn=<SumBackward0>` on `L`, recording the final operation in the computational graph.

The recorded trace forms the *computational graph* shown in Figure 1.26, breaking the loss into four primitive operations: $\hat{\mathbf{y}} = \mathbf{X}\mathbf{w}$, $\mathbf{r} = \hat{\mathbf{y}} - \mathbf{y}$, $\mathbf{s} = \mathbf{r} \odot \mathbf{r}$, and $L = \sum_i s_i$.

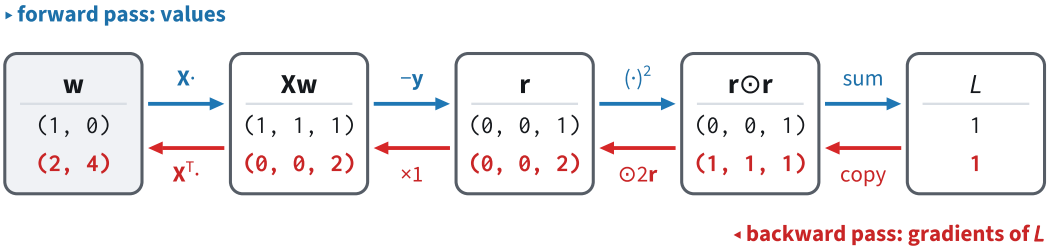


Figure 1.26: The computational graph of $L = \|\mathbf{X}\mathbf{w} - \mathbf{y}\|^2$ at $\mathbf{w} = (1, 0)$, with one node for $\mathbf{w}$ and for each value computed from it; the constants $\mathbf{X}$ and $\mathbf{y}$ are not drawn, and $\mathbf{r} \odot \mathbf{r}$ is the elementwise square. Each node shows its value (black) and the gradient of L with respect to it (red). The forward pass (blue) runs from left to right; the backward pass (red) starts from 1 at L and applies the local rule of each operation, from right to left.

1.5.1.2 The Backward Pass

Triggering the backward pass propagates sensitivities from $\partial L/\partial L = 1$ at the output back to $\mathbf{w}$ through each node's local derivative rule,

$$\nabla_{\mathbf{s}} L = \mathbf{1} \to \nabla_{\mathbf{r}} L = 2\mathbf{r} \to \nabla_{\hat{\mathbf{y}}} L = 2\mathbf{r} \to \nabla_{\mathbf{w}} L = \mathbf{X}^\top (2\mathbf{r}), \tag{1.25}$$

reproducing $(2, 4)$:

`L.backward()` runs the backward pass and stores the gradient in `w.grad`.

```
L.backward()
w.grad
```

```
tensor([2., 4.])
```

Because local backward rules require intermediate activations from the forward pass (such as $\mathbf{r}$ and $\mathbf{X}$), training consumes substantially more memory than inference (Figure 1.27).

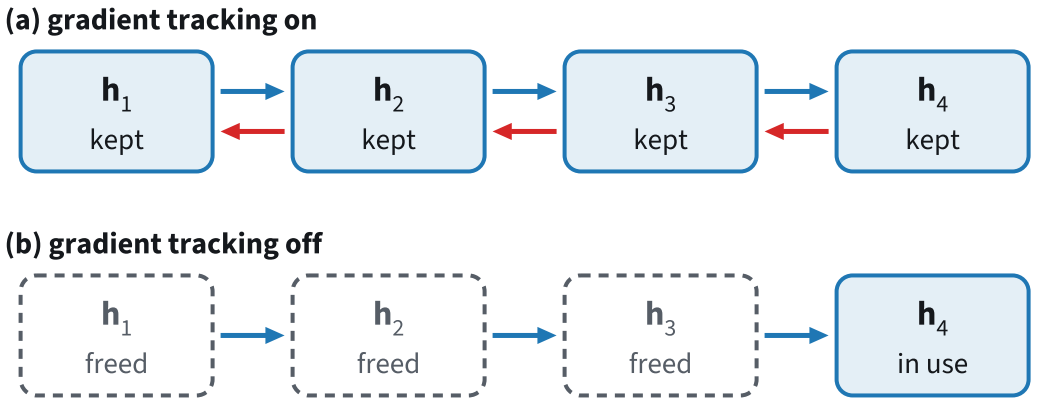


Figure 1.27: Memory during a forward pass through four layers with outputs $\mathbf{h}_1, \ldots, \mathbf{h}_4$. (a) When gradients are tracked, every output is kept, because the backward pass (red) will need it. (b) When they are not, as during inference, an output can be freed as soon as the next layer has used it.

1.5.1.3 Repeated Backward Passes

What happens when we evaluate the gradient a second time at the same parameter vector $\mathbf{w}$?

```
L = loss(w)
L.backward()
w.grad  # the new gradient was added to the old one
```

```
tensor([4., 8.])
```

In PyTorch, `w.grad` becomes `(4, 8)` because PyTorch *adds* each new gradient into `w.grad` by default, which makes summing gradients across micro-batches or multiple loss terms convenient. In a standard training loop, always clear the buffer before the next step via `w.grad.zero_()`.

1.5.1.4 Vector-Valued Functions

When an operation produces a vector of m outputs $\mathbf{u} \in \mathbb{R}^m$ from n inputs $\mathbf{x} \in \mathbb{R}^n$ with $m \times n$ Jacobian $\mathbf{J}$, a backward pass seeded with a weight vector $\mathbf{v} \in \mathbb{R}^m$ computes the transposed *vector–Jacobian product* $\mathbf{J}^\top \mathbf{v} = \nabla_{\mathbf{x}}(\mathbf{v}^\top \mathbf{u})$ without materializing $\mathbf{J}$. Passing $\mathbf{v} = \mathbf{1}$ on the per-example squared errors (r_1^2, r_2^2, r_3^2) differentiates their sum:

Calling `backward()` on a non-scalar tensor requires passing $\mathbf{v}$ explicitly via `gradient=torch.ones(3)` (equivalent to `losses.sum().backward()`).

```
w.grad.zero_()
losses = (X @ w - y) ** 2  # one loss per example
losses.backward(gradient=torch.ones(3))  # = losses.sum().backward()
w.grad
```

```
tensor([2., 4.])
```

1.5.2 Controlling the Graph

1.5.2.1 Detaching Values

Detaching a tensor creates a view or copy with the same numerical values but no link to its ancestors in the computational graph, so backpropagation treats it as a constant. Consider $u = x^2$ and $z = ux$ on $\mathbf{x} = (0, 1, 2, 3)$ (Figure 1.28): when u is attached, gradient flows along both paths to give $\partial z/\partial x = 3x^2$; when u is detached to c, differentiating $z = cx$ treats c as a constant and gives $\partial z/\partial x = c = x^2$ (essential whenever a network computes its own bootstrap target, as in deep Q-learning, Section 15.4).

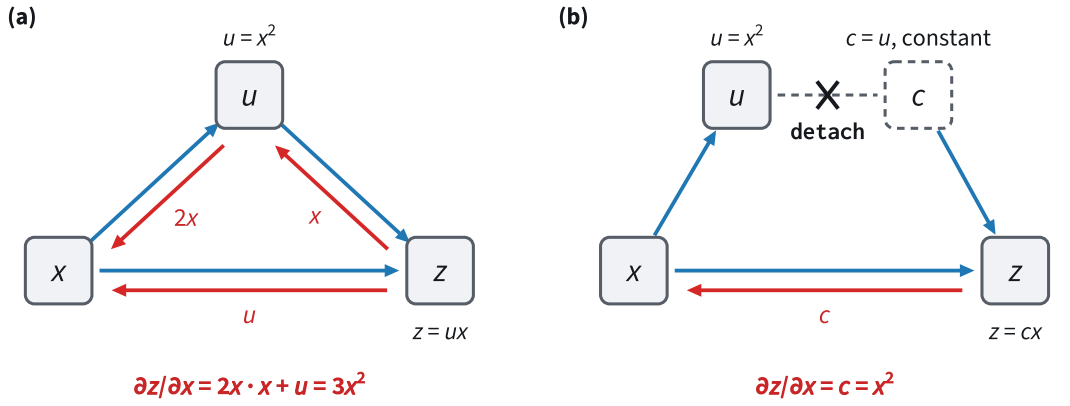


Figure 1.28: The product $z = ux$ with $u = x^2$, (a) attached and (b) detached. Blue arrows are operations of the forward pass; red arrows show the backward pass, labeled with the local derivatives it multiplies by. Detaching u creates a constant c with the same value and no connection to x.

`u.detach()` returns a tensor sharing `u`'s values but stripped of its history.

```
x = torch.arange(4.0, requires_grad=True)
u = x * x
c = u.detach()  # values of u, no history
z = c * x
z.sum().backward()
x.grad
```

```
tensor([0., 1., 4., 9.])
```

By contrast, if we compute `z = u * x` directly without detaching `u`, backpropagation flows through both factors of $z = x^3$ and returns $\partial z/\partial x = 3x^2 = (0, 3, 12, 27)$:

```
x.grad.zero_()
z = u * x
z.sum().backward()
x.grad
```

```
tensor([ 0.,  3., 12., 27.])
```

1.5.2.2 Disabling Gradient Tracking

During validation, inference, or parameter updates $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla L(\mathbf{w})$, we disable gradient tracking so those operations are not recorded on the loss graph:

`with torch.no_grad():` disables tracking (required when modifying a leaf tensor with `requires_grad=True` in place).

```
with torch.no_grad():  # nothing in this block is recorded
    w -= 0.1 * w.grad
    L = loss(w)
w, L
```

```
(tensor([ 0.8000, -0.4000], requires_grad=True), tensor(0.4000))
```

As in Section 1.4, one step with $\eta = 0.1$ moves $\mathbf{w}$ from $(1, 0)$ to $(0.8, -0.4)$ and lowers the loss from 1 to 0.4.

1.5.3 Differentiating Programs

1.5.3.1 Control Flow

Because autograd records the sequence of elementary operations that *actually executed* on a given input, it differentiates Python functions containing data-dependent `while` loops and `if`/`else` branches directly:

```
def f(a):
    b = a * 2
    while torch.linalg.vector_norm(b) < 1000:
        b = b * 2
    if b.sum() > 0:
        c = b
```

```
    else:
        c = 100 * b
    return c
```

For any nonzero scalar a (at $a = 0$ the `while` loop would never terminate), `f(a)` performs a fixed sequence of multiplications by 2 (and possibly 100), so `f(a) = k * a` is piecewise linear with local slope $k = f(a)/a$ (Figure 1.29). Specifically, $a = 0.3$ doubles 12 times to reach $0.3 \times 2^{12} = 1228.8 \geq 1000$ and takes the positive branch ($k = 2^{12} = 4096$), whereas $a = -0.75$ doubles 11 times to -1536 and takes the `else` branch ($k = 100 \times 2^{11} = 204{,}800$). Autograd recovers both slopes:

```
for value in (0.3, -0.75):
    a = torch.tensor(value, requires_grad=True)
    fa = f(a)
    fa.backward()
    print(f'a = {value:5.2f}   f(a) / a = {fa.item() / value:6.0f}   '
          f'gradient = {a.grad.item():6.0f}')
```

```
a =  0.30   f(a) / a =   4096   gradient =   4096
a = -0.75   f(a) / a = 204800   gradient = 204800
```

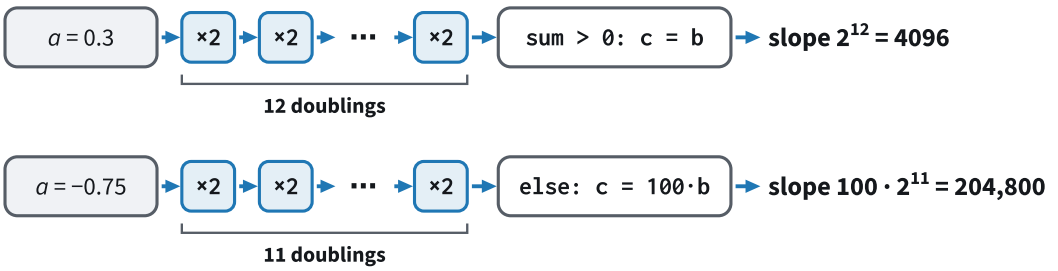


Figure 1.29: The graphs recorded by `f` for two inputs. For $a = 0.3$, `f` doubles a 12 times and the first branch runs; for $a = -0.75$, it doubles a 11 times and the second branch runs. Each graph is linear in a, with its own slope.

Keep in mind that autograd differentiates the *executed trace* rather than the abstract mathematical function: at a threshold point where the loop count jumps, $f(a)$ is discontinuous, yet autograd returns the slope of whichever branch ran.

1.5.4 Forward and Reverse Mode

For a mapping $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ with $m \times n$ Jacobian $\mathbf{J}$, the chain rule can be traversed in either direction (Figure 1.30; see Wengert (1964); Linnainmaa (1970) and Section 34.5):

- **Reverse mode (vector–Jacobian product, VJP):** A reverse sweep seeded with $\mathbf{v} \in \mathbb{R}^m$ at the outputs computes $\mathbf{J}^\top \mathbf{v}$; seeding with the j^{th} standard basis vector $\mathbf{v} = \mathbf{e}_j$ extracts row j of $\mathbf{J}$ (m sweeps for the full Jacobian).
- **Forward mode (Jacobian–vector product, JVP):** A forward sweep seeded with a tangent $\mathbf{d} \in \mathbb{R}^n$ at the inputs propagates derivatives forward alongside values to compute $\mathbf{Jd}$; seeding with $\mathbf{d} = \mathbf{e}_i$ extracts column i of $\mathbf{J}$ (n sweeps for the full Jacobian, with constant $2\times$ memory overhead).

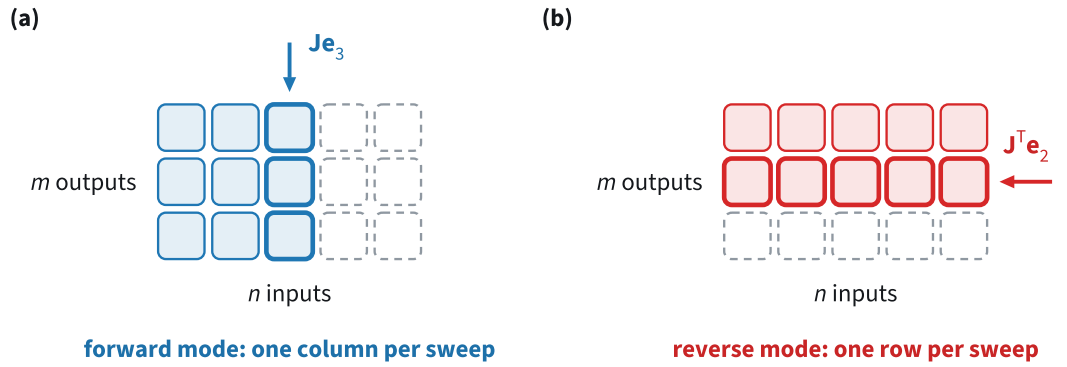


Figure 1.30: Two ways to fill the $m \times n$ Jacobian $\mathbf{J}$ of a function with n inputs and m outputs: (a) forward mode, n sweeps in all; (b) reverse mode, m sweeps in all.

PyTorch provides `torch.func.vjp`, `torch.func.jvp`, `torch.func.jacrev`, and `torch.func.jacfwd`.

A scalar loss function has $m = 1$ output and n parameters, so its Jacobian is a single row $\nabla_{\mathbf{w}} L^{\top}$. Reverse mode (*backpropagation*) computes all n components of the gradient in *one sweep*, whereas forward mode would require n sweeps, which explains why reverse mode is universal for training deep networks.

1.5.5 Summary

Automatic differentiation traces the elementary operations of a forward pass into a computational graph and applies local derivative rules in reverse to compute exact gradients, reproducing $\nabla_{\mathbf{w}} L = (2, 4)$ on our line model without hand-written derivative code. Detaching an intermediate value severs its history so backpropagation treats it as a constant, while disabling gradient tracking keeps parameter updates and inference evaluations off the graph and avoids caching forward activations. Because the graph records the operations that actually executed on a given input, autograd differentiates seamlessly through Python loops and conditionals. Finally, one forward-mode sweep computes a Jacobian–vector product $\mathbf{Jd}$ (one column per basis seed) whereas one reverse-mode sweep computes a vector–Jacobian product $\mathbf{J}^{\top}\mathbf{v}$ (one row per basis seed), so for a scalar loss ($m = 1$) a single reverse sweep yields the entire n-parameter gradient.

1.5.6 Exercises

1. code **Detached intermediates.** Let `u = x * x` and `c = u.detach()`, and consider `z = c * c`, `z = u * c`, and `z = u * u`. For each `z`, recompute `u` and `c`, reset `x.grad`, and call `backward` on the sum of the entries of `z`. Before running any code, predict the outcome of each call, including the resulting `x.grad`. Check your predictions, and explain the ratio between the gradients for `z = u * c` and `z = u * u`.

2. code **Manual parameter updates.** Perform the gradient step of this section as `w = w - 0.1 * w.grad`, outside `torch.no_grad()`, and inspect `w.requires_grad`, `w.grad_fn`, `w.is_leaf`, and `w.grad`. Explain what the update has done to the graph, predict what a second forward and backward pass leaves in `w.grad`, and check.

3. code **From scalar to vector input.** Replace the scalar input `a` of the function `f` defined

in Section 1.5.3.1 by a random vector of length 5, and differentiate the sum of the entries of `fa = f(a)` with respect to `a`.

   1. Express `f(a)` in terms of `a`, and give the Jacobian of `f` at `a` and the gradient of the sum of the entries of `fa`.
   2. Compute this gradient with automatic differentiation and check whether it still equals `fa / a` elementwise.
   3. Describe the set of inputs `a` at which `f` jumps. Which part of this set has no counterpart for a scalar input?

4. code **Programs versus functions.** Automatic differentiation differentiates the operations that a program executes, not the function that the program computes.

   1. The functions $|x|$ and $\sqrt{x^2}$ are equal. Compute the derivative of each at $x = 0$ by automatic differentiation, using the framework's absolute value and square root. Explain both results from the local derivatives of the operations involved. Neither function has a derivative at 0; which of the two results does more harm in training, and why?
   2. Write a program that returns x^2 for every input x but whose automatic derivative at $x = 0$ is 1 instead of 0. Explain why automatic differentiation cannot detect the difference.

5. **Cost of second derivatives.** ● The counting argument in Section 1.5.4 puts the cost of the gradient of a scalar loss with n parameters at one reverse sweep. Use the same argument to estimate, in units of function evaluations, the cost of the full Hessian (the $n \times n$ matrix of second derivatives) and the cost of a single Hessian–vector product. Explain the difference between the two costs.

1.6 Probability and Statistics

Machine learning is all about reasoning under uncertainty: predicting how likely a patient is to suffer a heart attack, judging whether a sensor reading is anomalous, or telling whether an accuracy difference from 0.90 ($230/256$ test images) to 0.87 on the next minibatch reflects a real change or random sampling fluctuation.

Probability theory's cleanest thought experiment is still the coin toss: if a coin lands on heads 56 times in 100 tosses, should we suspect that it is biased? Even a fair coin rarely lands on heads exactly 50 times, so we must ask whether a deviation of 6 heads is larger than chance routinely produces. *Probability* reasons forward from a known model (a fair coin) to the likelihood of outcomes, while *statistics* reasons backward from observed data (56 heads in 100 tosses) to infer properties of the underlying process.

In this section, we simulate coin tosses to watch sampling fluctuations unfold, introduce sample spaces, events, random variables, joint distributions, and independence, and derive the *standard error* $\sigma/\sqrt{n}$ via expectation and variance to settle the coin question. We then develop conditional probability and Bayes' theorem on a two-step medical diagnostic test.

```
import random
from torch.distributions.multinomial import Multinomial
```

1.6.1 Sampling

1.6.1.1 Simulated Coin Tosses

For a *fair* coin, symmetry between heads and tails implies $P(\text{heads}) = 0.5$. Drawing realizations from a random process is called *sampling*: below, we simulate 100 tosses first by testing whether uniform draws from $[0, 1)$ exceed 0.5, and then using our framework's *multinomial* sampler with $n = 100$ and probabilities $(0.5, 0.5)$:

```
num_tosses = 100
heads = sum([random.random() > 0.5 for _ in range(num_tosses)])
tails = num_tosses - heads
print("heads, tails: ", [heads, tails])
```

```
heads, tails:  [45, 55]
```

```
fair_probs = torch.tensor([0.5, 0.5])
Multinomial(100, fair_probs).sample()
```

```
tensor([48., 52.])
```

1.6.1.2 Frequencies and Probabilities

Dividing the observed counts by n gives the empirical *frequency* $\hat{p}$, an *estimator* of the true probability $p = P(\text{heads}) = 0.5$. Repeating the 100-toss experiment five times simultaneously produces a 5×2 matrix of empirical frequencies that fluctuate noticeably around 0.5:

```
Multinomial(100, fair_probs).sample((5,)) / 100
```

```
tensor([[0.5000, 0.5000],
        [0.4500, 0.5500],
        [0.5000, 0.5000],
        [0.4900, 0.5100],
        [0.5800, 0.4200]])
```

Increasing the sample size a hundredfold to $n = 10{,}000$ tosses shrinks the typical sampling error by $\sqrt{100} = 10$, pulling the empirical head and tail frequencies much closer to $(0.5, 0.5)$:

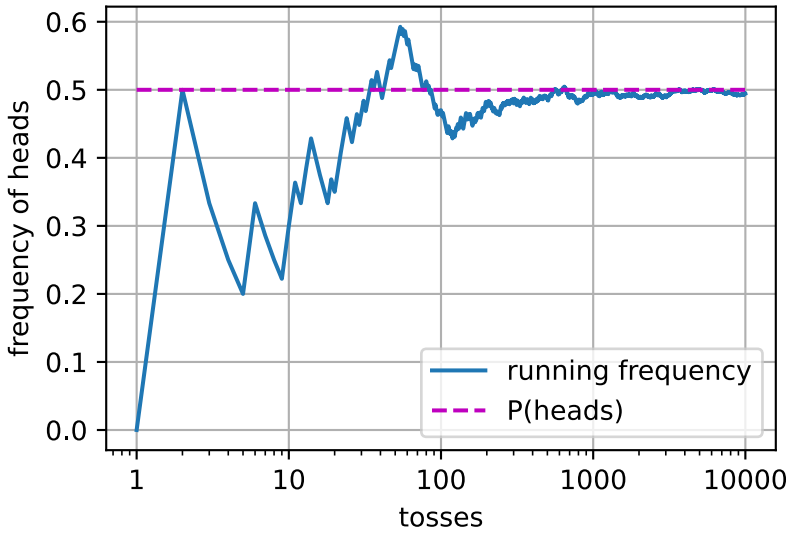
```
Multinomial(10000, fair_probs).sample() / 10000
```

```
tensor([0.4959, 0.5041])
```

Tracking the running frequency after every toss from 1 to $10{,}000$ via `cumsum` illustrates the *law of large numbers* (Section 36.5), which guarantees that $\hat{p} \to p$ as $n \to \infty$:

```
n = 10000
tosses = (torch.rand(n) < 0.5).to(torch.float32)  # 1 is heads, 0 is tails
estimates = tosses.cumsum(dim=0) / torch.arange(1, n + 1)
```

```
d2l.plot(torch.arange(1, n + 1), [estimates, torch.full((n,), 0.5)],
         'tosses', 'frequency of heads',
         legend=['running frequency', 'P(heads)'], xscale='log',
         figsize=(4.5, 3))
```



To know *how fast* $\hat{p}$ settles—and whether $56/100$ is suspicious—we now formalize random variables and their spread.

1.6.2 Events and Random Variables

1.6.2.1 Sample Spaces and Events

The set of all possible outcomes of a random experiment is its *sample space* $\mathcal{S}$ (for one die roll, $\mathcal{S} = \{1, 2, 3, 4, 5, 6\}$). An *event* is any subset $\mathcal{A} \subseteq \mathcal{S}$ (such as "rolling an even number," $\mathcal{A} = \{2, 4, 6\}$, or "at least 5," $\mathcal{B} = \{5, 6\}$). A *probability* function P assigns a real number to every event $\mathcal{A} \subseteq \mathcal{S}$ subject to Kolmogorov's three axioms (Kolmogorov 1933):

- **Nonnegativity:** $P(\mathcal{A}) \geq 0$;
- **Normalization:** $P(\mathcal{S}) = 1$;
- **Countable additivity:** for mutually exclusive events $\mathcal{A}_1, \mathcal{A}_2, \ldots$ ($\mathcal{A}_i \cap \mathcal{A}_j = \emptyset$ for $i \neq j$), $P(\bigcup_i \mathcal{A}_i) = \sum_i P(\mathcal{A}_i)$.

From these axioms follow the complement rule $P(\mathcal{A}') = 1 - P(\mathcal{A})$, $P(\emptyset) = 0$, and the inclusion–exclusion formula (Figure 1.31):

$$P(\mathcal{A} \cup \mathcal{B}) = P(\mathcal{A}) + P(\mathcal{B}) - P(\mathcal{A} \cap \mathcal{B}). \tag{1.26}$$

1.6.2.2 Random Variables and Distributions

A *random variable* X maps outcomes in $\mathcal{S}$ to numerical values (for example, the number of heads in 100 coin tosses maps 2^{100} sequences to $\{0, \ldots, 100\}$). A *discrete* random variable

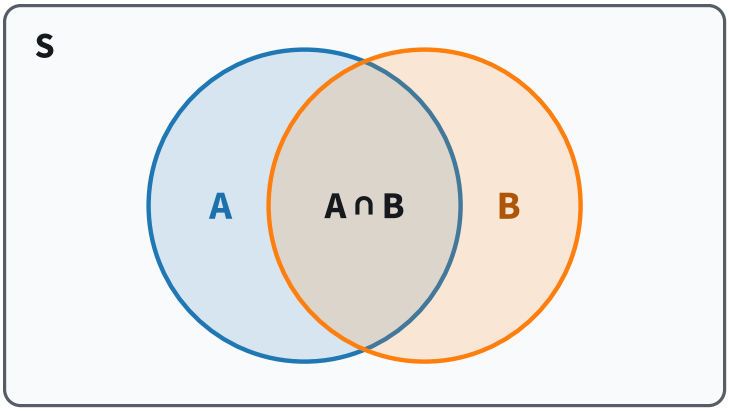


Figure 1.31: Two events $\mathcal{A}$ and $\mathcal{B}$ in a sample space $\mathcal{S}$. The sum $P(\mathcal{A}) + P(\mathcal{B})$ counts the overlap $\mathcal{A} \cap \mathcal{B}$ twice.

assigns probability mass $P(X = v)$ to each value v (its *distribution* $P(X)$). A *continuous* random variable (such as a person's exact height) is described instead by a *probability density* $p(x)$ (Figure 1.32), where the probability of landing in $[a, b]$ is the area under the density curve, $P(a \le X \le b) = \int_a^b p(x)\,dx$ (Section 36.1).

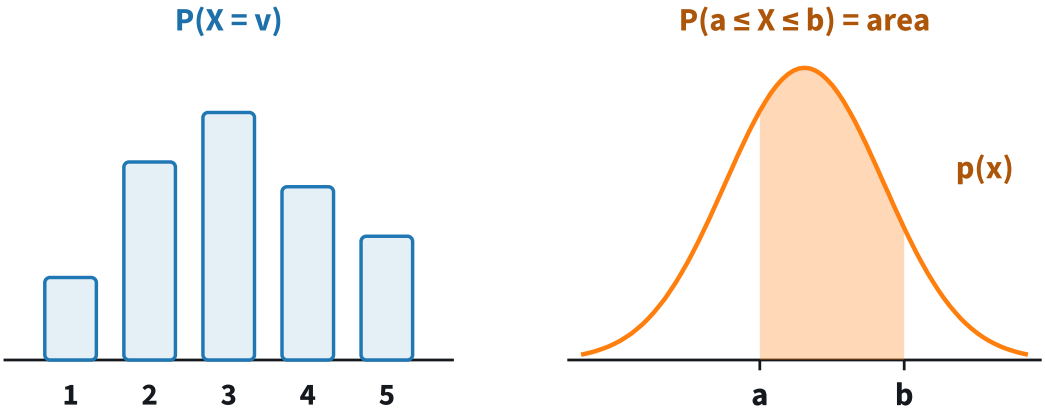


Figure 1.32: A discrete random variable puts probability on individual values (left). A continuous random variable has a density, and the probability of an interval is the area under the density (right).

1.6.2.3 Joint, Marginal, and Independent Distributions

The *joint distribution* $P(X = x, Y = y)$ of two random variables gives the probability that $X = x$ and $Y = y$ occur together. For example, for rain (X) and carrying an umbrella (Y) on a random day:

$P(X,Y)$	umbrella	no umbrella
rain	0.25	0.05
no rain	0.10	0.60

Summing out one variable recovers the *marginal distribution* of the other (*marginalization*, Figure 1.33):

$$P(X = x) = \sum_{y} P(X = x, Y = y), \tag{1.27}$$

so $P(\text{rain}) = 0.25 + 0.05 = 0.30$ and $P(\text{umbrella}) = 0.25 + 0.10 = 0.35$.

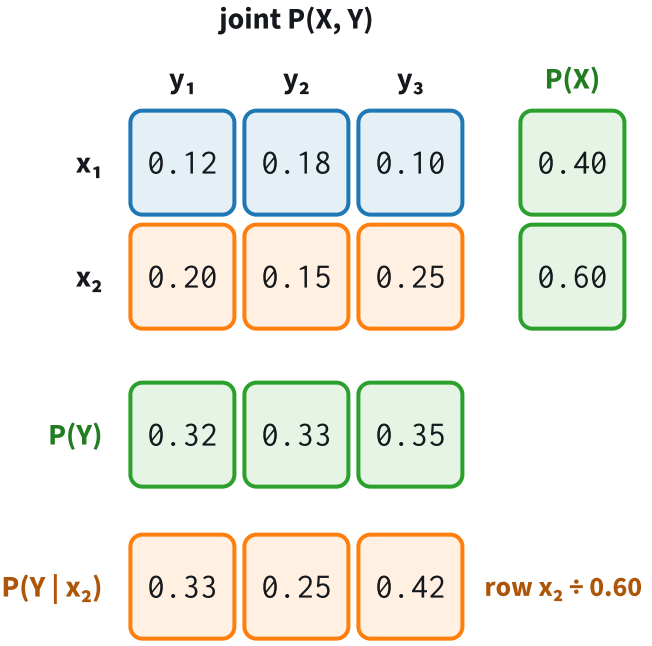


Figure 1.33: A joint distribution $P(X,Y)$ as a table. Summing a row or a column gives a marginal distribution. Dividing a row by its sum gives a conditional distribution, here $P(Y \mid X = x_2)$.

Two random variables X and Y are *independent* ($X \perp Y$) if their joint distribution factorizes into the product of their marginals:

$$P(X,Y) = P(X)P(Y). \tag{1.28}$$

Successive coin tosses are independent, whereas rain and umbrellas are not ($0.25 \neq 0.30 \cdot 0.35 = 0.105$).

1.6.3 Expectation and Variance

1.6.3.1 Expectation

The *expectation* (or *mean*) of a discrete random variable X is its probability-weighted average:

$$\mathbb{E}[X] = \sum_{x} x\, P(X = x), \tag{1.29}$$

and for any function f, $\mathbb{E}[f(X)] = \sum_x f(x)\, P(X = x)$. For an investment returning $0\times$ with probability 0.5, $2\times$ with probability 0.4, and $10\times$ with probability 0.1, $\mathbb{E}[X] = 0.5(0) + 0.4(2) + 0.1(10) = 1.8$. For a coin-toss indicator $X = \mathbf{1}(\text{heads}) \in \{0, 1\}$, $\mathbb{E}[X] = 1 \cdot p + 0 \cdot (1 - p) = p$.

Expectation is always *linear*: $\mathbb{E}[aX + bY] = a\,\mathbb{E}[X] + b\,\mathbb{E}[Y]$ holds whether or not X and Y are independent. Consequently, the sample frequency $\hat{p} = \frac{1}{n}\sum_{i=1}^{n} X_i$ satisfies $\mathbb{E}[\hat{p}] = \frac{1}{n}\sum_{i=1}^{n}\mathbb{E}[X_i] = p$, making $\hat{p}$ an *unbiased* estimator of p.

1.6.3.2 Variance and Sums of Independent Variables

To measure spread around $\mu = \mathbb{E}[X]$, the *variance* averages the squared deviations, and the *standard deviation* $\sigma = \sqrt{\mathrm{Var}(X)}$ restores the original units:

$$\mathrm{Var}(X) = \mathbb{E}\left[(X - \mu)^2\right] = \mathbb{E}[X^2] - \mu^2, \qquad \mathrm{Var}(aX + b) = a^2\,\mathrm{Var}(X). \tag{1.30}$$

For our investment, $\mathbb{E}[X^2] = 0.5(0) + 0.4(4) + 0.1(100) = 11.6$, so $\mathrm{Var}(X) = 11.6 - 1.8^2 = 8.36$ ($\sigma \approx 2.9$). For a single coin-toss indicator $X \in \{0, 1\}$, $X^2 = X$, so $\mathrm{Var}(X) = p - p^2 = p(1 - p)$, which reaches its maximum $1/4$ at $p = 0.5$. When estimating variance from n observations around their sample mean as in Section 1.2, dividing by $n - 1$ (*Bessel's correction*) compensates for the slight downward bias of dividing by n.

For a sum of two random variables, expanding $\mathbb{E}[(X - \mu_X + Y - \mu_Y)^2]$ gives

$$\begin{aligned} \mathrm{Var}(X + Y) &= \mathrm{Var}(X) + \mathrm{Var}(Y) + 2\,\mathrm{Cov}(X, Y), \\ \mathrm{Cov}(X, Y) &= \mathbb{E}\left[(X - \mu_X)(Y - \mu_Y)\right], \end{aligned}$$

where the *covariance* $\mathrm{Cov}(X, Y)$ and dimensionless *correlation* $\rho = \mathrm{Cov}(X, Y)/(\sigma_X \sigma_Y) \in [-1, 1]$ measure linear co-movement (Figure 1.34). More generally, for a random vector $\mathbf{x} \in \mathbb{R}^d$ with mean $\mu = \mathbb{E}[\mathbf{x}]$, collecting all pairwise covariances $\Sigma_{ij} = \mathrm{Cov}(x_i, x_j)$ forms the symmetric $d \times d$ *covariance matrix* $\Sigma = \mathbb{E}[(\mathbf{x} - \mu)(\mathbf{x} - \mu)^\top]$, and any linear combination $\mathbf{v}^\top \mathbf{x}$ has variance $\mathrm{Var}(\mathbf{v}^\top \mathbf{x}) = \mathbf{v}^\top \Sigma \mathbf{v}$.

When X and Y are *independent*, their expectation factorizes so $\mathrm{Cov}(X, Y) = \mathbb{E}[X - \mu_X]\mathbb{E}[Y - \mu_Y] = 0$, and **variances of independent variables add**:

$$\mathrm{Var}(X + Y) = \mathrm{Var}(X) + \mathrm{Var}(Y). \tag{1.31}$$

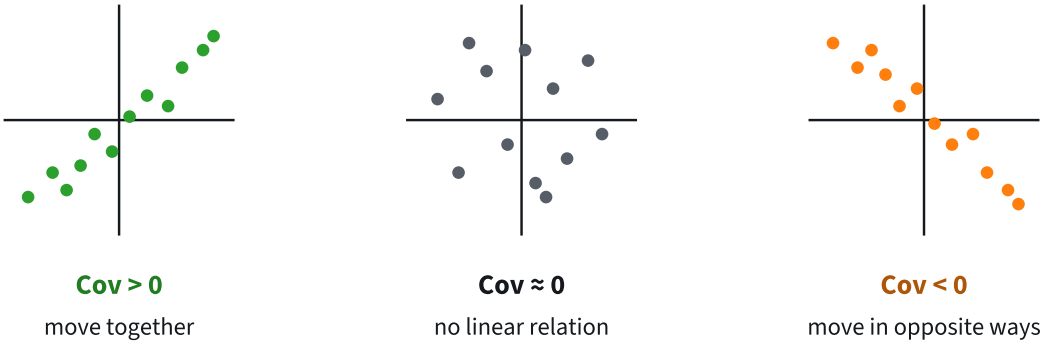


Figure 1.34: Samples of three pairs of random variables, with positive, near-zero, and negative covariance.

Zero covariance does not, however, imply independence: if X takes $-1, 0, +1$ with equal probability and $Y = X^2$, then Y is completely determined by X, yet by symmetry $\mathbb{E}[X] = 0$ and $\mathbb{E}[XY] = \mathbb{E}[X^3] = 0$, giving $\mathrm{Cov}(X, Y) = 0$.

1.6.4 Estimation Error

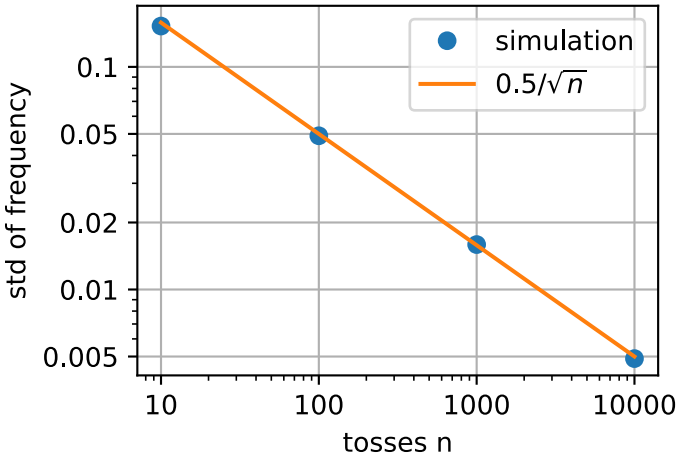
1.6.4.1 The Standard Error of a Frequency

Because n independent coin tosses each have variance $p(1-p)$, their sum has variance $np(1-p)$, and dividing by n scales the variance by $1/n^2$:

$$\mathrm{Var}(\hat{p}) = \frac{p(1-p)}{n}. \tag{1.32}$$

Taking the square root gives the *standard error* of $\hat{p}$: $\sqrt{p(1-p)/n}$, which for a fair coin ($p = 0.5$) is $0.5/\sqrt{n}$ (0.05 at $n = 100$; 0.005 at $n = 10{,}000$). Because of the $1/\sqrt{n}$ rate, halving the error requires $4\times$ the data. More generally, the average of any n independent draws with variance σ^2 has standard error $\sigma/\sqrt{n}$. Simulating 1000 repetitions at each $n \in \{10, 100, 1000, 10000\}$ confirms the $0.5/\sqrt{n}$ line of slope $-1/2$:

```
ns = [10, 100, 1000, 10000]
stds = [float((torch.rand(1000, n) < 0.5).to(torch.float32)
               .mean(dim=1).std()) for n in ns]
d2l.plot(ns, [stds, [0.5 / n ** 0.5 for n in ns]], 'tosses n',
         'std of frequency', legend=['simulation', r'$0.5/\sqrt{n}$'],
         xscale='log', yscale='log', fmts=('o', '-'))
```



1.6.4.2 Testing for Bias and Uncertainty

We can now answer our coin question. In 100 tosses, $\hat{p} = 0.56$ is $0.06/0.05 = 1.2$ standard errors from 0.5; in $10{,}000$ tosses, the same frequency $\hat{p} = 0.56$ ($5{,}600$ heads) is $0.06/0.005 = 12$ standard errors away. Simulating 1000 fair-coin experiments shows how often a fair coin strays at least that far (estimating the *p-value*):

```
def fraction_as_far(heads, n, num_experiments=1000):
    """Fraction of fair-coin experiments at least as far from n/2 as heads."""
    counts = (torch.rand(num_experiments, n) < 0.5).sum(dim=1)
    far = (counts - n / 2).abs() >= abs(heads - n / 2)
    return far.to(torch.float32).mean()

fraction_as_far(56, 100), fraction_as_far(5600, 10000)
```

```
(tensor(0.2820), tensor(0.))
```

Roughly a quarter of fair 100-toss runs deviate by 6 or more heads, so $56/100$ is no evidence of bias, whereas zero out of 1000 fair 10,000-toss runs come anywhere near $5{,}600$. Two general theorems quantify these tail probabilities: *Chebyshev's inequality* (Section 36.1) guarantees $P(|X - \mu| \geq k\sigma) \leq 1/k^2$ for any distribution with finite variance, while the *central limit theorem* (Section 36.3) shows that averages of independent draws approach a Gaussian bell curve, landing within 2 standard errors about 95% of the time. The same arithmetic governs model evaluation: a classifier with true accuracy 0.9 has standard error $\sqrt{0.9 \cdot 0.1/256} \approx 0.019$ on a single 256-image minibatch (so $\pm 2\%$ swings are ordinary noise), versus 0.003 across a $10{,}000$-image test set.

Notice finally that collecting more data shrinks *epistemic uncertainty* (uncertainty about the model parameter p, decaying as $1/\sqrt{n}$), but never reduces *aleatoric uncertainty* (the intrinsic 50/50 randomness of the next coin toss itself, or an inherently ambiguous image) (Der Kiureghian and Ditlevsen 2009).

1.6.5 Conditional Probability

1.6.5.1 Conditioning, Bayes' Theorem, and Conditional Independence

Restricting attention to outcomes where $X = x$ ($P(X = x) > 0$) and renormalizing defines the *conditional probability* of $Y = y$ given $X = x$:

$$P(Y = y \mid X = x) = \frac{P(X = x, Y = y)}{P(X = x)} \quad \Longrightarrow \quad P(X, Y) = P(Y \mid X)P(X). \tag{1.33}$$

In our weather table, conditioning on rain rescales the rain row $(0.25, 0.05)$ by 0.30 to give $P(\text{umbrella} \mid \text{rain}) = 0.25/0.30 \approx 0.83$ (versus 0.35 unconditionally). Writing the joint probability in both orders, $P(X, Y) = P(Y \mid X)P(X) = P(X \mid Y)P(Y)$, and dividing by $P(Y) = \sum_x P(Y \mid X = x)P(X = x)$ yields *Bayes' theorem* (Bayes 1763):

$$P(X \mid Y) = \frac{P(Y \mid X)P(X)}{P(Y)}. \tag{1.34}$$

For a hypothesis H and observed data D, Bayes' theorem updates the *prior* $P(H)$ via the *likelihood* $P(D \mid H)$ and evidence $P(D)$ to produce the *posterior* $P(H \mid D) \propto P(D \mid H)P(H)$.

Two variables X and Y are *conditionally independent* given Z ($X \perp Y \mid Z$) if $P(X, Y \mid Z) = P(X \mid Z)P(Y \mid Z)$. Conditioning can either remove or introduce dependence (Figure 1.35): observing a *common cause* Z (such as a child's age) removes the correlation between shoe size and reading level, and observing the present state Y in a *Markov chain* $X \to Y \to Z$ renders past X and future Z independent; conversely, conditioning on a *common effect* Z (*explaining away*) couples two independent causes, as when learning that a hospitalized patient has a broken bone lowers the probability they were admitted for lung cancer.

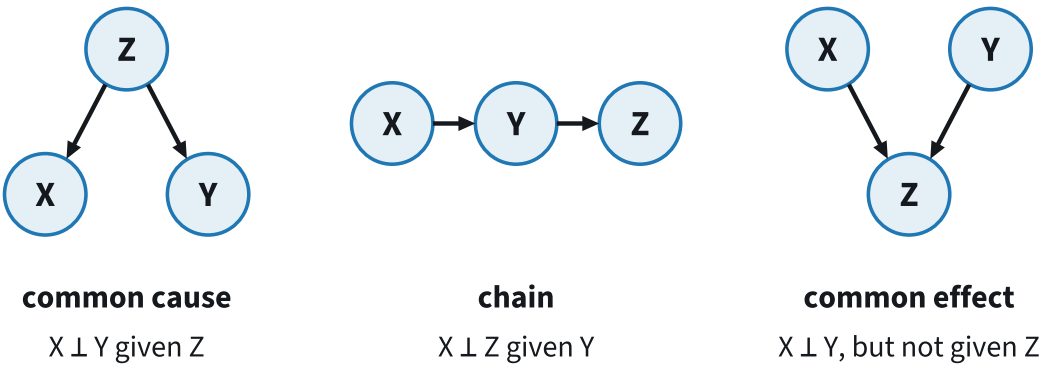


Figure 1.35: Conditioning changes dependence. A common cause Z makes X and Y dependent, and observing Z removes the dependence. In a chain, X and Z are independent given Y. A common effect Z makes its independent causes dependent once Z is observed.

1.6.5.2 Example: Diagnostic Testing

Suppose a patient takes an HIV screening test $D_1 \in \{0, 1\}$ with true HIV status $H \in \{0, 1\}$. The test has 100% sensitivity and a 1% false-positive rate:

$P(D_1 \mid H)$	$H = 1$ (HIV)	$H = 0$ (healthy)
test positive, $D_1 = 1$	1.00	0.01
test negative, $D_1 = 0$	0.00	0.99

If the population prevalence is $P(H = 1) = 0.0015$ ($P(H = 0) = 0.9985$) and the patient tests positive ($D_1 = 1$), marginalizing gives $P(D_1 = 1) = 0.01(0.9985) + 1(0.0015) = 0.011485$, and Bayes' theorem gives

$$P(H = 1 \mid D_1 = 1) = \frac{P(D_1 = 1 \mid H = 1)P(H = 1)}{P(D_1 = 1)} = \frac{0.0015}{0.011485} = 0.1306. \quad (1.35)$$

Out of $10{,}000$ people (Figure 1.36), all 15 infected people test positive, but so do 1% of the $9{,}985$ healthy people ($\approx$ 100 false alarms), leaving only $15/115 \approx 13\%$ of positive tests truly infected.

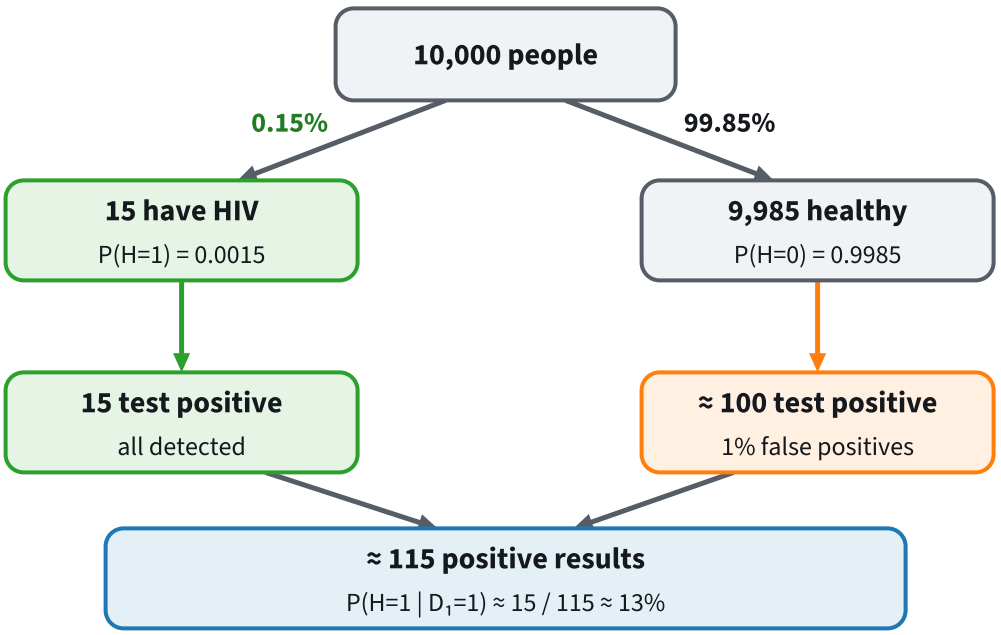


Figure 1.36: The HIV test applied to 10,000 people. The 15 infected people test positive, and so do about 100 of the 9,985 healthy people, so only 15 of about 115 positive results are correct.

Now suppose a follow-up test D_2 (with $P(D_2 = 1 \mid H = 1) = 0.98$ and false-positive rate $P(D_2 = 1 \mid H = 0) = 0.03$) also comes back positive. Assuming D_1 and D_2 are conditionally independent given H, the joint false-positive rate drops to $0.01 \cdot 0.03 = 0.0003$

while sensitivity is $1 \cdot 0.98 = 0.98$. Marginalizing gives $P(D_1 = 1, D_2 = 1) = 0.0003(0.9985) + 0.98(0.0015) = 0.00176955$, and Bayes' theorem raises the posterior probability of infection (Figure 1.37) to

$$P(H = 1 \mid D_1 = 1, D_2 = 1) = \frac{0.98 \cdot 0.0015}{0.00176955} = 0.8307. \tag{1.36}$$

Simulating two million virtual patients yields an empirical fraction around 0.82–0.83, within two ± 0.006 standard errors ($\sqrt{0.83 \cdot 0.17/3500}$) of the exact value 0.8307 across the $\approx 3{,}500$ double-positive patients:

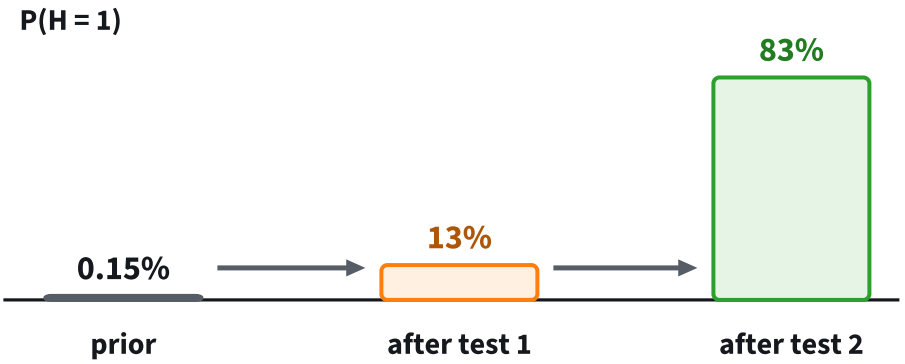


Figure 1.37: The probability that the patient has HIV: the prior, after one positive test, and after two conditionally independent positive tests.

```
n = 2000000
H = torch.rand(n) < 0.0015
D1 = torch.rand(n) < torch.where(H, 1.00, 0.01)
D2 = torch.rand(n) < torch.where(H, 0.98, 0.03)
H[D1 & D2].to(torch.float32).mean()  # Exact value: 0.8307
```

```
tensor(0.8220)
```

1.6.6 Summary

Probability reasons forward from a model to the likelihood of outcomes, whereas statistics reasons backward from observed data to infer properties of the underlying process. Built on Kolmogorov's three axioms, joint distributions $P(X, Y)$ marginalize by summing out variables and factorize as $P(X)P(Y)$ when variables are independent. Expectation $\mathbb{E}[X]$ is always linear, while variance $\mathrm{Var}(X) = \mathbb{E}[(X - \mu)^2]$ adds across independent variables, giving the average of n independent draws standard error $\sigma/\sqrt{n}$: measuring deviations in standard errors shows why $56/100$ heads (1.2 standard errors) is routine chance whereas $5{,}600/10{,}000$ (12 standard errors) is decisive, and why collecting more data shrinks *epistemic* uncertainty as $1/\sqrt{n}$ while leaving *aleatoric* uncertainty unchanged. Finally, Bayes' theorem $P(H \mid D) = P(D \mid H)P(H)/P(D)$ updates prior beliefs in light of evidence, explaining why a single positive test for a rare condition (0.15% prevalence, 1% false-positive rate) yields only a 13.1% posterior until a second conditionally independent test raises it to 83.1%.

1.6.7 Exercises

1. **Portfolio return and risk.** Assume that you manage a portfolio of n stocks whose returns $\mathbf{s} \in \mathbb{R}^n$ are random with mean $\mu = \mathbb{E}[\mathbf{s}]$ and covariance matrix Σ. You invest a fraction $\alpha_i \geq 0$ of your capital in stock i, with $\sum_i \alpha_i = 1$.
   1. Compute the expected return and the variance of the return of the portfolio α.
   2. Assume that every stock has variance σ^2 and that every pair of distinct stocks has correlation $\rho \geq 0$. Compute the variance of the equally weighted portfolio $\alpha_i = 1/n$ and its limit as $n \to \infty$. Compare the case $\rho = 0$ with Equation 1.32.
   3. (Optional.) Formulate an optimization problem that maximizes the expected return while constraining the variance to an upper bound. This is the Markowitz portfolio problem (Markowitz 1952). Methods for constrained problems of this kind are developed in Section 35.6.

2. code **Chebyshev for the coin-toss estimator.** Equation 1.32 gives the variance of the frequency $\hat{p}$ of heads after n tosses.
   1. For a fair coin and $n = 100$, use Chebyshev's inequality to bound $P(|\hat{p} - p| \geq 0.1)$.
   2. Estimate the same probability by simulating 10,000 repetitions of 100 tosses, compare it with the bound, and account for the gap.
   3. Determine how many tosses Chebyshev's inequality requires to guarantee $P(|\hat{p} - p| \geq 0.01) \leq 0.05$ for a fair coin, and compare with the n at which the standard error $\sqrt{p(1-p)/n}$ itself equals 0.01.

3. **Comparing two classifiers.** ● Two classifiers are evaluated on the same test set of $n = 10{,}000$ images. Classifier A labels 9,050 images correctly and classifier B labels 9,000 correctly. Treat the test images as independent draws.
   1. Compute the standard error of each accuracy.
   2. Assume that the two accuracies are independent and compute the standard error of their difference. Is the difference of 0.005 large compared with the noise?
   3. The two accuracies are measured on the same images, so they are not independent. Suppose that the classifiers disagree on 300 images, A being correct on 175 of them and B on 125. For image i, let $Z_i = 1$ if only A is correct, $Z_i = -1$ if only B is correct, and $Z_i = 0$ otherwise, so that the difference in accuracy is the average of the Z_i. Compute the standard error of this average and compare it with your answer to the previous part.
   4. Use $\mathrm{Var}(X - Y) = \mathrm{Var}(X) + \mathrm{Var}(Y) - 2\,\mathrm{Cov}(X, Y)$ to explain the difference. What sign does the covariance between the two classifiers' correctness on an image have, and why?

4. **Factorizing a chain.** Let X, Y, and Z be random variables such that Z is conditionally independent of X given Y, as in the chain of Figure 1.35.
   1. Write the joint probability $P(X, Y, Z)$ as a product of factors that each involve at most two of the variables.
   2. Generalize to a sequence $X_1, \dots, X_T$ in which each X_t depends on the earlier variables only through X_{t-1}. How many numbers specify the joint distribution if every X_t takes k values, compared with an arbitrary joint distribution of T such variables?

5. **Conditioning on a common effect.** Toss two independent fair coins, and let $X, Y \in$

$\{0, 1\}$ indicate heads on each. Let $Z = \max(X, Y)$ indicate that at least one coin came up heads. Compute $P(X = 1)$, $P(X = 1 \mid Z = 1)$, and $P(X = 1 \mid Z = 1, Y = 1)$. Use these values to decide whether X and Y are conditionally independent given $Z = 1$, and relate the change from the second value to the third to Figure 1.35.

6. **A second, correlated test.** ● Revisit Section 1.6.5.2 with two new tests whose outcomes are conditionally independent for patients with HIV but not for healthy patients. Either test on its own has a false positive rate of 10% and a false negative rate of 1%:

$$P(D_i = 1 \mid H = 0) = 0.1, \quad P(D_i = 0 \mid H = 1) = 0.01 \quad \text{for } i = 1, 2. \tag{1.37}$$

For $H = 1$, the joint distribution factorizes as $P(D_1, D_2 \mid H = 1) = P(D_1 \mid H = 1)\, P(D_2 \mid H = 1)$. For $H = 0$, the outcomes are coupled through $P(D_1 = D_2 = 1 \mid H = 0) = 0.02$. The baseline probability is $P(H = 1) = 0.0015$ as before.

   1. Work out the joint probability table for D_1 and D_2 given $H = 0$.
   2. Derive the probability that the patient has HIV after the first test returns positive, before the second test is run.
   3. Derive the probability that the patient has HIV after both tests return positive.
   4. Repeat the previous calculation for tests that are conditionally independent given $H = 0$ as well, and explain why the two posteriors differ.

7. **A defense lawyer's statistics.** A suspect had a documented history of abusing his now-murdered wife. His lawyer argues that, statistically, only one in a thousand wife-abusers goes on to murder his wife, so the history of abuse is weak evidence and the jury should acquit. Write M for "the suspect murdered his wife," K for "his wife was killed," and B for "he had a history of abusing her."

   1. Express the lawyer's one-in-a-thousand figure and the posterior probability of guilt in these terms, and identify which probability the lawyer has confused with which.
   2. Assume that, over the time span the lawyer's statistic refers to, an abused wife is murdered by someone other than her husband with probability 10^{-5}. Compute the posterior probability of guilt and compare it with the lawyer's figure.

Adapted from MIT 18.05, problem set 7, Problem 4.

Chapter 2

Linear Regression

Regression is the workhorse of quantitative prediction whenever the target is a meaningful number rather than a discrete category: estimating a home's sale price from its floor area and age, forecasting a patient's hospital stay from vital signs, or predicting next quarter's electricity demand. It is also an ideal proving ground for deep learning. Nearly every ingredient of neural network training—parameterized forward passes, loss functions grounded in maximum likelihood, minibatch stochastic gradient descent, reusable training harnesses, generalization diagnostics, and weight-norm regularization—already appears in its simplest, most transparent form in *linear regression*.

We begin in Section 2.1 by formulating the affine model and the squared error loss, solving the problem both in closed form via orthogonal projection and iteratively via minibatch stochastic gradient descent (SGD), and connecting squared loss to Gaussian maximum likelihood. Before touching real data, Section 2.2 constructs a synthetic benchmark with known ground-truth parameters $(\mathbf{w}^*, b^*)$ and noise level σ, giving us an exact target against which to test our code. The next three sections build the training pipeline in stages: first from raw tensors and automatic differentiation (Section 2.3), then factored into the reusable `Module`, `DataModule`, and `Trainer` classes that power the rest of the book (Section 2.4), and finally expressed in a few lines with high-level framework layers and optimizers (Section 2.5).

Fitting the training data is only half the problem. Using polynomial curve fitting, Section 2.6 separates training error from generalization error, decomposes expected test error into squared bias, variance, and noise, and shows how validation splits and K-fold cross-validation select model complexity without peeking at test data. Finally, Section 2.7 tames overfitting in high dimensions by adding a squared ℓ_2 penalty, showing how weight decay selectively shrinks directions that the training inputs determine poorly.

2.1 Linear Regression

Suppose you want to estimate the sale price of a house from its floor area and age. Unlike classification, which assigns an input to a discrete category, *regression* predicts a continuous numerical quantity. Finding a rule that maps quantitative inputs to a numerical output has a distinguished two-century pedigree. In 1805, Adrien-Marie Legendre published the *method of least squares* to determine comet orbits from noisy astronomical sightings (Legendre 1805); four years later, Carl Friedrich Gauss grounded least squares in probability

theory by asking what distribution of measurement errors makes the least-squares fit the most probable one, arriving at the Gaussian distribution (Gauss 1809) (having already used the method in 1801 to relocate the newly discovered dwarf planet Ceres after it vanished behind the sun's glare).

Today, regression powers everything from clinical risk scoring and demand forecasting to the bounding-box heads of object detectors. Better still, *linear regression*, where the prediction is an affine combination of the inputs, introduces nearly every core concept of deep learning in a setting simple enough to analyze completely by hand. Viewed as a computational graph, linear regression is a single artificial neuron: a *fully connected layer* with one output unit. In this section, we formulate the model and the squared loss, solve it both analytically via the normal equations and iteratively via *minibatch stochastic gradient descent* (the workhorse optimizer of deep learning), benchmark vectorized array operations against Python loops, and connect the loss function to maximum-likelihood estimation under different noise distributions.

```
import time
import numpy as np
import torch
```

2.1.1 The Model and the Loss

2.1.1.1 Features and Labels

To estimate a house's sale price from its area (in square meters) and age (in years), we collect a *training dataset* of historical sales. Each house in that record is an *example* (or *data point*, *sample*, or *instance*). The quantity we aim to predict, here the price, is the *label* (or *target*), while the measurable attributes on which we base the prediction (area and age) are the *features* (or *covariates*).

Suppose our dataset contains n examples. We index them by $i \in \{1, \dots, n\}$, writing $\mathbf{x}^{(i)} = (x_1^{(i)}, x_2^{(i)})^\top \in \mathbb{R}^2$ for the feature vector of the i^{th} house and $y^{(i)} \in \mathbb{R}$ for its observed sale price. More generally, when each example is described by d features, $\mathbf{x}^{(i)} \in \mathbb{R}^d$, and the full dataset is the collection of pairs $\{(\mathbf{x}^{(1)}, y^{(1)}), \dots, (\mathbf{x}^{(n)}, y^{(n)})\}$.

2.1.1.2 The Linear Model

Every supervised learning method starts with a *model class*: a parameterized family of functions mapping features to predictions. The linear assumption posits that the target can be approximated by a weighted sum of the features plus an offset:

$$\text{price} = w_{\text{area}} \cdot \text{area} + w_{\text{age}} \cdot \text{age} + b. \tag{2.1}$$

Here w_{area} and w_{age} are the *weights*, and b is the *bias* (also called the *offset* or *intercept*). Each weight sets the marginal influence of its feature—how much the predicted price changes per extra square meter of floor space, holding age fixed—while the bias b anchors the prediction when all features are zero. Even if no real house has zero area, keeping b is essential: without it, the fitted plane is forced to pass through the origin, severely restricting the functions our model can express. Strictly speaking, Equation 2.1 is an *affine* function of the features

(a linear map plus a constant translation), though machine learning universally refers to it as a linear model.

With d features, the model's prediction $\hat{y}$ (where the hat distinguishes an estimate from the observed label y) takes the form

$$\hat{y} = w_1 x_1 + \cdots + w_d x_d + b = \mathbf{w}^\top \mathbf{x} + b, \tag{2.2}$$

where $\mathbf{w} = (w_1, \ldots, w_d)^\top \in \mathbb{R}^d$ collects the weights into a single column vector. Whenever we evaluate the model across all n examples at once, we stack the feature vectors $\mathbf{x}^{(i)}$ as the rows of the *design matrix* $\mathbf{X} \in \mathbb{R}^{n \times d}$ and the predictions $\hat{y}^{(i)}$ into the column vector $\hat{\mathbf{y}} \in \mathbb{R}^n$:

$$\hat{\mathbf{y}} = \mathbf{X}\mathbf{w} + b, \tag{2.3}$$

where the scalar bias b is added to every component of $\mathbf{X}\mathbf{w}$ via broadcasting (Section 1.1.3.2). Figure 2.1 shows the shapes involved. Searching for the best linear predictor now reduces to finding the parameters $(\mathbf{w}, b) \in \mathbb{R}^d \times \mathbb{R}$ that make $\hat{\mathbf{y}}$ match the observed labels $\mathbf{y} = (y^{(1)}, \ldots, y^{(n)})^\top$ as closely as possible. Even when the true relationship between features and labels is only approximately linear and is contaminated by unmeasured factors and measurement noise, a linear model frequently provides a fast, interpretable, and surprisingly competitive baseline.

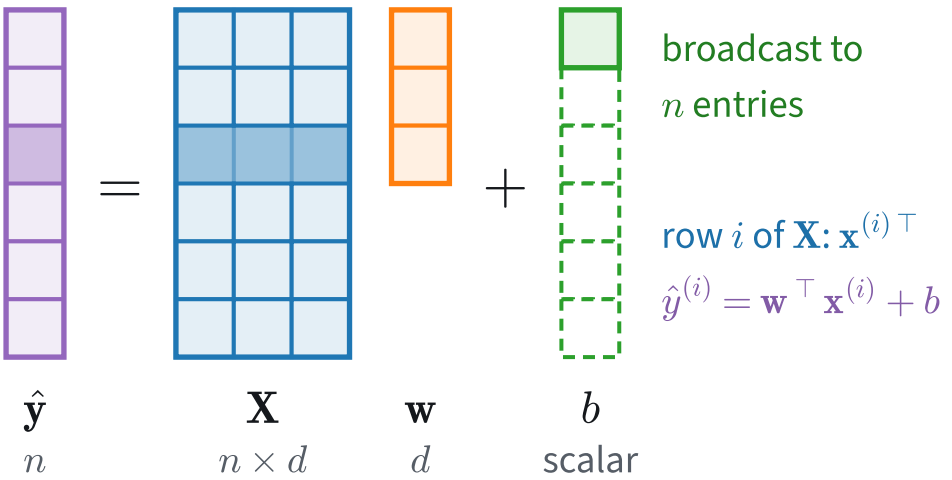


Figure 2.1: The shapes in $\hat{\mathbf{y}} = \mathbf{X}\mathbf{w} + b$, drawn for $n = 6$ examples and $d = 3$ features. Each row of $\mathbf{X}$ is one example and produces one entry of $\hat{\mathbf{y}}$ (shaded); $\mathbf{w}$ holds one weight per column, and the scalar b is broadcast to all n entries.

2.1.1.3 The Squared Loss

Before we can search for the best parameters, we need a quantitative measure of how badly any candidate $(\mathbf{w}, b)$ misses the observed labels. A *loss function* assigns a nonnegative penalty to the gap between prediction $\hat{y}$ and truth y, vanishing when the two agree. For regression, the standard choice is the *squared error*:

$$l^{(i)}(\mathbf{w}, b) = \frac{1}{2}\left(\hat{y}^{(i)} - y^{(i)}\right)^2. \tag{2.4}$$

The factor of $\frac{1}{2}$ is purely a cosmetic convenience: it cancels the factor of 2 that drops down when we differentiate the square, leaving the location of the minimum unchanged. Because the difference $\hat{y}^{(i)} - y^{(i)}$ is squared, overestimates and underestimates of equal magnitude are penalized symmetrically, while large misses are penalized much more heavily than small ones: a prediction off by 3 incurs nine times the loss of a prediction off by 1. Geometrically (Figure 2.2), each residual $\hat{y}^{(i)} - y^{(i)}$ is the vertical distance between the data point and the regression line, and $2\,l^{(i)}$ is the area of the square built on that vertical segment.

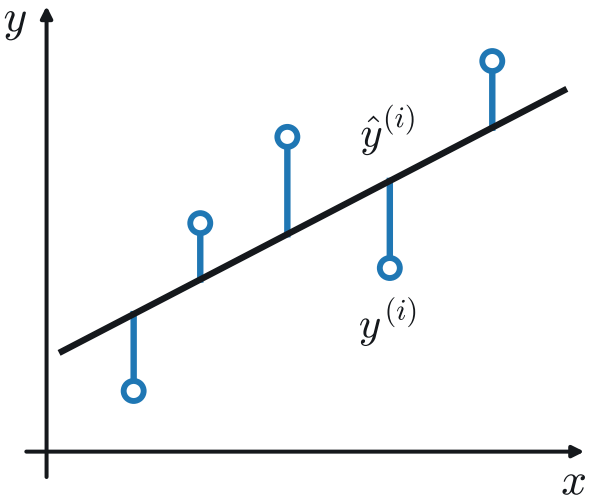


Figure 2.2: Fitting a line to data by minimizing the sum of squared vertical residuals.

To score $(\mathbf{w}, b)$ across the entire training set of n examples, we average the per-example losses to form the empirical *loss* (or *cost* / *objective*):

$$\begin{aligned} L(\mathbf{w}, b) &= \frac{1}{n}\sum_{i=1}^{n} l^{(i)}(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n} \frac{1}{2}\left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right)^2 \\ &= \frac{1}{2n}\|\mathbf{X}\mathbf{w} + b - \mathbf{y}\|_2^2. \end{aligned}$$

Dividing by n puts the objective on a per-example scale, so that doubling the dataset size does not double the numerical magnitude of the loss or its gradients. Training the model means solving the optimization problem

$$\mathbf{w}^*, b^* = \underset{\mathbf{w}, b}{\operatorname{argmin}}\ L(\mathbf{w}, b). \tag{2.5}$$

Because errors are squared, a prediction off by 10 incurs a hundred times the penalty of a prediction off by 1, so a single corrupted label can dominate the fit. To see how severely, let

us place twenty points on the line $y = 2x$, corrupt a single label to $y^{(6)} = 10{,}000$ (as a data-entry typo might), and fit a line through the origin $\hat{y} = wx$ in two ways: first by the closed-form squared-loss minimizer $w = \sum_i x^{(i)} y^{(i)} / \sum_i (x^{(i)})^2$, and second by minimizing the *mean absolute error* $\frac{1}{n}\sum_i |wx^{(i)} - y^{(i)}|$ over a fine grid of candidate slopes:

```
x = np.arange(1.0, 21.0)
y = 2 * x
y[5] = 10000                            # corrupt a single label
w_sq = (x * y).sum() / (x * x).sum()    # squared-loss minimizer
w_grid = np.linspace(0, 30, 3001)       # slopes 0, 0.01, ..., 30
mae = np.abs(w_grid[:, None] * x - y).mean(axis=1)
w_mae = w_grid[mae.argmin()]            # MAE minimizer on the grid
print(f'true w: 2.00, squared loss: {w_sq:.2f}, MAE: {w_mae:.2f}')
```

```
true w: 2.00, squared loss: 22.88, MAE: 2.00
```

A single corrupted point out of twenty yanks the squared-loss slope from 2.00 all the way to 22.88, while the absolute-error slope does not budge from 2.00. Under squared loss, one error of nearly $10{,}000$ overwhelms the combined pull of all nineteen clean points; under absolute error, its pull grows only linearly, so the nineteen clean points easily outvote it. Which behavior is desirable depends on what kind of noise we expect in our labels, a question we answer from first principles in Section 2.1.3.

2.1.2 Minimizing the Loss

How do we find $(\mathbf{w}^*, b^*)$ when there are d features and n examples? Linear regression is one of the very few models in machine learning that admits both an exact closed-form formula and a scalable iterative optimizer. Comparing the two reveals why deep learning overwhelmingly favors iterative gradient methods.

2.1.2.1 The Normal Equations

First, we can absorb the scalar bias b into the weight vector by appending a constant column of 1s to the design matrix: each input becomes $(\mathbf{x}^{(i)}, 1)^\top \in \mathbb{R}^{d+1}$ and the parameter vector becomes $(\mathbf{w}, b)^\top \in \mathbb{R}^{d+1}$ (or equivalently, we can center both $\mathbf{X}$ and $\mathbf{y}$ by their sample means; see Exercise 1). Writing $\mathbf{X}$ and $\mathbf{w}$ in this augmented notation, the loss is simply

$$L(\mathbf{w}) = \frac{1}{2n}\|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2. \tag{2.6}$$

We can find its global minimum by pure algebra, completing the square without taking a derivative, whenever $\mathbf{X}^\top\mathbf{X}$ is invertible (which holds if and only if the columns of $\mathbf{X}$ are linearly independent, requiring $n \geq d$). Define $\hat{\mathbf{w}} = (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{y}$, so that $\mathbf{X}^\top\mathbf{X}\hat{\mathbf{w}} = \mathbf{X}^\top\mathbf{y}$. Adding and subtracting $\mathbf{X}\hat{\mathbf{w}}$ inside the norm and expanding the square yields

$$\begin{aligned} \|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2 &= \|(\mathbf{y} - \mathbf{X}\hat{\mathbf{w}}) + \mathbf{X}(\hat{\mathbf{w}} - \mathbf{w})\|_2^2 \\ &= \|\mathbf{y} - \mathbf{X}\hat{\mathbf{w}}\|_2^2 + \|\mathbf{X}(\hat{\mathbf{w}} - \mathbf{w})\|_2^2 + 2(\mathbf{y} - \mathbf{X}\hat{\mathbf{w}})^\top\mathbf{X}(\hat{\mathbf{w}} - \mathbf{w}). \end{aligned} \tag{2.7}$$

Look at the cross term at the end:

$$(\mathbf{y} - \mathbf{X}\hat{\mathbf{w}})^\top \mathbf{X} = (\mathbf{X}^\top \mathbf{y} - \mathbf{X}^\top \mathbf{X}\hat{\mathbf{w}})^\top = \mathbf{0}^\top.$$

It vanishes identically for *every* $\mathbf{w}$. What remains is a sum of two nonnegative squared norms. The first term, $\|\mathbf{y} - \mathbf{X}\hat{\mathbf{w}}\|_2^2$, does not depend on $\mathbf{w}$ at all; the second term, $\|\mathbf{X}(\hat{\mathbf{w}} - \mathbf{w})\|_2^2$, is zero if and only if $\mathbf{w} = \hat{\mathbf{w}}$ (since $\mathbf{X}$ has full column rank). Thus $\hat{\mathbf{w}}$ is the unique global minimizer. We could reach the same equation via calculus by setting the gradient $\nabla_{\mathbf{w}} L(\mathbf{w}) = -\frac{1}{n}\mathbf{X}^\top(\mathbf{y} - \mathbf{X}\mathbf{w})$ to zero, producing the classic *normal equations*:

$$\mathbf{X}^\top \mathbf{X}\, \mathbf{w}^* = \mathbf{X}^\top \mathbf{y}, \qquad \text{or} \qquad \mathbf{w}^* = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}. \tag{2.8}$$

When the columns of $\mathbf{X}$ are linearly dependent (for instance, when two features are redundant or when $d > n$), $\mathbf{X}^\top \mathbf{X}$ is singular: a least-squares minimizer still exists, but infinitely many weight vectors achieve the same minimum loss (Exercise 6).

In code, you should almost never form $(\mathbf{X}^\top \mathbf{X})^{-1}$ explicitly; dedicated least-squares solvers (`lstsq`) use QR or SVD factorizations that are far more numerically stable at a similar cost. Let us verify the normal equations on the three-point line model from Section 1.4 with inputs $0, 1, 2$, labels $1, 1, 0$, and augmented rows $(1, \text{input})$, so that $\hat{\mathbf{w}} = (w_1, w_2)^\top = (7/6, -1/2)^\top$ holds the intercept first and the slope second. We also check that the residual vector $\mathbf{r} = \mathbf{X}\hat{\mathbf{w}} - \mathbf{y} = (1/6, -1/3, 1/6)^\top$ satisfies $\mathbf{X}^\top \mathbf{r} = \mathbf{0}$ up to `float32` precision:

```
X = torch.tensor([[1.0, 0.0], [1.0, 1.0], [1.0, 2.0]])  # rows (1, input)
y = torch.tensor([[1.0], [1.0], [0.0]])
w_hat = torch.linalg.lstsq(X, y).solution
r = X @ w_hat - y                                        # errors
print('w_hat:', tuple(round(float(v), 4) for v in w_hat[:, 0]))
print('r:', tuple(round(float(v), 4) for v in r[:, 0]))
print(f'max |X^T r|: {float((X.T @ r).abs().max()):.0e}')
```

```
w_hat: (1.1667, -0.5)
r: (0.1667, -0.3333, 0.1667)
max |X^T r|: 2e-07
```

2.1.2.2 Least Squares as Projection

Why are these called the *normal* equations? The identity Equation 2.7 with the vanishing cross term is literally the Pythagorean theorem in $\mathbb{R}^n$ (Figure 2.3). As $\mathbf{w}$ ranges over $\mathbb{R}^d$, the possible prediction vectors $\hat{\mathbf{y}} = \mathbf{X}\mathbf{w}$ sweep out a d-dimensional subspace of $\mathbb{R}^n$: the column space (or span) of $\mathbf{X}$. Usually the observed label vector $\mathbf{y} \in \mathbb{R}^n$ does not lie in that subspace, because no linear combination of the d feature columns reproduces all n noisy observations.

Minimizing $\|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2$ means finding the point $\hat{\mathbf{y}} = \mathbf{X}\hat{\mathbf{w}}$ in the column space of $\mathbf{X}$ that lies closest to $\mathbf{y}$ in Euclidean distance. By elementary geometry, the closest point in a flat subspace is the *orthogonal projection* of $\mathbf{y}$ onto that subspace: the unique point whose difference vector $\mathbf{y} - \mathbf{X}\hat{\mathbf{w}}$ is perpendicular (*normal*) to every column of $\mathbf{X}$. Requiring $\mathbf{y} - \mathbf{X}\hat{\mathbf{w}}$ to be orthogonal to all columns of $\mathbf{X}$ is written $\mathbf{X}^\top(\mathbf{y} - \mathbf{X}\hat{\mathbf{w}}) = \mathbf{0}$, which is none other than the normal equations.

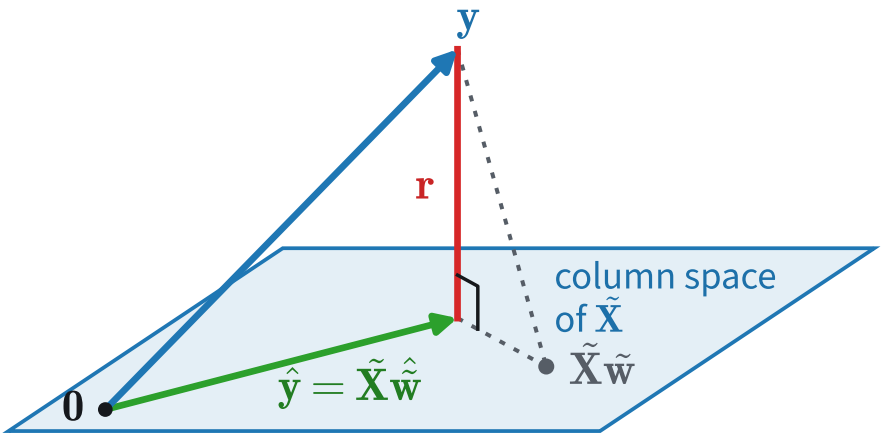


Figure 2.3: Least squares as orthogonal projection in $\mathbb{R}^n$. The column space of $\mathbf{X}$ contains every prediction vector $\mathbf{Xw}$; the closest point to $\mathbf{y}$ is the orthogonal projection $\hat{\mathbf{y}} = \mathbf{X}\hat{\mathbf{w}}$, whose residual $\mathbf{y} - \hat{\mathbf{y}}$ is perpendicular (normal) to every column of $\mathbf{X}$.

2.1.2.3 Minibatch Stochastic Gradient Descent

If linear regression has a one-line analytic solution, why not stop here? Two obstacles rule out closed-form solutions across modern deep learning:

1. **Computational scaling:** Forming $\mathbf{X}^\top\mathbf{X} \in \mathbb{R}^{d\times d}$ costs $O(nd^2)$ operations and solving the $d \times d$ system costs $O(d^3)$. When a dataset has millions of examples and a model has millions or billions of parameters, storing or inverting a $d\times d$ matrix is completely out of the question.
2. **Nonlinearity:** As soon as we replace the squared loss with another objective (like cross-entropy in Chapter 3) or stack nonlinear hidden layers (Chapter 4), no closed-form solution exists at all.

Instead, we train almost every deep learning model iteratively using *gradient descent* and its stochastic variants. Recall from Section 1.4 that the gradient $\nabla L(\mathbf{w}, b)$ points in the direction of steepest ascent of the loss. Stepping in the *opposite* direction reduces the loss, yet evaluating the full gradient $\nabla L(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n} \nabla l^{(i)}(\mathbf{w}, b)$ requires a pass over all n training examples just to take a single parameter step. When $n = 10^7$, full-batch gradient descent is glacially slow and statistically wasteful, because the average gradient over 100 randomly chosen examples is already a usable estimate of the average over 10 million. How accurate that estimate is depends on the size of the sample, not on the size of the dataset it was drawn from.

Minibatch stochastic gradient descent (minibatch SGD) exploits this statistical redundancy. At each step t, we draw a small random *minibatch* $\mathcal{B}_t \subset \{1, \dots, n\}$ of $|\mathcal{B}|$ examples, average the loss gradients over $\mathcal{B}_t$ alone, and update the parameters with a positive *learning rate* (step size) $\eta > 0$:

$$(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \frac{\eta}{|\mathcal{B}|}\sum_{i\in\mathcal{B}_t} \partial_{(\mathbf{w},b)}\, l^{(i)}(\mathbf{w}, b). \tag{2.9}$$

Because $\mathcal{B}_t$ is sampled uniformly from the training set, the minibatch gradient is an *unbiased estimator* of the true full-batch gradient, with standard error decaying as $1/\sqrt{|\mathcal{B}|}$. For linear regression under the squared loss Equation 2.4, differentiating $l^{(i)} = \frac{1}{2}(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)})^2$ explicitly gives the update rule

$$\begin{aligned} \mathbf{w} &\leftarrow \mathbf{w} - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}_t} \mathbf{x}^{(i)} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right), \\ b &\leftarrow b - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}_t} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right). \end{aligned} \tag{2.10}$$

Look at what each term does: the scalar $\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}$ is the signed prediction error on example i. If the model overpredicts ($\hat{y}^{(i)} > y^{(i)}$), the update subtracts a multiple of $\mathbf{x}^{(i)}$ from $\mathbf{w}$ and nudges b downward, pulling the next prediction closer to $y^{(i)}$. Notice also the conceptual distinction: $\mathbf{w}$ and b are *model parameters* learned by optimization, whereas the minibatch size $|\mathcal{B}|$ and learning rate η are *hyperparameters* chosen before training starts (and tuned on held-out validation data, as we discuss in Section 2.6).

2.1.2.4 Gradient Noise and the Learning Rate

How does the trajectory of minibatch SGD compare with full-batch gradient descent? Consider the contours of the empirical loss $L(\mathbf{w})$ around the least-squares solution $\hat{\mathbf{w}}$ (Figure 2.4). Full-batch gradient descent computes the gradient of L over all n examples at every step, marching smoothly across the elliptic contours into the minimum $\hat{\mathbf{w}}$ (provided η is not so large that it overshoots and diverges).

Minibatch SGD follows a different path. Even at the global minimum $\hat{\mathbf{w}}$, where the average gradient across all n examples is zero, individual noisy examples still have nonzero residuals $\hat{y}^{(i)} - y^{(i)} \neq 0$. A random minibatch $\mathcal{B}_t$ almost never has its per-example gradients cancel out to zero. Consequently, at constant learning rate η, minibatch SGD quickly descends into a small neighborhood of $\hat{\mathbf{w}}$ and then fluctuates perpetually inside a *noise ball* around $\hat{\mathbf{w}}$ whose squared radius scales as $\eta/|\mathcal{B}|$.

Halving η or doubling $|\mathcal{B}|$ shrinks that terminal noise ball, and decaying $\eta \to 0$ over the course of training lets the iterates settle onto $\hat{\mathbf{w}}$ (Chapter 9). In practice, however, reaching the bottom of the training loss to ten decimal places is rarely desirable: our ultimate goal is low error on *unseen* data, and because $\hat{\mathbf{w}}$ itself deviates from the true population parameters by statistical estimation error of order $1/\sqrt{n}$, a small optimization jitter inside that statistical uncertainty costs nothing in generalization.

2.1.2.5 Vectorization for Speed

When implementing minibatch updates Equation 2.10, you should never loop over the examples $i \in \mathcal{B}_t$ one by one in Python. Instead, we *vectorize* the computation, expressing the entire minibatch forward pass and gradient step as array and matrix operations dispatched to C++/CUDA libraries that run in parallel across SIMD vector units or GPU cores.

To see the difference firsthand, let us benchmark predicting the labels of $n = 1{,}000$ examples with weights $\mathbf{w} = (2, -3.4)^\top$ and bias $b = 4.2$: first one example at a time in a Python `for` loop, and then for all $1{,}000$ examples at once with a single matrix product `X @ w + b`:

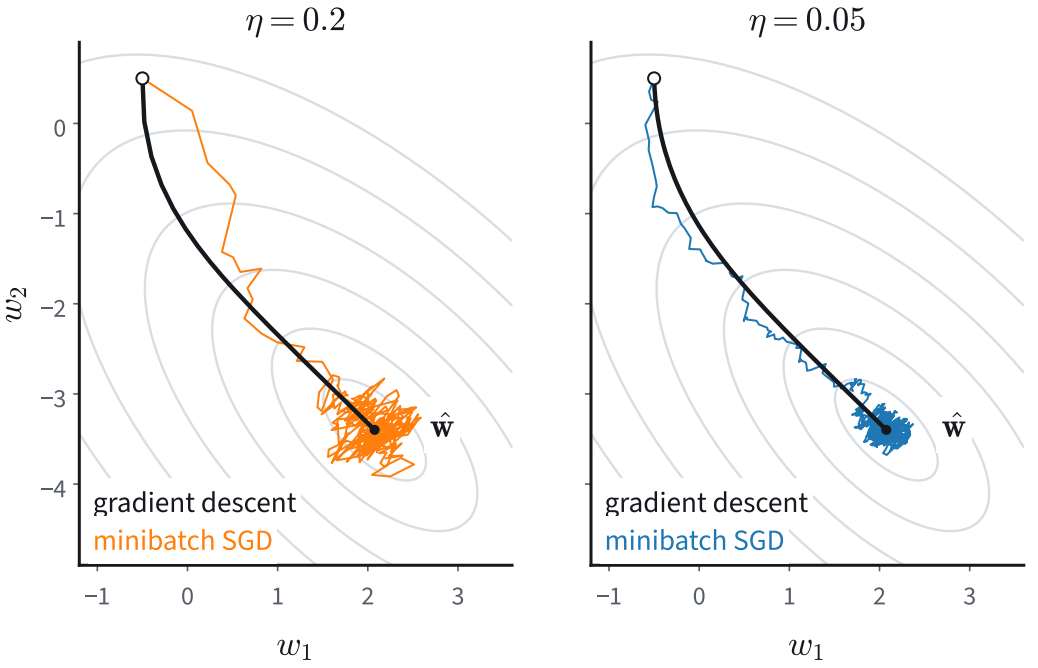


Figure 2.4: Full-batch gradient descent (black) marches smoothly along the gradient of L to the minimum $\hat{\mathbf{w}}$, taking one pass over all n examples per step. Minibatch SGD (color) takes $n/|\mathcal{B}|$ inexpensive steps per pass using noisy gradient estimates, descending rapidly to a small noise ball around $\hat{\mathbf{w}}$; quartering the learning rate (right) shrinks the ball.

```
n = 1000
X = torch.ones(n, 2)               # the values do not affect the timing
w = torch.tensor([[2.0], [-3.4]])
b = 4.2
# Wait until the device has finished (CUDA runs asynchronously)
wait = lambda x: torch.cuda.synchronize() if torch.cuda.is_available() else None
```

Our timing helper `seconds` runs a warm-up call first (to absorb one-time setup and JIT compilation costs) and calls `wait` before stopping the clock, since GPU and accelerator frameworks dispatch operations asynchronously and may return to Python while the hardware is still computing.

```
def seconds(f):
    """Time one call of f, after a warm-up call."""
    wait(f())
    start = time.perf_counter()
    wait(f())
    return time.perf_counter() - start
```

```
loop = seconds(lambda: [X[i:i + 1] @ w + b for i in range(n)])
vectorized = seconds(lambda: X @ w + b)
print(f'one example at a time: {loop * 1e3:8.3f} ms')
print(f'all examples at once:  {vectorized * 1e3:8.3f} ms')
print(f'speedup: {loop / vectorized:.0f}x')
```

```
one example at a time:   11.358 ms
all examples at once:     0.024 ms
speedup: 482x
```

Even though both paths perform the same arithmetic, computing all $1{,}000$ predictions in a single vectorized call is hundreds of times faster. The loop pays Python interpreter overhead and kernel-launch latency on every single example, whereas the matrix multiplication pays that overhead once and hands the entire batch to a single compiled linear-algebra routine. Throughout this book, vectorizing over minibatches—turning n inner products $\mathbf{w}^\top \mathbf{x}^{(i)}$ into one matrix-vector product $\mathbf{X}\mathbf{w}$—is the single most important habit for writing fast deep learning code.

2.1.3 Squared Loss and Gaussian Noise

So far we motivated the squared error loss $\frac{1}{2}(\hat{y} - y)^2$ by geometric convenience and clean derivatives. Now let us ask Gauss's deeper question: *what probabilistic assumption about the data makes minimizing squared error the statistically optimal thing to do?*

2.1.3.1 The Normal Distribution

Recall the normal (or Gaussian) distribution with mean μ and variance σ^2 (standard deviation σ), whose probability density is

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(x - \mu)^2\right).$$

This density forms a symmetric bell-shaped curve centered at μ, whose width is set by the standard deviation σ (about 68% of the probability lies within σ of the mean and about 95% within 2σ). Because the *squared* deviation $(x-\mu)^2$ appears inside the negative exponential, extreme deviations are vanishingly unlikely: a value 5σ from the mean has density $e^{-12.5} \approx 4 \times 10^{-6}$ times that at the peak. We write $\epsilon \sim \mathcal{N}(\mu, \sigma^2)$ to denote a normally distributed random variable. Because sums of many small independent disturbances converge toward a normal distribution (Section 36.3.2.3), the Gaussian is the natural default model for well-behaved measurement noise.

Formally, linear regression with additive Gaussian noise assumes that every observed label is the linear signal plus an independent normal noise term:

$$y = \mathbf{w}^\top \mathbf{x} + b + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma^2). \tag{2.11}$$

Under this model, the conditional mean $\mathbb{E}[y \mid \mathbf{x}]$ is indeed $\mathbf{w}^\top \mathbf{x} + b$ (matching Section 2.1.1.2), and labels scatter symmetrically around the line with the same noise scale σ at every input $\mathbf{x}$ (Figure 2.5).

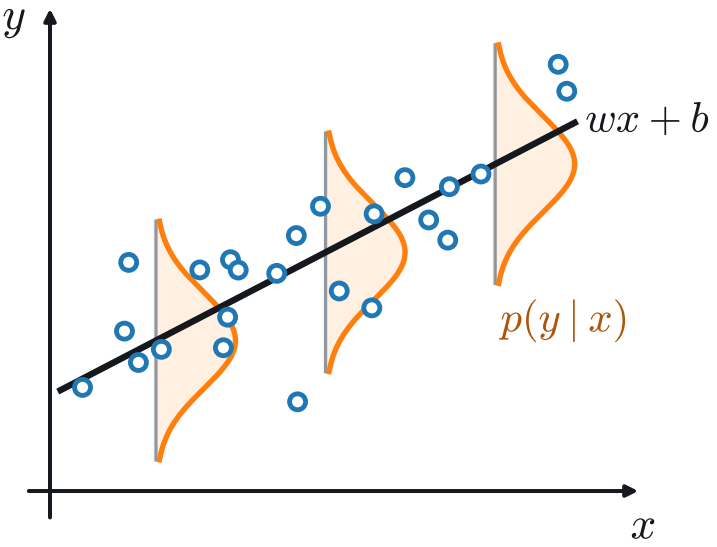


Figure 2.5: The Gaussian noise model. For every input x, the label y is drawn from a normal density $p(y \mid x)$ centered on the line $wx + b$, with the same width everywhere.

2.1.3.2 Maximum Likelihood

Under this additive Gaussian noise model, given features $\mathbf{x}$, the conditional distribution of the observed label y is normal with mean $\mathbf{w}^\top \mathbf{x} + b$ and variance σ^2:

$$P(y \mid \mathbf{x}) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(y - \mathbf{w}^\top \mathbf{x} - b)^2\right). \tag{2.12}$$

According to the principle of *maximum likelihood*, we should choose $(\mathbf{w}, b)$ to maximize the conditional probability density of observing the entire dataset $\{y^{(1)}, \dots, y^{(n)}\}$ given $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(n)}\}$. Assuming the examples are drawn independently, the dataset likelihood factors into the product $\prod_{i=1}^{n} P(y^{(i)} \mid \mathbf{x}^{(i)})$. Maximizing a positive product is equivalent to minimizing its negative logarithm, the *negative log-likelihood*:

$$-\log P(\mathbf{y} \mid \mathbf{X}) = \sum_{i=1}^{n} \frac{1}{2} \log(2\pi\sigma^2) + \frac{1}{2\sigma^2} \sum_{i=1}^{n} \left(y^{(i)} - \mathbf{w}^\top \mathbf{x}^{(i)} - b\right)^2. \tag{2.13}$$

Look closely at the two terms on the right. The first term, $\frac{n}{2} \log(2\pi\sigma^2)$, does not depend on $\mathbf{w}$ or b at all. The second term is simply $\frac{n}{\sigma^2} L(\mathbf{w}, b)$, our mean squared error loss scaled by the positive constant n/σ^2. Therefore, minimizing the squared error loss produces the *maximum-likelihood estimator* of $(\mathbf{w}, b)$ under independent homoscedastic Gaussian noise, regardless of the value of σ^2.

2.1.3.3 Choosing a Loss from a Noise Model

What does the squared loss do when the noise is *not* Gaussian? The answer needs no likelihood at all. Fix an input $\mathbf{x}$ and ask which single number c minimizes the expected squared error $\mathbb{E}[(y - c)^2 \mid \mathbf{x}]$. Setting the derivative $2(c - \mathbb{E}[y \mid \mathbf{x}])$ to zero gives the conditional mean $c = \mathbb{E}[y \mid \mathbf{x}]$, for any noise distribution with finite variance. The expected absolute error $\mathbb{E}[|y - c| \mid \mathbf{x}]$ is minimized by the conditional *median* instead. Tilt it, so that each unit of underestimation costs q and each unit of overestimation costs $1 - q$, and the minimizer becomes the q-quantile; forecasters, who need a 90% demand level more often than a mean, call this the *pinball loss*. The corrupted label in Section 2.1.1.3 showed this difference: a mean follows one enormous value, and a median ignores it. Choosing a loss therefore means choosing which statistic of y the model predicts, whatever the noise turns out to be.

The maximum-likelihood connection adds a principled recipe for designing loss functions whenever Gaussian noise is *not* the right model of reality (Figure 2.6):

Output type	Noise model	Loss (negative log-likelihood)	Statistic predicted
real-valued, well-behaved noise	Gaussian	squared error (this section)	mean of y
real-valued, outliers or heavy tails	Laplace	absolute error (exercise 2)	median of y
positive, multiplicative fluctuations (prices)	Gaussian on $\log y$	squared error on $\log y$ (exercise 3)	mean of $\log y$
counts (sales, arrivals)	Poisson	$\nu(\mathbf{x}) - y \log \nu(\mathbf{x})$ (exercise 4)	mean of y

- **Heavy-tailed outliers (Laplace noise $\to \ell_1$ loss):** If occasional corrupted measurements produce wild outliers, the quadratic growth of $(\hat{y} - y)^2$ lets a single bad data point yank the fitted line toward itself. Modeling the noise with heavier exponential tails via the Laplace distribution $p(\epsilon) = \frac{1}{2\tau} \exp(-|\epsilon|/\tau)$ turns the negative log-likelihood into the *absolute error* (ℓ_1) loss $|\hat{y} - y|$ (Exercise 2), whose gradient has constant magnitude ± 1 no matter how far an outlier sits.

- **Combining smooth optima with outlier robustness (Huber loss):** Because $|\hat{y}-y|$ has a sharp corner at zero where its derivative jumps between -1 and $+1$, practitioners often use the *Huber loss*, which splices a quadratic bowl for small residuals ($|\hat{y}-y| \le \delta$) onto linear slopes for large residuals ($|\hat{y}-y| > \delta$).
- **Count data (Poisson noise):** When the target $y \in \{0, 1, 2, \ldots\}$ is a discrete count (such as daily customer arrivals), a Poisson likelihood leads to its own convex negative log-likelihood (Exercise 4).

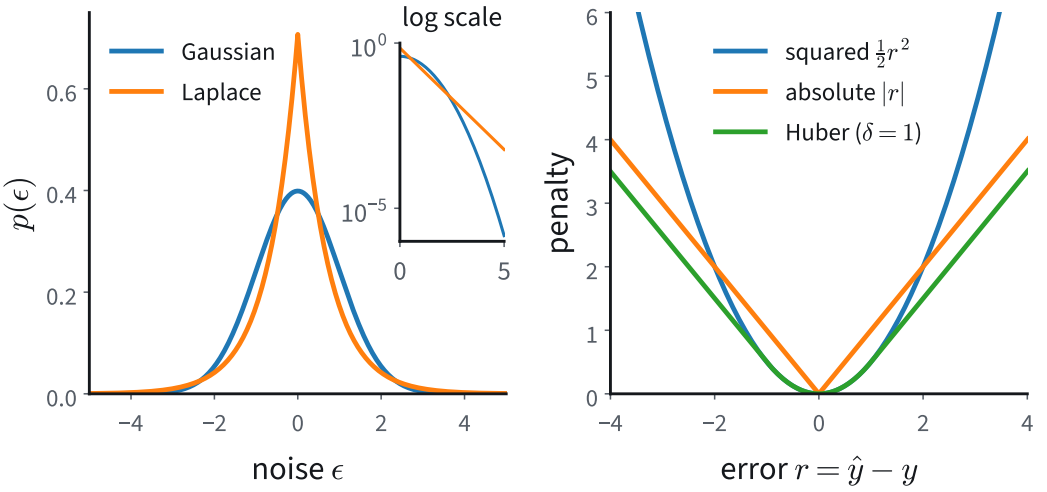


Figure 2.6: Three regression loss functions (right) and the noise densities $p(\epsilon) \propto \exp(-l(\epsilon))$ that the first two correspond to under maximum likelihood (left): squared error (Gaussian) and absolute error (Laplace), with the Huber loss between them. The log-scale inset shows the heavier Laplace tails.

2.1.4 Linear Regression as a Neural Network

To connect linear regression to the deep architectures of later chapters, we can draw it as a single-layer neural network (Figure 2.7). The diagram shows the input features $x_1, \ldots, x_d$ as nodes in the *input layer*, each wired with its own synaptic weight $w_1, \ldots, w_d$ into a single computing unit o_1 in the *output layer*, which sums the weighted inputs, adds the bias b, and emits $\hat{y} = o_1$.

By convention, we count only layers that hold tunable parameters, so linear regression is a *one-layer* network. Because every input feature connects to the output node, this layer is called a *fully connected layer* (or *dense layer*). Individual neurons in the cortex likewise integrate weighted signals from thousands of synapses on their dendrites, an analogy that inspired early pioneers like McCulloch and Pitts (McCulloch and Pitts 1943), though our focus throughout this book is on the mathematics and engineering of composing such layers into deep networks.

2.1.5 Summary

Linear regression predicts a continuous target via an affine combination of the features, $\hat{\mathbf{y}} = \mathbf{X}\mathbf{w} + b$, and scores misfit by the mean squared error $L(\mathbf{w}, b) = \frac{1}{2n}\|\mathbf{X}\mathbf{w} + b - \mathbf{y}\|_2^2$. Ge-

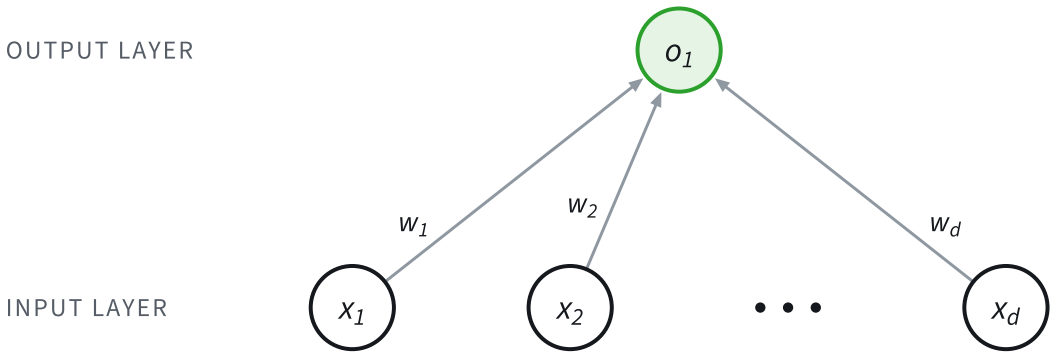


Figure 2.7: Linear regression is a single-layer neural network (a single artificial neuron).

ometrically, the closed-form normal equations $\mathbf{X}^\top \mathbf{X}\mathbf{w}^* = \mathbf{X}^\top \mathbf{y}$ project $\mathbf{y}$ orthogonally onto the column space of $\mathbf{X}$, but forming and solving that $d \times d$ system costs $O(nd^2 + d^3)$ operations and does not generalize to nonlinear networks. Minibatch stochastic gradient descent instead updates the parameters with vectorized matrix operations on small random batches $\mathcal{B}_t$, descending rapidly in $O(|\mathcal{B}|d)$ work per step to a small noise ball around the optimum. Statistically, minimizing squared error is maximum-likelihood estimation under additive Gaussian noise; replacing the Gaussian model with heavier-tailed or discrete distributions yields alternative losses such as absolute error and the Poisson negative log-likelihood.

2.1.6 Exercises

1. **Intercept and centering.** Appending a constant feature 1 to every input turns the affine model $\mathbf{w}^\top \mathbf{x} + b$ into a linear model in $(\mathbf{x}, 1)$. Centering the data removes the bias in a different way. Let $\bar{\mathbf{x}}$ and $\bar{y}$ be the averages of the features and of the labels over the training set.
   1. Show that every minimizer $(\hat{\mathbf{w}}, \hat{b})$ of the squared loss satisfies $\hat{b} = \bar{y} - \hat{\mathbf{w}}^\top \bar{\mathbf{x}}$, and conclude that the errors $\hat{y}^{(i)} - y^{(i)}$ of a least-squares fit with a bias sum to zero.
   2. Show that $\hat{\mathbf{w}}$ is a least-squares fit without bias to the centered data $(\mathbf{x}^{(i)} - \bar{\mathbf{x}},\ y^{(i)} - \bar{y})$.
2. **Laplace noise.** Assume that the additive noise ϵ follows the Laplace distribution with scale 1 (see Section 36.3), that is, $p(\epsilon) = \frac{1}{2} \exp(-|\epsilon|)$.
   1. Write out the negative log-likelihood of the data under the model $-\log p(\mathbf{y} \mid \mathbf{X})$.
   2. Determine whether a closed-form solution exists.
   3. Write the minibatch stochastic gradient descent update for $(\mathbf{w}, b)$ under this loss in the form of Equation 2.10, using 0 where $|\cdot|$ has no derivative, at zero, and elsewhere its slope $\operatorname{sign}(\cdot) = \pm 1$. A substitute of this kind is called a *subgradient*. Compare how the size of each example's contribution depends on its error under this loss and under the squared loss.
   4. code Apply this update to the corrupted-label data of Section 2.1.1.3, for a line through the origin, using the full dataset in every step, the learning rate 0.002, and 2,000 steps from $w = 0$. The minimizer of the absolute error is exactly $w = 2$. Where do the iterates end? Explain why they do not settle at the minimizer, and propose a fix.
3. **Beyond Gaussian noise.** Suppose you use regression to estimate the price of houses or of stocks.

   1. Model the price directly as $y = \mathbf{w}^\top \mathbf{x} + b + \epsilon$ with $\epsilon \sim \mathcal{N}(0, \sigma^2)$, and take $\sigma = 100{,}000$ dollars. Compute the probability that the model assigns to a negative price for a house whose predicted price is $50{,}000$ dollars. Explain why no single σ suits both this house and a house whose predicted price is $5{,}000{,}000$ dollars.
   2. Now regress on the logarithm of the price, $\log \text{price} = \mathbf{w}^\top \mathbf{x} + b + \epsilon$, where $\epsilon \sim \mathcal{N}(0, \sigma^2)$ now acts on the log scale. Compute the conditional mean and standard deviation of the price and show that their ratio does not depend on $\mathbf{x}$. Determine whether $\exp(\mathbf{w}^\top \mathbf{x} + b)$ is an unbiased prediction of the price. The same modeling choice underlies the Black–Scholes model of option pricing (Black and Scholes 1973).
4. **Counting apples.** Suppose we want to use regression to estimate the *number* of apples sold in a grocery store.
   1. Identify the problems with a Gaussian additive noise model for such a target.
   2. The Poisson distribution models counts. It is given by $p(k \mid \nu) = \nu^k e^{-\nu}/k!$, where $\nu > 0$ is the rate and $k \in \{0, 1, 2, \ldots\}$ is the observed count. Prove that $\mathbb{E}[k] = \nu$.
   3. Derive the negative log-likelihood of a count k under a rate $\nu(\mathbf{x})$ that depends on the features, dropping terms that do not depend on the parameters, and check it against the table in Section 2.1.3.3.
   4. Compare the parameterizations $\nu(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + b$ and $\log \nu(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + b$. Determine for each whether $\nu(\mathbf{x}) > 0$ holds for all inputs and parameters, and state the consequence for the loss of part (c). Write that loss in terms of $(\mathbf{w}, b)$ for the second parameterization.
5. code **The size of the noise ball.** Generate $n = 1{,}000$ examples with two features drawn from $\mathcal{N}(0, 1)$ and labels $y = \mathbf{x}^\top \mathbf{w}^* + \epsilon$, where $\mathbf{w}^* = (2, -3.4)$ and $\epsilon \sim \mathcal{N}(0, 1)$, and compute the least-squares solution $\hat{\mathbf{w}}$ with `lstsq`.
   1. Run minibatch SGD without a bias, starting at $\hat{\mathbf{w}}$, with minibatches of 10 and $\eta = 0.1$ for 20,000 steps. Estimate the mean squared distance $\mathbb{E}\|\mathbf{w} - \hat{\mathbf{w}}\|^2$ of the iterates from $\hat{\mathbf{w}}$.
   2. Repeat for $\eta \in \{0.05, 0.025\}$ and for minibatch sizes 20 and 40. Check that the mean squared distance is roughly proportional to $\eta / |\mathcal{B}|$.
   3. Why does the distance not depend on how far $\hat{\mathbf{w}}$ is from $\mathbf{w}^*$? What would change if the labels had no noise at all?
6. **Singular design matrix.** ● Full column rank makes the least-squares minimizer unique. Suppose instead that $\mathbf{X}^\top \mathbf{X}$ is singular.
   1. Show that a minimizer of $\|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2$ still exists and characterize the set of all minimizers.
   2. Write the singular value decomposition as $\mathbf{X} = \mathbf{U}\Sigma\mathbf{V}^\top$ with orthogonal $\mathbf{U}$ and $\mathbf{V}$ and a rectangular diagonal Σ of nonnegative singular values. The Moore–Penrose pseudoinverse is

$$\mathbf{X}^\dagger = \mathbf{V}\Sigma^\dagger\mathbf{U}^\top, \tag{2.14}$$

      where $\Sigma^\dagger$ transposes Σ and inverts its nonzero entries. Show that $\mathbf{X}^\dagger\mathbf{y}$ is the minimizer of smallest norm within the set of part (a).
   3. Starting from $\mathbf{w} = \mathbf{0}$, which minimizer does gradient descent converge to? State whether the answer changes for minibatch stochastic gradient descent.

2.2 Synthetic Data

Real-world datasets are messy, and worse yet, they never come with an answer key: when you train a new algorithm on real housing records and get a weight vector $\hat{\mathbf{w}}$, you have no way of knowing the true data-generating parameters $\mathbf{w}^*$ or the irreducible noise floor. Before unleashing an implementation on real data, a seasoned researcher first tests it on *synthetic data* generated from a known ground-truth model.

On a synthetic dataset, we control the feature distribution, the true weights $\mathbf{w}^*$ and bias b^*, the sample size n, and the noise level σ. That gives us two direct checks on our code: we can test whether the learned parameters $(\hat{\mathbf{w}}, \hat{b})$ approach $(\mathbf{w}^*, b^*)$, and we can compare the final loss against the theoretical noise floor $\frac{1}{2}\sigma^2$. In this section, we build the synthetic dataset that we use throughout the next three sections, inspect its scatter plots, and compute the least-squares fit as a reference benchmark.

2.2.1 Generating the Data

To keep visualization effortless while exercising multi-feature matrix algebra, we generate $n = 1{,}000$ training examples with $d = 2$ features. We draw the design matrix $\mathbf{X} \in \mathbb{R}^{1000\times 2}$ with independent standard normal entries $X_{ij} \sim \mathcal{N}(0, 1)$, and generate the labels from the linear model

$$\mathbf{y} = \mathbf{X}\mathbf{w}^* + b^* + \epsilon, \tag{2.15}$$

using ground-truth parameters

$$\mathbf{w}^* = (2, -3.4)^\top, \qquad b^* = 4.2, \tag{2.16}$$

and additive Gaussian noise $\epsilon^{(i)} \sim \mathcal{N}(0, \sigma^2)$ with standard deviation $\sigma = 0.01$. Notice that the signal $(\mathbf{X}\mathbf{w}^* + b^*)$ varies on a scale of several units, whereas the noise $\sigma = 0.01$ is hundreds of times smaller—so any healthy optimizer should recover $\mathbf{w}^*$ and b^* to within a few thousandths.

```
def synthetic_data(w, b, noise, n):
    """n examples with standard normal features and labels Xw + b + noise."""
    X = d2l.randn(n, len(w))
    eps = d2l.randn(n, 1) * noise
    return X, d2l.matmul(X, d2l.reshape(w, (-1, 1))) + b + eps
```

Let us draw $2{,}000$ examples using $\mathbf{w}^* = (2, -3.4)$, $b^* = 4.2$, and $\sigma = 0.01$: we will reserve the first $1{,}000$ for training and the remaining $1{,}000$ for validation. Because the validation examples come from the exact same distribution but are never touched during fitting, evaluating our loss on them reveals how accurately the model predicts unseen data—a distinction we examine closely in Section 2.6.

```
w_star, b_star, noise = d2l.tensor([2, -3.4]), 4.2, 0.01
X, y = synthetic_data(w_star, b_star, noise, 2000)
```

Each row of `X` stores a two-dimensional feature vector and each row of `y` stores its corresponding scalar label. Inspecting the first example and backing out its noise $\epsilon^{(1)} = y^{(1)} - (\mathbf{w}^{*\top}\mathbf{x}^{(1)} + b^*)$ gives us an immediate sanity check.

```
x, label = X[0], y[0, 0]
eps = label - (d2l.reduce_sum(w_star * x) + b_star)
print(f'features: ({float(x[0]):.4f}, {float(x[1]):.4f})')
print(f'label: {float(label):.4f}, noise: {float(eps):.4f}')
```

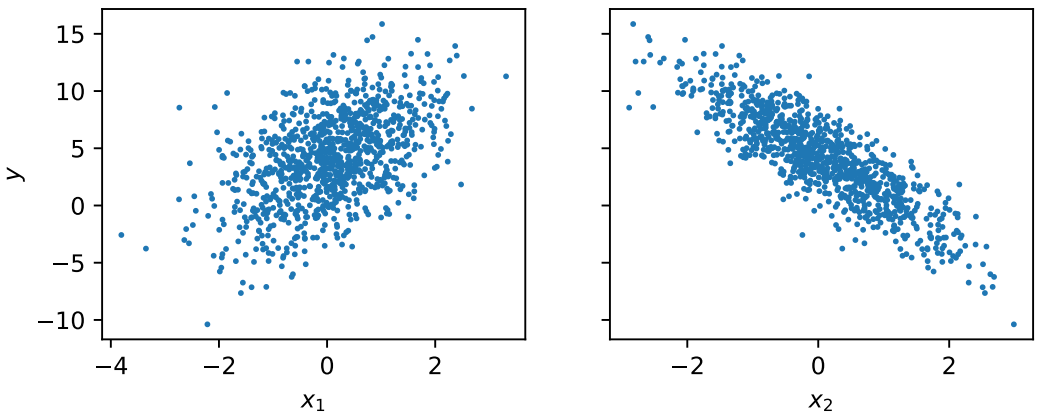
```
features: (1.2794, 1.2895)
label: 2.3786, noise: 0.0042
```

For our first example, verify the math by hand: $2x_1^{(1)} - 3.4x_2^{(1)} + 4.2$ matches $y^{(1)}$ up to the tiny $\mathcal{N}(0, 0.01^2)$ perturbation.

2.2.2 Inspecting the Data

Before fitting any model, let us plot the label y against each feature x_1 and x_2 individually:

```
d2l.set_figsize((7, 2.5))
_, axes = d2l.plt.subplots(1, 2, sharey=True)
for j, ax in enumerate(axes):
    ax.scatter(d2l.numpy(X[:1000, j]), d2l.numpy(y[:1000, 0]), s=2)
    ax.set_xlabel(f'$x_{j + 1}$')
    ax.set_ylabel('$y$' if j == 0 else '')
```



Each scatter plot reveals a clear linear trend with the expected sign: y climbs with x_1 (slope $+2$) and falls more steeply with x_2 (slope -3.4). Why does each 2D scatter plot look like a thick band rather than a razor-thin line when $\sigma = 0.01$ is so tiny? Because plotting y against x_2 alone ignores each point's variation in x_1: the unplotted term $2x_1$ has standard deviation 2, which spreads the vertical position of the points across a band several units wide. Viewing marginal 2D projections of a multivariate relationship always folds the unplotted coordinates into apparent vertical scatter.

2.2.3 Recovering the Parameters

Since we derived the closed-form normal equations in Section 2.1.2.1, let us compute the exact least-squares solution $(\hat{\mathbf{w}}, \hat{b})$ on our 1,000 examples right now using `lstsq` on the augmented design matrix $[\mathbf{X}, \mathbf{1}]$. This gives us the gold-standard finite-sample fit that minibatch SGD in Section 2.3 will aim to match:

```
n = 1000                                        # The training examples
X1 = torch.cat((X[:n], torch.ones(n, 1)), 1)    # A column of ones for b
w_hat = torch.linalg.lstsq(X1, y[:n]).solution[:, 0]
```

Now we print the actual estimation errors alongside our predicted standard error $\sigma/\sqrt{n}$.

```
e_w = d2l.numpy(w_hat[:2] - w_star)
print(f'error in w: ({e_w[0]:.1e}, {e_w[1]:.1e})')
print(f'error in b: {float(w_hat[2] - b_star):.1e}')
print(f'standard error: {noise / n ** 0.5:.1e}')
```

```
error in w: (3.3e-04, 5.7e-04)
error in b: 1.1e-04
standard error: 3.2e-04
```

Even with `lstsq` solving the empirical loss to machine precision, $(\hat{\mathbf{w}}, \hat{b})$ differs from the true $(2, -3.4, 4.2)$ in the fourth decimal place. That tiny discrepancy is *statistical estimation error*, not numerical error: the 1,000 noise draws ϵ happen to correlate slightly with the columns of $\mathbf{X}$, tilting the least-squares plane by a quantity of order $\sigma/\sqrt{n} = 0.01/\sqrt{1000} \approx 3 \times 10^{-4}$ (Exercise 1). This is a limit of the data, not of the solver: no unbiased estimator computed from these 1,000 noisy points has a smaller expected squared error (the *Cramér–Rao bound*), although a single run can land closer by luck. And as Exercise 2 explores, when the true data-generating function is nonlinear, least squares finds the best linear approximation to it over the feature distribution.

2.2.4 Summary

Generating synthetic data from a known linear model $\mathbf{y} = \mathbf{X}\mathbf{w}^* + b^* + \epsilon$ with $\mathbf{w}^* = (2, -3.4)^\top$, $b^* = 4.2$, and $\sigma = 0.01$ gives us two quantitative checks that real data lack: an irreducible loss floor of $\frac{1}{2}\sigma^2 = 5 \times 10^{-5}$ and known ground-truth parameters. Plotting y against a single coordinate x_j folds the variation of the unplotted features into the vertical spread, turning a nearly noiseless plane into a wide 2D band. Even an exact least-squares solver deviates from $(\mathbf{w}^*, b^*)$ by statistical estimation error of order $\sigma/\sqrt{n}$ ($\approx 3 \times 10^{-4}$ for $n = 1{,}000$), setting the accuracy target for iterative optimization in the next section.

2.2.5 Exercises

1. code **Recovery under noise.** ● Repeat the least-squares fit of Section 2.2.3 on 1,000 independent datasets for each setting below, and measure the standard deviation of the estimation error $\hat{w}_1 - w_1^*$ of the first weight.
    1. Vary `noise` over $\{0.001, 0.01, 0.1, 1\}$ with 1,000 training examples, and the number n of training examples over $\{10, 100, 1{,}000, 10{,}000\}$ with `noise=0.01`. Plot both measurements against the prediction $\sigma/\sqrt{n}$ on logarithmic axes.

2. Find the smallest n at which the measurement agrees with $\sigma/\sqrt{n}$ to within 10%, and say which approximation behind that prediction fails for fewer examples.
3. Write $\tilde{\mathbf{X}} = [\mathbf{X}, \mathbf{1}]$, $\tilde{\mathbf{w}}^* = (\mathbf{w}^*, b^*)$, and $\hat{\tilde{\mathbf{w}}} = (\hat{\mathbf{w}}, \hat{b})$. Show from the normal equations that $\hat{\tilde{\mathbf{w}}} - \tilde{\mathbf{w}}^* = (\tilde{\mathbf{X}}^\top \tilde{\mathbf{X}})^{-1} \tilde{\mathbf{X}}^\top \epsilon$, and derive $\mathrm{Cov}(\hat{\tilde{\mathbf{w}}}) = \sigma^2 (\tilde{\mathbf{X}}^\top \tilde{\mathbf{X}})^{-1}$ from it. Then generate correlated features $x_2 = \rho x_1 + \sqrt{1-\rho^2}\, z$, with x_1 and z independent and standard normal, for $\rho \in \{0, 0.9, 0.99\}$. Predict the standard error of $\hat{w}_1$ from this formula, check it numerically, and explain why the predictions $\hat{\tilde{\mathbf{w}}}^\top \tilde{\mathbf{x}}$, with $\tilde{\mathbf{x}} = (\mathbf{x}, 1)$, on new examples drawn from the same distribution remain accurate when the weights do not.

2. code **A model that does not fit.** ● Add the term $c\, x_1^2$ with $c = 0.5$ to every label, so that the labels no longer follow a linear function of the features, and fit the linear model by least squares as in this section.
   1. Using $\mathbb{E}[x_1^2] = 1$ and $\mathbb{E}[x_1^3] = 0$, predict which of $\hat{w}_1$, $\hat{w}_2$, and $\hat{b}$ end up away from the true values, and by about how much.
   2. Check the prediction for $n \in \{100, 1{,}000, 10{,}000\}$ training examples. Which estimation errors shrink as n grows, which do not, and how do the shrinking ones compare with $\sigma/\sqrt{n}$?
   3. A test that flags any parameter error above 10^{-3}, about three standard errors, would pass a correct implementation on the data of this section. Would it declare a correct implementation faulty on these data? Explain what passing or failing such a test can tell you when the model does not fit the data.
3. **The loss floor and its fluctuation.** The next section judges a training run by its validation loss, the average of $\frac{1}{2}(\hat{y} - y)^2$ over the 1,000 validation examples.
   1. Show that a model with the true parameters has expected loss $\sigma^2/2$ on a new example, and that no other parameters have a smaller one. This is the floor that a correct training run levels off at.
   2. For $\epsilon \sim \mathcal{N}(0, \sigma^2)$ one has $\mathbb{E}[\epsilon^4] = 3\sigma^4$. Compute the standard deviation of the loss $\frac{1}{2}\epsilon^2$ of one example, and from it the standard error of the validation loss of the true parameters over $n' = 1{,}000$ examples, as a fraction of the floor.
   3. code Check both numbers by evaluating the true parameters on many independent validation sets. Then use the result of part (a) to find the smallest parameter error that raises the expected loss by one standard error of the validation loss, and compare it with the parameter-error threshold of 10^{-3} from the previous exercise.

2.3 Linear Regression Implementation from Scratch

Now that we have the theory of linear regression (Section 2.1) and a synthetic dataset with known ground-truth parameters (Section 2.2), we are ready to put both into code. In this section, we build every component of the training pipeline from scratch—parameter initialization, the linear forward pass, the squared loss, automatic differentiation, the minibatch iterator, and the SGD update loop—relying only on basic tensor arithmetic and automatic differentiation. Writing each step by hand strips away hidden machinery so we can see how gradients flow and how hyperparameters such as the learning rate shape convergence. Once every moving part is clear, Section 2.4 organizes these components into reusable classes (`Module`, `DataModule`, and `Trainer`), and Section 2.5 re-expresses the same model in a handful of lines using framework layers, built-in loss functions, and optimizers.

2.3.1 Model, Loss, and Gradient

Let us start by generating $2{,}000$ synthetic examples ($1{,}000$ for training and $1{,}000$ for validation) with true parameters $\mathbf{w}^* = (2, -3.4)^\top$, $b^* = 4.2$, and noise $\sigma = 0.01$:

```
w_star, b_star, noise = d2l.tensor([2, -3.4]), 4.2, 0.01
X, y = d2l.synthetic_data(w_star, b_star, noise, 2000)
X_train, y_train, X_val, y_val = X[:1000], y[:1000], X[1000:], y[1000:]
```

2.3.1.1 The Parameters

Our model has a weight vector $\mathbf{w} \in \mathbb{R}^2$ and a scalar bias $b \in \mathbb{R}$. We initialize the weights with small random draws from $\mathcal{N}(0, 0.01^2)$ and set the bias to 0 (for linear regression, starting from all zeros works just as well—see Exercise 1—though breaking symmetries with random weights becomes essential once we stack hidden layers in Chapter 4).

We set `requires_grad=True` on both tensors so PyTorch's autograd engine records every operation involving them.

```
def init_params(sigma=0.01):
    w = torch.normal(0, sigma, (2, 1), requires_grad=True)
    b = torch.zeros(1, requires_grad=True)
    return w, b
```

2.3.1.2 The Forward Pass

Given a minibatch design matrix $\mathbf{X} \in \mathbb{R}^{|\mathcal{B}| \times d}$, parameters $\mathbf{w} \in \mathbb{R}^{d \times 1}$, and scalar bias b, the linear prediction $\hat{\mathbf{y}} = \mathbf{X}\mathbf{w} + b$ is a single matrix-vector product plus a broadcasted addition:

```
def linreg(X, w, b):
    """The linear regression model."""
    return d2l.matmul(X, w) + b
```

2.3.1.3 The Loss

Next we implement the mean squared error $L(\mathbf{w}, b) = \frac{1}{2|\mathcal{B}|} \sum_{i \in \mathcal{B}} (\hat{y}^{(i)} - y^{(i)})^2$. We reshape `y` into a column vector `(-1, 1)` to match `y_hat`: if one tensor had shape `(n, 1)` and the other `(n,)`, subtracting them would silently broadcast into an `(n, n)` matrix of all pairwise differences. What makes this bug treacherous is that training does not fail. The mean over all n^2 squared differences is smallest when every prediction equals the average label, so SGD drives $\mathbf{w}$ to $\mathbf{0}$ and b to $\bar{y}$, and the loss falls smoothly to half the variance of the labels, $\frac{1}{2}(2^2 + 3.4^2) \approx 7.8$ for our data. A falling loss curve is therefore weak evidence that the code is right. Later in this section we check the loss against the value it should reach, and the parameters against the truth.

```
def squared_loss(y_hat, y):
    """Half the squared error, averaged over the minibatch."""
    return d2l.reduce_mean((y_hat - d2l.reshape(y, (-1, 1))) ** 2 / 2)
```

2.3.1.4 The Gradient

Before letting automatic differentiation take over, let us check on a single minibatch of 32 examples that automatic differentiation produces the same gradients as our hand-derived formulas from Section 2.1.2.3:

$$\frac{\partial L}{\partial \mathbf{w}} = \frac{1}{|\mathcal{B}|}\mathbf{X}^\top(\hat{\mathbf{y}} - \mathbf{y}), \qquad \frac{\partial L}{\partial b} = \frac{1}{|\mathcal{B}|}\sum_{i\in\mathcal{B}}(\hat{y}^{(i)} - y^{(i)}). \tag{2.17}$$

Let us slice out the first 32 training examples `(Xb, yb)` and compare the analytical gradients with automatic differentiation:

```
Xb, yb = X_train[:32], y_train[:32]
w, b = init_params()
```

```
squared_loss(linreg(Xb, w, b), yb).backward()
with torch.no_grad():
    r = linreg(Xb, w, b) - yb                # One error per example
    by_hand = torch.cat([Xb.T @ r, r.sum(0, keepdim=True)]) / len(yb)
    by_autograd = torch.cat([w.grad, b.grad[None]])
print('gradient:', d2l.numpy(by_autograd).reshape(-1).round(4))
print(f'largest difference: {float(abs(by_hand - by_autograd).max()):.1g}')
```

```
gradient: [-4.9511  7.7252 -4.6794]
largest difference: 0
```

The largest entrywise discrepancy is zero (or at most $\sim 10^{-7}$, `float32` rounding precision), confirming that autograd computes the minibatch gradient we expect. From here on, we let the framework differentiate our forward pass and loss automatically.

2.3.2 Training from Scratch

2.3.2.1 The Update

At each step, we update the parameters in the direction opposite their gradients, $(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \eta\,(\nabla_{\mathbf{w}} L, \nabla_b L)$, taking care not to record the parameter update itself inside the computational graph.

2.3.2.2 Minibatches

To sweep through the dataset in random minibatches of size $|\mathcal{B}|$, our generator `minibatches` draws a fresh random permutation of the indices $0, \ldots, n-1$ at the start of each epoch and yields successive slices of size `batch_size` (Exercise 2):

```
def minibatches(X, y, batch_size):
    """Minibatches of (X, y), in a new random order on every call."""
    indices = np.random.permutation(len(X))
    for i in range(0, len(indices), batch_size):
        batch = indices[i : i + batch_size]
        yield X[batch], y[batch]
```

With $1{,}000$ training examples and `batch_size=32`, one epoch yields $\lceil 1000/32 \rceil = 32$ minibatches: 31 full batches of 32 examples plus one remainder batch of 8. Let us inspect one pass over the generator and see what happens if we accidentally iterate over the *same* generator a second time.

```
loader = minibatches(X_train, y_train, 32)
sizes = [len(y) for X, y in loader]
print(f'{len(sizes)} minibatches, the last of size {sizes[-1]}')
print(f'second pass: {len([y for X, y in loader])} minibatches')
```

```
32 minibatches, the last of size 8
second pass: 0 minibatches
```

Each yielded pair `(Xb, yb)` has shapes `(32, 2)` and `(32, 1)`. The first pass (one *epoch*) yields all 32 minibatches as expected, while the second pass yields zero. A Python generator is single-use and exhausts silently once consumed, which is why our training loop calls `minibatches(...)` fresh at the start of every epoch.

2.3.2.3 The Training Loop

We now assemble the complete training loop in `train(lr, num_epochs, batch_size)`. In each epoch, we iterate over `minibatches(X, y, batch_size)`, compute the minibatch loss, differentiate it with respect to `w` and `b`, and apply the SGD step:

```
def train(lr, num_epochs=10, batch_size=32):
    w, b = init_params()
    val_losses = []
    for epoch in range(num_epochs):
        for Xb, yb in minibatches(X_train, y_train, batch_size):
            loss = squared_loss(linreg(Xb, w, b), yb)  # forward pass, loss
            loss.backward()                            # backward pass
            with torch.no_grad():                      # not recorded:
                for param in (w, b):
                    param -= lr * param.grad           # update
                    param.grad.zero_()                 # clear
        with torch.no_grad():
            val_losses.append(squared_loss(linreg(X_val, w, b), y_val).item())
    return w, b, val_losses
```

The loop ends by clearing each gradient, because PyTorch adds a new gradient to the stored one instead of replacing it (Section 1.5).

Notice that the core training loop is only a dozen lines of code, and those same four steps (forward pass $\rightarrow$ loss $\rightarrow$ backward pass $\rightarrow$ parameter update, drawn in Figure 2.8) train every deep network in the rest of this book.

2.3.3 Checking the Implementation

2.3.3.1 The Expected Loss

How small should the loss become when training succeeds? Even at the true parameters $(\mathbf{w}^*, b^*)$, the residuals equal the Gaussian noise $\hat{y}^{(i)} - y^{(i)} = -\epsilon^{(i)} \sim \mathcal{N}(0, \sigma^2)$, so the

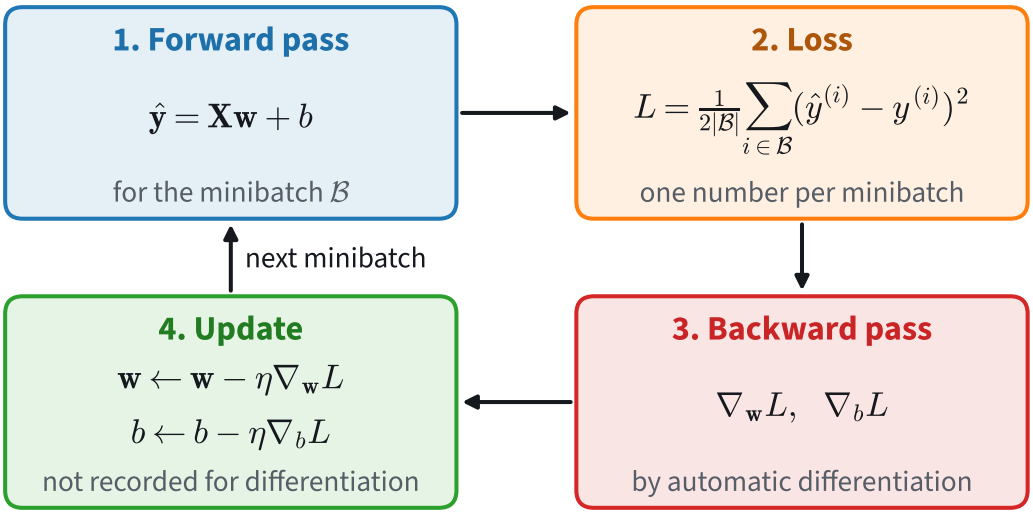


Figure 2.8: One step of minibatch SGD. The forward pass computes the predictions for a minibatch, the loss reduces them to a single number, the backward pass differentiates that number with respect to the parameters, and the update moves the parameters against the gradient. The cycle then repeats with the next minibatch.

expected loss is the *irreducible noise floor*:

$$\mathbb{E}\left[L(\mathbf{w}^*, b^*)\right] = \frac{1}{2}\mathbb{E}\left[\epsilon^2\right] = \frac{1}{2}\sigma^2 = \frac{1}{2}(0.01)^2 = 0.00005. \tag{2.18}$$

At initialization ($\mathbf{w} \approx \mathbf{0}, b = 0$), by contrast, the residual is $y = \mathbf{x}^\top\mathbf{w}^* + b^* + \epsilon$, which has mean $b^* = 4.2$ and variance $\|\mathbf{w}^*\|_2^2 + \sigma^2 = 2^2 + (-3.4)^2 + 0.01^2 = 15.5601$. Thus the expected initial loss is

$$\mathbb{E}\left[L(\mathbf{0}, 0)\right] = \frac{1}{2}\left((b^*)^2 + \|\mathbf{w}^*\|_2^2 + \sigma^2\right) = \frac{1}{2}(17.64 + 15.5601) \approx 16.6. \tag{2.19}$$

Over the course of training, we therefore expect the loss to plunge by five orders of magnitude, from roughly 16.6 down to the noise floor 0.00005.

The same computation also predicts how *fast* the loss should fall. At arbitrary parameters, the residual is $r = \mathbf{x}^\top(\mathbf{w} - \mathbf{w}^*) + (b - b^*) - \epsilon$. Because the two features are independent standard normal variables and the noise is independent of both, every cross term has expectation zero, and

$$\mathbb{E}\left[L(\mathbf{w}, b)\right] = \tfrac{1}{2}\|\mathbf{w} - \mathbf{w}^*\|_2^2 + \tfrac{1}{2}(b - b^*)^2 + \tfrac{1}{2}\sigma^2. \tag{2.20}$$

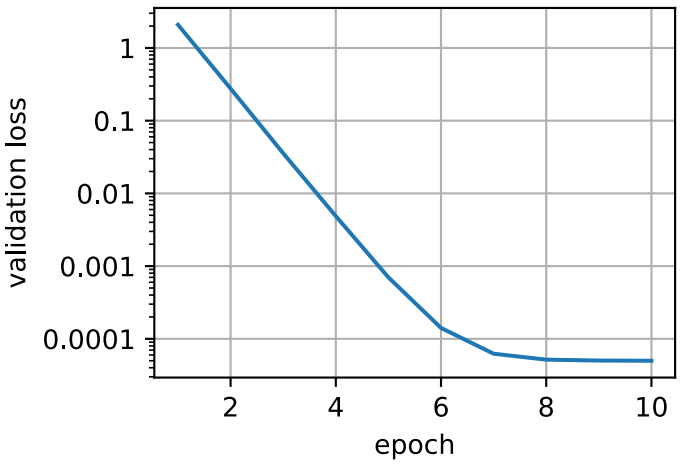
The gradient of this expected loss with respect to $(\mathbf{w}, b)$ is simply the *parameter error* $(\mathbf{w} - \mathbf{w}^*, b - b^*)$, so a gradient step of size η shrinks the parameter error by the factor $1 - \eta$ and the *excess loss* above the floor by $(1 - \eta)^2$. Minibatch gradients match the expected gradient only on average, so this gives an approximate trajectory rather than an exact identity. At

$\eta = 0.03$, one epoch of 32 minibatch updates should reduce the excess loss by a factor of roughly $1/(1-0.03)^{64} \approx 7$.

2.3.3.2 The Training Run

Let us run `train` for 10 epochs with learning rate $\eta = 0.03$ and minibatch size $|\mathcal{B}| = 32$, plotting the validation loss after each epoch on a logarithmic scale:

```
w, b, val_losses = train(lr=0.03)
d2l.plot(list(range(1, 11)), val_losses, xlabel='epoch',
         ylabel='validation loss', yscale='log')
```



Sure enough, the validation loss plunges along a straight line, dropping roughly $7\times$ per epoch before flattening out near the predicted floor of 5×10^{-5} around epoch 7 or 8. Let us compare the measured excess loss against our closed-form prediction epoch by epoch over the first seven epochs.

```
floor = noise ** 2 / 2                              # sigma^2 / 2
excess = (float(d2l.reduce_sum(w_star ** 2)) + b_star ** 2) / 2  # 16.6
for epoch, loss in enumerate(val_losses[:7], start=1):
    excess *= (1 - 0.03) ** (2 * 32)                # 32 steps per epoch
    print(f'epoch {epoch}: measured {loss - floor:.1e}, '
          f'predicted {excess:.1e}')
```

```
epoch 1: measured 2.1e+00, predicted 2.4e+00
epoch 2: measured 2.8e-01, predicted 3.4e-01
epoch 3: measured 3.6e-02, predicted 4.8e-02
epoch 4: measured 4.8e-03, predicted 6.8e-03
epoch 5: measured 6.5e-04, predicted 9.7e-04
epoch 6: measured 9.1e-05, predicted 1.4e-04
epoch 7: measured 1.2e-05, predicted 2.0e-05
```

Measured and predicted excess losses track each other tightly over the early epochs and gradually drift apart, by at most a modest factor of 3 or 4 by epoch 7. Why the slight drift? Our idealized prediction assumed the sample covariance matrix $\mathbf{H} = \frac{1}{n}\sum_i \tilde{\mathbf{x}}^{(i)}(\tilde{\mathbf{x}}^{(i)})^\top$ (see

exercise 1) equals the exact identity matrix. On a finite sample of $1{,}000$ examples, its eigenvalues sit within roughly 10% of 1, and over several epochs these small deviations compound, so the empirical run drifts slightly ahead of or behind the idealized prediction. Both the decay rate and the noise floor match theory.

2.3.3.3 Recovered Parameters

Because our dataset is synthetic, we can also compare the learned $(\mathbf{w}, b)$ directly against the true $\mathbf{w}^* = (2, -3.4)^\top$ and $b^* = 4.2$:

```
w_error = d2l.numpy(d2l.reshape(w, w_star.shape) - w_star)
b_error = d2l.numpy(b - b_star)
print('error in w:', ', '.join(f'{e:.1e}' for e in w_error))
print('error in b:', ', '.join(f'{e:.1e}' for e in b_error))
```

```
error in w: 1.7e-04, 5.1e-04
error in b: -4.6e-04
```

Both the weights and the bias are recovered to within a thousandth, matching the statistical estimation floor $\sigma/\sqrt{n} \approx 3 \times 10^{-4}$ we found with the least-squares solver in Section 2.2.

2.3.3.4 The Learning Rate

What happens if we choose the learning rate poorly? Let us run the exact same 10-epoch loop across four learning rates spanning three orders of magnitude, $\eta \in \{0.003, 0.03, 0.3, 3\}$, and print the final validation loss of each run (Exercise 1):

```
for lr in [0.003, 0.03, 0.3, 3]:
    val_loss = train(lr)[2][-1]
    print(f'lr = {lr:<6} validation loss: {val_loss:.1e}')
```

```
lr = 0.003  validation loss: 2.1e+00
lr = 0.03   validation loss: 5.0e-05
lr = 0.3    validation loss: 5.1e-05
lr = 3      validation loss: nan
```

The results match our theory across the board. At $\eta = 0.003$, the error contracts by only 0.997 per step, leaving the loss around 2 to 3 after ten epochs (right near $16.6 \times 0.997^{640} \approx 2.4$). At $\eta = 0.3$, the model rockets down to the noise floor within the very first epoch. And at $\eta = 3$, every update roughly doubles the error with alternating sign, quickly overflowing to `nan`. Keep in mind that the stability ceiling of 2 is a property of our *data scaling*, not a universal constant of SGD: it arises because our features have unit variance. Scaling a feature by $s > 1$ increases the loss curvature along that weight by s^2 and shrinks the maximum stable learning rate to roughly $2/s^2$—a major practical reason why we standardize input features before training (Section 1.2).

Next, Section 2.4 packages the training machinery we just built into reusable object-oriented classes (`Module`, `DataModule`, and `Trainer`), and Section 2.5 shows how to swap our hand-written tensors, loss, and updater for ready-made framework components.

2.3.4 Summary

A complete training loop requires only four primitives: a parameterized forward pass $\hat{\mathbf{y}} = \mathbf{X}\mathbf{w} + b$, a scalar loss $\frac{1}{2|\mathcal{B}|}\|\hat{\mathbf{y}} - \mathbf{y}\|_2^2$, reverse-mode automatic differentiation, and a minibatch SGD update run outside gradient recording. Because a falling loss curve alone does not prove that code is bug-free (a silent shape mismatch can still lower loss toward half the label variance), we verify our implementation quantitatively on synthetic data: the validation loss decays exponentially at the predicted rate $(1 - \eta)^{64}$ per epoch to the noise floor $\frac{1}{2}\sigma^2 = 5 \times 10^{-5}$, and the learned parameters match $(\mathbf{w}^*, b^*)$ within $O(\sigma/\sqrt{n})$. Across learning rates, small step sizes crawl slowly, moderate step sizes reach the noise floor quickly, and step sizes exceeding $2/\mu_{\max}$ overshoot the quadratic bowl and diverge.

2.3.5 Exercises

1. code **Initialization at the extremes.** `init_params` draws the weights with standard deviation `sigma=0.01`. Predict whether training reaches the noise floor when the weights are initialized to exactly zero, and when `sigma=1000`. Use Equation 2.20 to estimate how many more epochs the second case needs at learning rate 0.03, then check your estimate.

2. code **Second derivatives.** Compute by hand the 3×3 Hessian of the minibatch loss with respect to $\theta = (\mathbf{w}, b)$, and state which quantities it depends on. Then compute it with automatic differentiation via `torch.func.hessian` (Section 1.5) for one fixed minibatch of the synthetic data, once at initialization and once after training. Compare both results with your formula and with the curvature of Equation 2.20.

3. code **Stability and minibatch size.** ● The expected loss predicts that steps along its gradient diverge for $\eta > 2$. Let $\tilde{\mathbf{x}} = (\mathbf{x}, 1)$ be an input with a 1 appended for the bias.

    1. For full-batch gradient descent, show that each step multiplies the difference between the parameters and the least-squares solution by the matrix $\mathbf{I} - \eta\mathbf{H}$, where $\mathbf{H}$ is the average of $\tilde{\mathbf{x}}\tilde{\mathbf{x}}^\top$ over the training set. Compute the eigenvalues of $\mathbf{H}$ and predict the largest stable learning rate.
    2. Train with minibatches of 1,000, 32, and 1 example, giving each run the same number of updates, say 320. For each minibatch size, find the largest learning rate in a grid of your choice at which the final training loss is below 10^{-3}.
    3. Show that a step on a single example multiplies the component of the parameter error along $\tilde{\mathbf{x}}$ by $1 - \eta\|\tilde{\mathbf{x}}\|^2$, and use $\mathbb{E}\|\tilde{\mathbf{x}}\|^2 = 3$ to explain the limit you found for minibatches of one example.

4. **Why reshuffle.** `minibatches` draws a fresh random order in every epoch. Explain which property of the minibatch gradient this preserves. Then construct a fixed ordering of the training set, such as sorting by label, under which minibatch SGD without reshuffling behaves differently, and describe the trajectory of $(\mathbf{w}, b)$ within an epoch and across epochs.

5. code **Robust losses.** ● Replace `squared_loss` with the mean absolute error `d2l.reduce_mean(d2l.abs(y_hat - d2l.reshape(y, (-1, 1))))`.

    1. Train on the synthetic data and compare the recovered parameters and the loss curve with those of the squared loss.

    2. Set one training label to $y^{(5)} = 10{,}000$, the corruption used in Section 2.1.1.3,

and train with both losses. Record b after every minibatch update, plot it against the iteration for both runs, and explain the pattern of the squared-loss curve, including what determines its final value.

3. On the corrupted data, replace the loss with the minibatch average of the Huber loss

$$l_\delta(y, y') = \begin{cases} \frac{1}{2}(y - y')^2 & \text{if } |y - y'| \le \delta \\ \delta\left(|y - y'| - \frac{\delta}{2}\right) & \text{otherwise,} \end{cases} \tag{2.21}$$

whose quadratic part matches the squared loss of this section. Train for $\delta \in \{1, 10, 10^2, 10^3, 10^4\}$, and plot the distance $|\widehat{b} - b^*|$ of the learned bias from the generating bias $b^* = 4.2$ against δ on logarithmic axes. Explain the shape of the curve from the derivative of l_δ with respect to y', and relate the result to the penalty curves of Figure 2.6.

2.4 Object-Oriented Design for Implementation

In Section 2.3, we implemented linear regression from raw tensors and trained it with a standalone twelve-line `train` function wired directly to one model and one dataset. As we move on to multilayer perceptrons, convolutional networks, recurrent networks, and Transformers, rewriting data loaders, epoch loops, metric averagers, and live plots in every section would bury the new ideas under repetitive plumbing.

To keep our focus on modeling while keeping every line of the training harness transparent and modifiable, we organize the `d2l` library around three clean object-oriented abstractions (Figure 2.9), inspired by frameworks like PyTorch Lightning (Falcon and The PyTorch Lightning team 2019):

1. **`Module` (models):** Holds trainable parameters, defines the `forward` pass, computes the `loss`, specifies `training_step` and `validation_step`, and returns its optimizer via `configure_optimizers`.
2. **`DataModule` (data):** Encapsulates dataset downloading, preprocessing, and splitting, exposing `train_dataloader()` and `val_dataloader()` iterables.
3. **`Trainer` (training loop):** Connects any `Module` to any `DataModule`, running the epoch-by-epoch minibatch training and validation passes, averaging metrics across batches, and plotting live learning curves on a `ProgressBoard`.

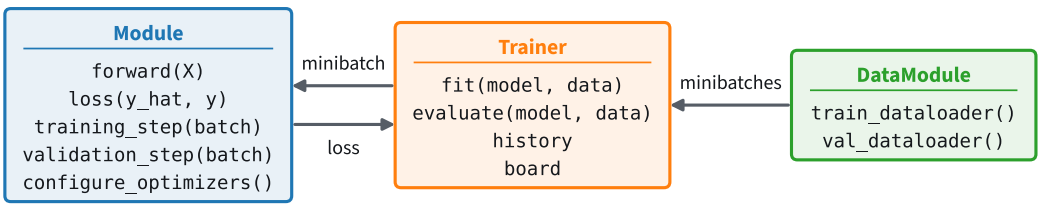


Figure 2.9: The three core classes of the `d2l` training harness: `DataModule` supplies training and validation minibatches, `Module` defines the model, loss, and optimizer, and `Trainer` drives the training and validation loops.

```
import collections
```

2.4.1 Utilities

Before building the three core classes, we introduce two small Python utilities that keep notebook definitions readable and self-contained:

1. **`@add_to_class(Class)`:** In a textbook or notebook, presenting a 60-line class in one giant code block makes it hard to explain each method as it appears. The decorator `@add_to_class(Class)` lets us define a class's skeleton first and attach new methods to it incrementally in later code cells.
2. **`HyperParameters` (`save_hyperparameters()`):** Writing `self.a = a; self.b = b; ...` at the top of every `__init__` method is pure boilerplate. Inheriting from `HyperParameters` and calling `self.save_hyperparameters()` inspects the caller's frame and automatically binds every `__init__` argument (except any listed in `ignore`) as an attribute on `self`.

```
def add_to_class(Class):
    """Register functions as methods in created class."""
    def wrapper(obj):
        setattr(Class, obj.__name__, obj)
        return obj
    return wrapper
```

Let us try it out on a toy class `A` and an already-instantiated object `a`:

```
class A:
    def __init__(self):
        self.b = 1

a = A()
```

Now we attach a new method `do` in a separate cell. Because Python resolves methods on the class at call time, our existing instance `a` immediately gains access to `do`:

```
@add_to_class(A)
def do(self):
    print('Instance attribute "b" is', self.b)

a.do()
```

```
Instance attribute "b" is 1
```

Our second helper eliminates boilerplate in constructors, which otherwise fill up with repetitive assignments like `self.lr = lr` and `self.batch_size = batch_size`. Because most constructor arguments in deep learning are hyperparameters, we inherit from `d2l.HyperParameters` and call `self.save_hyperparameters()`, which automatically binds every constructor argument (except any listed in `ignore`) as an attribute on `self`. Under the hood it inspects the calling stack frame using Python's `inspect` module; its implementation lives in the online appendix "Utility Functions and Classes".

```
class B(d2l.HyperParameters):
    def __init__(self, a, b, c):
        self.save_hyperparameters(ignore=['c'])
        print('self.a =', self.a, 'self.b =', self.b)
        print('self.c exists:', hasattr(self, 'c'))

b = B(a=1, b=2, c=3)
```

```
self.a = 1 self.b = 2
self.c exists: False
```

Every class we build in the remainder of this section inherits from `HyperParameters`.

2.4.2 Models

The `Module` class is the base class of every model we implement. It inherits from the framework's native neural network container as well as `d2l.HyperParameters`. By default:

- Calling `model(X)` invokes `model.forward(X)`.
- `training_step(batch)` passes the features `batch[:-1]` through the model, compares the predictions against the labels `batch[-1]`, and returns the mean loss of the minibatch (later models may return a dictionary of metrics that includes `'loss'`).
- `validation_step(batch)` computes the same loss on a minibatch of validation data (and in Section 3.3, `Classifier` will override `validation_step` to report both `'loss'` and `'acc'`).
- The helper `as_metrics` normalizes any single scalar or dictionary return value into a metric dictionary so `Trainer` can log arbitrary named quantities uniformly.

In PyTorch the container is `nn.Module`.

```
class Module(nn.Module, d2l.HyperParameters):
    """The base class of models."""
    def loss(self, y_hat, y):
        raise NotImplementedError

    def forward(self, X):
        assert hasattr(self, 'net'), 'Neural network is not defined'
        return self.net(X)

    def training_step(self, batch):
        """The mean loss on one minibatch, or a dict of batch-mean metrics
        with key 'loss'."""
        return self.loss(self(*batch[:-1]), batch[-1])

    def validation_step(self, batch):
        """The mean loss on one minibatch of validation data, or a dict of
        batch-mean metrics with key 'loss'."""
        return self.loss(self(*batch[:-1]), batch[-1])

    def configure_optimizers(self):
        raise NotImplementedError
```

By convention, every minibatch is a tuple whose final element is the label tensor and whose preceding elements are the model inputs. Both `training_step` and `validation_step` unpack

`*batch[:-1]` into `self(...)`, compute the loss against `batch[-1]`, and simply *return* the result—performing no printing, plotting, or logging of their own. Keeping models completely unaware of the trainer is a deliberate design choice: it means any custom loop can train or evaluate a `Module` without hidden side effects.

In this chapter, each step returns a single scalar—the mean minibatch loss. Starting in Section 3.3, steps may instead return a dictionary mapping metric names to minibatch means (always including `'loss'`, alongside quantities like `'acc'`). Our tiny helper `as_metrics` normalizes a bare scalar loss into `{'loss': out}` so the trainer handles both uniformly.

```
def as_metrics(out):
    """A step returns a loss or a dict of metrics; return a dict."""
    return out if isinstance(out, dict) else {'loss': out}
```

2.4.2.1 Linear Regression as a Module

To see how `Module` works in practice before using built-in framework layers, let us wrap our from-scratch linear regression parameters $(\mathbf{w}, b)$, forward pass, squared loss, and from-scratch `SGD` updater into `LinearRegressionScratch(d2l.Module)`:

```
class LinearRegressionScratch(d2l.Module):
    """The linear regression model implemented from scratch."""
    def __init__(self, num_inputs, lr, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.w = nn.Parameter(d2l.normal(0, sigma, (num_inputs, 1)))
        self.b = nn.Parameter(d2l.zeros(1))
```

Next, we attach `forward` and `loss`—identical to `linreg` and `squared_loss` from Section 2.3, simply reading `self.w` and `self.b` off the model instance.

```
@d2l.add_to_class(LinearRegressionScratch)
def forward(self, X):
    return d2l.matmul(X, self.w) + self.b
```

```
@d2l.add_to_class(LinearRegressionScratch)
def loss(self, y_hat, y):
    l = (y_hat - d2l.reshape(y, (-1, 1))) ** 2 / 2
    return d2l.reduce_mean(l)
```

Next, we wrap our from-scratch minibatch SGD update rule into a reusable `SGD(d2l.HyperParameters)` class and return it from `LinearRegressionScratch.configure_optimizers`:

```
class SGD(d2l.HyperParameters):
    """Minibatch stochastic gradient descent."""
    def __init__(self, params, lr):
        self.save_hyperparameters()
```

```
    def step(self):
        for param in self.params:
            param -= self.lr * param.grad

    def zero_grad(self):
        for param in self.params:
            if param.grad is not None:
                param.grad.zero_()
```

Finally, `configure_optimizers` wires the model to our `SGD` optimizer:

```
@d2l.add_to_class(LinearRegressionScratch)
def configure_optimizers(self):
    return SGD([self.w, self.b], self.lr)
```

2.4.3 Data

The `DataModule` base class is the counterpart to `Module` for datasets. Each concrete dataset subclass implements `train_dataloader()` and `val_dataloader()` (typically delegating both to `get_dataloader(train=True/False)`):

```
class DataModule(d2l.HyperParameters):
    """The base class of data."""
    def __init__(self, root='../data', num_workers=4):
        self.save_hyperparameters()

    def get_dataloader(self, train):
        raise NotImplementedError

    def train_dataloader(self):
        return self.get_dataloader(train=True)

    def val_dataloader(self):
        return self.get_dataloader(train=False)
```

Let us package our synthetic regression generator from Section 2.2 into `SyntheticRegressionData(d2l.DataModule)`. It generates `num_train + num_val` examples up front (1,000 training and 1,000 validation examples by default) so that `Trainer` can track both training loss and held-out validation loss after every epoch:

```
class SyntheticRegressionData(d2l.DataModule):
    """Synthetic data for linear regression."""
    def __init__(self, w, b, noise=0.01, num_train=1000, num_val=1000,
                 batch_size=32):
        super().__init__()
        self.save_hyperparameters()
        self.X, self.y = d2l.synthetic_data(w, b, noise, num_train + num_val)
```

2.4.3.1 A Loader for Every Epoch

Inside `get_dataloader`, we want an object that can be iterated over once per epoch—reshuffling the training split at the start of every epoch while reading the validation split in deterministic order. A `get_tensorloader` method builds it from the split's tensors.

`TensorDataset` aligns tensors along their leading example axis, and `DataLoader` slices them into minibatches, reshuffling automatically on every pass whenever `shuffle=True`.

```
@d2l.add_to_class(d2l.DataModule)
def get_tensorloader(self, tensors, train, indices=slice(0, None)):
    tensors = tuple(a[indices] for a in tensors)
    dataset = torch.utils.data.TensorDataset(*tensors)
    return torch.utils.data.DataLoader(dataset, self.batch_size,
                                       shuffle=train)
```

Now `SyntheticRegressionData.get_dataloader` is just two lines—slicing either the training split or the validation split and delegating to `get_tensorloader`:

```
@d2l.add_to_class(SyntheticRegressionData)
def get_dataloader(self, train):
    i = slice(0, self.num_train) if train else slice(self.num_train, None)
    return self.get_tensorloader((self.X, self.y), train, i)
```

Let us instantiate our synthetic dataset and repeat the two-pass test that tripped up our plain generator earlier:

```
data = SyntheticRegressionData(w=d2l.tensor([2, -3.4]), b=4.2)
```

```
loader = data.train_dataloader()
sizes = [len(y) for X, y in loader]
print(f'{len(sizes)} minibatches, the last of size {sizes[-1]}')
print(f'second pass: {len([y for X, y in loader])} minibatches')
for _ in range(2):  # Each new pass starts in a new random order
    X, y = next(iter(loader))
    print('a new pass starts with labels', d2l.numpy(y[:3, 0]).round(2))
```

```
32 minibatches, the last of size 8
second pass: 32 minibatches
a new pass starts with labels [-0.64 -5.75  3.55]
a new pass starts with labels [-3.35  0.07 -0.03]
```

This time, both passes yield all 32 minibatches (ending with the 8-example remainder), and every pass begins in a fresh random order. Better yet, we can query `len(loader)` directly without iterating through the dataset:

```
len(loader)
```

```
32
```

2.4.4 Training

2.4.4.1 The Training Loop and the History

Now we build `Trainer`, the engine that wires a `Module` and a `DataModule` together. Calling `trainer.fit(model, data)` prepares the data loaders, attaches the model and its optimizer, and runs `max_epochs` passes. In every epoch, `fit_epoch` sweeps through the training minibatches (updating parameters and averaging training metrics) and then calls `validate` over the validation minibatches, recording the epoch averages in `self.history` and updating a live `ProgressBoard` plot:

```
class Trainer(d2l.HyperParameters):
    """Fit a Module to a DataModule, keeping a per-epoch history."""
    def __init__(self, max_epochs, num_gpus=0, gradient_clip_val=0):
        self.save_hyperparameters()
        self.board = ProgressBoard(xlabel='epoch', ylabel='loss')

    def prepare_data(self, data):
        self.train_dataloader = data.train_dataloader()
        self.val_dataloader = data.val_dataloader()

    def prepare_model(self, model):
        self.model = model

    def prepare_optimizer(self, model):
        self.optim = model.configure_optimizers()
```

To support learning-rate schedules in Section 9.7, `prepare_optimizer` also unpacks an optional `(optimizer, scheduler)` tuple when returned by `configure_optimizers`.

```
@d2l.add_to_class(Trainer)
def prepare_optimizer(self, model):
    """`configure_optimizers` may also return a learning-rate scheduler."""
    # PyTorch steps a schedule with a separate object (sec_scheduler)
    optim = model.configure_optimizers()
    self.optim, self.scheduler = (optim if isinstance(optim, tuple)
                                  else (optim, None))
```

The `fit(model, data)` method implements the same outer loop as the `train` function of Section 2.3. After running the three setup hooks, it loops over `max_epochs` epochs: in each epoch, `fit_epoch()` sweeps through the training loader and returns mean training metrics, `validate()` evaluates the validation loader (when present), and `record()` logs and plots both.

```
@d2l.add_to_class(Trainer)
def fit(self, model, data):
    self.prepare_data(data)
    self.prepare_model(model)
    self.prepare_optimizer(model)
    self.history = collections.defaultdict(list)
    self.board.clear()
    self.board.xlim = [0, self.max_epochs]
```

```
    for self.epoch in range(self.max_epochs):
        # The training mean averages over the epoch, so it sits mid-epoch
        self.record('train', self.fit_epoch(), x=self.epoch + 0.5)
        if self.val_dataloader is not None:
            self.record('val', self.validate(), x=self.epoch + 1)
        self.board.show(final=False)
    self.board.show()

@d2l.add_to_class(Trainer)
def record(self, split, metrics, x):
    for name in sorted(metrics, key=lambda name: (name != 'loss', name)):
        self.history[f'{split}_{name}'].append(metrics[name])
        self.board.draw(x, metrics[name], f'{split}_{name}')
```

Notice that `fit` exposes the current epoch number on `self.epoch` and returns nothing directly—every logged curve lives in `trainer.history`, a dictionary of lists keyed by split and metric name (`'train_loss'`, `'val_loss'`, and later `'val_acc'`). Thus `trainer.history['val_loss'][-1]` immediately retrieves the final validation loss. Sorting `'loss'` first ensures consistent curve colors on the progress board across models.

The two curves are not drawn at the same positions, and Figure 2.10 shows why. The training loss of epoch e is a mean over all minibatch steps of that epoch, each taken with different parameters, so `fit` draws it in the middle of the epoch, at $e + \frac{1}{2}$. The validation loss is computed once, with the parameters the epoch ends on, and is drawn at $e + 1$.

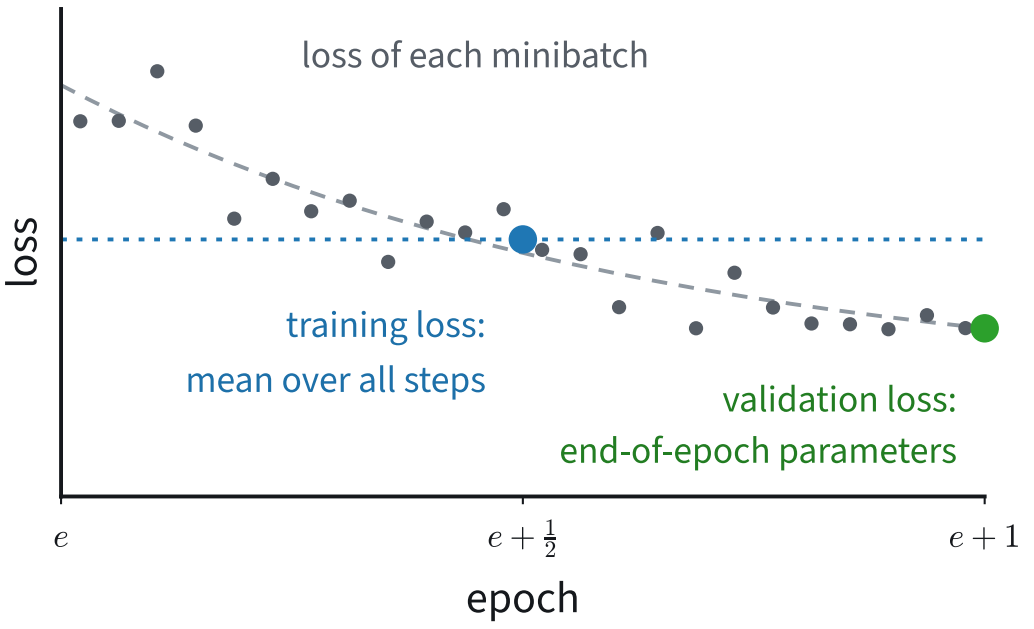


Figure 2.10: One epoch on the progress board. The losses of the individual minibatches (gray) fall as the parameters improve; their mean is the training loss, drawn at $e + \frac{1}{2}$. The validation loss is computed with the parameters at the end of the epoch and is drawn at $e + 1$.

2.4.4.2 Epoch Means

Because the last minibatch of an epoch is often smaller than `batch_size` (for instance, $1{,}000 = 31 \times 32 + 8$), an unweighted average of per-batch mean losses would give the 8 examples in the final batch the exact same vote as the 32 examples in every earlier batch. Our `Averager` helper accumulates each batch's metric weighted by its batch size `n`, so `avg.mean()` always returns the exact per-example average across the entire split:

```
class Averager:
    """Example-weighted means of batch metrics, summed on the device."""
    def __init__(self):
        self.sums, self.n = {}, 0

    def add(self, metrics, n):
        for name, value in metrics.items():
            self.sums[name] = self.sums.get(name, 0) + value * n
        self.n += n

    def mean(self):
        """One host synchronization per metric; the loss comes first."""
        names = sorted(self.sums, key=lambda name: (name != 'loss', name))
        return {name: float(self.sums[name]) / self.n for name in names}
```

2.4.4.3 Training and Validation Passes

Next, `validate` and `fit_epoch` execute the per-batch validation and training loops across our four frameworks (with optional gradient clipping `gradient_clip_val` ready for recurrent networks in Chapter 8):

```
@d2l.add_to_class(d2l.Trainer)
def prepare_batch(self, batch):
    return batch
```

Because this exact trainer will run every model throughout the rest of the book, it is slightly longer than bare linear regression requires—yet its algorithmic core is still the exact four-step SGD cycle of Figure 2.8 (forward pass, loss, backward pass, and parameter update), with a few forward-looking hooks for later chapters that remain dormant here.

The two `if` blocks are hooks for gradient clipping and learning-rate schedulers. Calling `.detach()` before passing metrics to `Averager` prevents the running loss sum from retaining the computational graph of every minibatch in memory, and `model.train()`/`model.eval()` toggle layers such as dropout and batch normalization once we introduce them.

```
@d2l.add_to_class(d2l.Trainer)
def fit_epoch(self):
    self.model.train()
    metrics = d2l.Averager()
    for batch in map(self.prepare_batch, self.train_dataloader):
        out = d2l.as_metrics(self.model.training_step(batch))
        self.optim.zero_grad()
        out['loss'].backward()
```

```
        if self.gradient_clip_val > 0:  # To be discussed later
            self.clip_gradients(self.gradient_clip_val, self.model)
        with torch.no_grad():
            self.optim.step()
        if self.scheduler is not None:
            self.scheduler.step()
        metrics.add({k: v.detach() for k, v in out.items()}, len(batch[-1]))
    return metrics.mean()

@d2l.add_to_class(d2l.Trainer)
def validate(self):
    self.model.eval()
    metrics = d2l.Averager()
    with torch.no_grad():
        for batch in map(self.prepare_batch, self.val_dataloader):
            metrics.add(d2l.as_metrics(self.model.validation_step(batch)),
                        len(batch[-1]))
    return metrics.mean()
```

2.4.4.4 The Progress Board

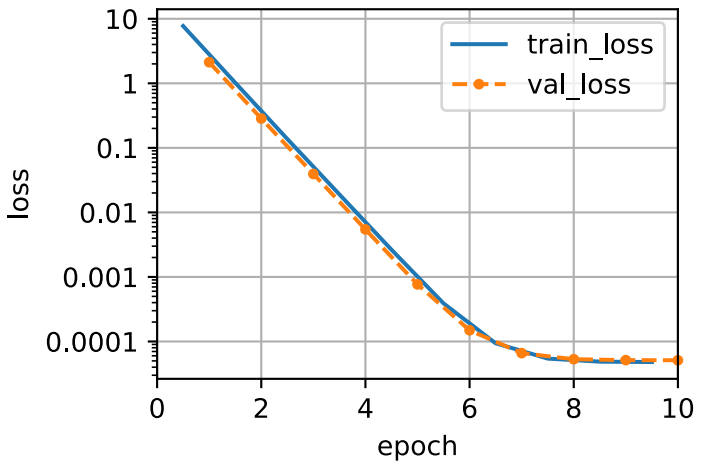
Finally, `ProgressBoard` renders live epoch-by-epoch learning curves during notebook execution (and can be silenced via `trainer.board.display = False` whenever we run silent sweeps):

```
class ProgressBoard(d2l.HyperParameters):
    """Curves that grow during training; `show` redraws them."""
    def __init__(self, xlabel=None, ylabel=None, xlim=None, ylim=None,
                 xscale='linear', yscale='linear', figsize=(3.5, 2.5),
                 display=True):
        self.save_hyperparameters()
        self.clear()

    def clear(self):
        """Forget all curves; the next `show` starts a new figure."""
        self.data, self.handle = {}, None

    def draw(self, x, y, label):
        """Append the point (x, y) to the curve called `label`."""
        xs, ys = self.data.setdefault(label, ([], []))
        xs.append(x)
        ys.append(y)

    def show(self, final=True):
        raise NotImplementedError
```

2.4.5 Training Linear Regression Again

Let us test our newly assembled `Module`, `DataModule`, and `Trainer` classes end-to-end by training `LinearRegressionScratch` on `SyntheticRegressionData` for 10 epochs and checking the recovered parameters:

```
model = LinearRegressionScratch(2, lr=0.03)
trainer = d2l.Trainer(max_epochs=10)
trainer.board.yscale = 'log'
trainer.fit(model, data)
```



The validation loss traces the exact same trajectory as in Section 2.3, plunging roughly $7\times$ per epoch until it hits the noise floor near 5×10^{-5}, with the training curve lying right on top of it (with only 3 parameters and $1{,}000$ examples, the model barely memorizes any label noise). Let us verify the final validation loss and parameter recovery errors numerically:

```
w_error = d2l.numpy(d2l.reshape(model.w, data.w.shape) - data.w)
b_error = d2l.numpy(model.b - data.b)
print(f"validation loss: {trainer.history['val_loss'][-1]:.1e}")
print('error in w:', ', '.join(f'{e:.1e}' for e in w_error))
print('error in b:', ', '.join(f'{e:.1e}' for e in b_error))
```

```
validation loss: 5.1e-05
error in w: 4.3e-04, 3.0e-04
error in b: -6.7e-04
```

Both training and validation loss drop smoothly to the 0.00005 noise floor, and the recovered parameters match $(\mathbf{w}^*, b^*)$ to within about a thousandth, confirming that our object-oriented harness reproduces our standalone scratch loop while freeing future chapters from boilerplate. In Section 2.5, we take the next step and replace our hand-written tensors, squared loss, and `SGD` class with each framework's built-in layers, losses, and optimizers.

2.4.6 Summary

To keep later chapters focused on modeling rather than repetitive training boilerplate, the `d2l` library separates models (`Module`), datasets (`DataModule`), and optimization loops (`Trainer`), supported by `@add_to_class`, `HyperParameters`, `Averager`, and `ProgressBoard`. `DataModule` returns multi-epoch loaders that reshuffle training examples on every pass without exhausting after one epoch, while `Averager` weights each minibatch metric by its example count so partial final batches never skew epoch-level means. Because `Module` exposes a clean return-value con-

tract (`training_step`, `validation_step`, and `configure_optimizers`), any model—whether built from raw tensors like `LinearRegressionScratch` or from framework layers—trains and validates through the same `trainer.fit(model, data)` call.

2.4.7 Exercises

1. **Reading the training curves.** The trainer draws the mean training loss of each epoch at the middle of the epoch and the validation loss at its end.
    1. Early in training, the training curve often lies above the validation curve, although both sets of examples come from the same distribution. Explain why from the way the two means are computed.
    2. Suppose `fit` computed the training loss like the validation loss, with the parameters at the end of each epoch. What would this cost per epoch, and what would the gap between the two curves then measure?
2. code **Early stopping.** Training for a fixed number of epochs may stop too early or too late. Write a subclass of `d2l.Trainer` whose `fit` ends training once the validation loss has not decreased for `patience` consecutive epochs, deciding from `self.history` alone. Explain why no model or dataset class has to change. Test your trainer on the synthetic regression data with a large `max_epochs`.
3. code **Planck's law.** ● The same `Module` and `Trainer` fit any model whose loss can be differentiated, not only a linear one. Use Planck's law to determine the temperature of a black body from measurements of its spectral radiance at several wavelengths. The spectral radiance is

$$B(\lambda, T) = \frac{2hc^2}{\lambda^5} \cdot \left(\exp \frac{hc}{\lambda k T} - 1 \right)^{-1}, \tag{2.22}$$

   where λ is the wavelength, T is the temperature, c is the speed of light, h is Planck's constant, and k is the Boltzmann constant. Generate measurements at a known T, then write a `d2l.Module` subclass whose `forward` computes $B(\lambda, T)$ with T as its only parameter and fit it with `d2l.Trainer`. Choose units in which B and T are of order one before fitting.
4. **A partial last minibatch.** With 1,000 examples per split and `batch_size=32`, each loader ends a pass with a partial minibatch of 8 examples. `DataLoader` discards it when given `drop_last=True`. The training loader reshuffles every epoch, whereas the validation loader reads in a fixed order. For each loader, state which examples a discarded partial minibatch leaves out over many epochs, and decide for which loader discarding is acceptable.
5. code **On-the-fly generation.** Write a `DataModule` subclass that stores no **X** or **y**. Its training loader instead draws every minibatch fresh from the data-generating process when the minibatch is requested.
    1. A stream of fresh minibatches has no natural end, yet the `Trainer` ends an epoch only when a pass over the training loader ends. Decide how many minibatches one epoch yields, make every pass stop there, and make `len` return that number.
    2. Decide whether the validation loader should also draw fresh examples on every pass, and implement your choice. Justify it by what the validation loss is used for.

2.5 Concise Implementation of Linear Regression

In Section 2.3, we built every piece of linear regression from raw tensors: parameter allocation, the affine forward pass, the squared loss, reverse-mode differentiation, the minibatch generator, and the SGD update step. In Section 2.4, we organized those components into reusable base classes (`d2l.Module`, `d2l.DataModule`, and `d2l.Trainer`) and verified that `LinearRegressionScratch` trained through `trainer.fit(model, data)` reproduced our benchmark results.

Writing and testing raw parameter tensors and update rules by hand was straightforward for a 3-parameter linear model, but modern architectures like ResNet-50 (Section 7.4) or GPT (Section 11.2) manage millions to billions of parameters across dozens of layers and train with stateful adaptive optimizers. Deep learning frameworks—PyTorch (Paszke et al. 2019), JAX (Frostig et al. 2018) (with Flax and Optax), TensorFlow (Abadi et al. 2016), and MXNet (Chen et al. 2015)—therefore provide hardware-tuned layers, numerically tested loss primitives, and standard optimizers out of the box. In this section, we see how concisely we can express linear regression by plugging those high-level building blocks into `d2l.Module` and `d2l.Trainer`.

2.5.1 The Fully Connected Layer

In our concise implementation, a subclass `LinearRegression(d2l.Module)` wraps a single built-in fully connected layer with one output unit, initializing its weights from $\mathcal{N}(0, 0.01^2)$ and its bias to zero.

PyTorch's `nn.LazyLinear(1)` requires only the output dimension at construction time and infers the number of input features d automatically on the first forward pass.

```
class LinearRegression(d2l.Module):
    """Linear regression built from framework components."""
    def __init__(self, lr):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.LazyLinear(1)

    def weight_init(self, module):
        if isinstance(module, nn.Linear):
            nn.init.normal_(module.weight, std=0.01)
            nn.init.zeros_(module.bias)
```

Notice that `LinearRegression` does not even need to define `forward`: it inherits `Module.forward` from Section 2.4, which delegates directly to `self.net(X)`.

2.5.2 The Loss Function

Instead of writing $(\hat{y} - y)^2/2$ by hand, we delegate `loss` to the framework's built-in mean squared error primitive. Standard mean squared error omits the $\frac{1}{2}$ factor of our scratch loss, so it is twice our scratch value and levels off at $\sigma^2 = 10^{-4}$ rather than $\sigma^2/2$.

```
@d2l.add_to_class(LinearRegression)
def loss(self, y_hat, y):
    fn = nn.MSELoss()
    return fn(y_hat, y)
```

2.5.3 The Optimizer

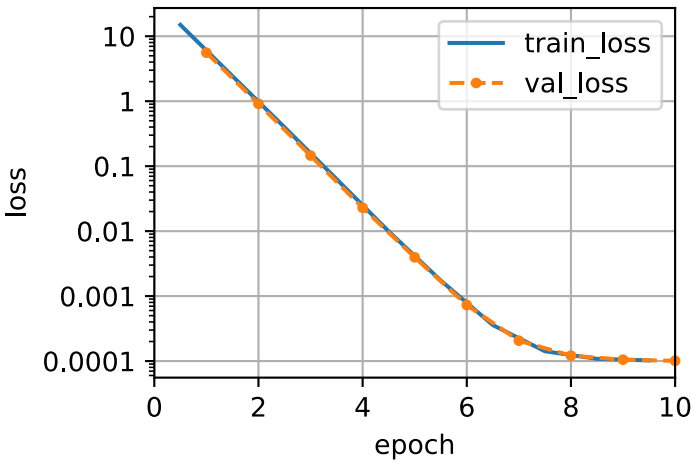
Similarly, `configure_optimizers` returns the framework's built-in minibatch SGD optimizer, wired to the module's registered parameters and learning rate `self.lr`:

```
@d2l.add_to_class(LinearRegression)
def configure_optimizers(self):
    return torch.optim.SGD(self.parameters(), self.lr)
```

2.5.4 Training with Framework Components

With `LinearRegression` defined, training reduces to instantiating the data module, the model (with learning rate $\eta = 0.015$, halved from our scratch rate $\eta = 0.03$ to compensate for the doubled MSE gradient), and `d2l.Trainer(max_epochs=10)`, then calling `trainer.fit(model, data)`:

```
data = d2l.SyntheticRegressionData(w=d2l.tensor([2, -3.4]), b=4.2)
model = LinearRegression(lr=0.015)
model(data.X[:1])  # Creates the parameters of the lazy layer
model.net.apply(model.weight_init)
trainer = d2l.Trainer(max_epochs=10)
trainer.board.yscale = 'log'
trainer.fit(model, data)
```




```
print(f"validation loss: {trainer.history['val_loss'][-1]:.2e}")
```

```
validation loss: 1.01e-04
```

The final validation loss lands within roughly 3% of the theoretical noise floor $\sigma^2 = 10^{-4}$, mirroring our from-scratch implementation. Let us add a small helper method `get_w_b` to extract the learned weight tensor and bias from the framework layer and compare them against $(\mathbf{w}^*, b^*)$:

```
@d2l.add_to_class(LinearRegression)
def get_w_b(self):
    return self.net.weight.detach(), self.net.bias.detach()
```

```
w, b = model.get_w_b()
w_error = d2l.numpy(d2l.reshape(w, data.w.shape) - data.w)
b_error = d2l.numpy(b - data.b)
print('error in w:', ', '.join(f'{e:.1e}' for e in w_error))
print('error in b:', ', '.join(f'{e:.1e}' for e in b_error))
```

```
error in w: -2.1e-04, 9.5e-04
error in b: -1.1e-03
```

Once again, the errors in $\hat{\mathbf{w}}$ and $\hat{b}$ stay within about a thousandth, matching our scratch run because the dominant error sources (label noise across $1{,}000$ examples and the remaining distance after 10 epochs) are identical regardless of who wrote the layer.

Everything we checked in the past three sections leaned on knowing the ground-truth parameters, a luxury we never have on real datasets; in Section 2.6, we tackle how to judge models strictly from held-out data.

2.5.5 Summary

Using high-level framework APIs, linear regression reduces to a single dense layer inside a `d2l.Module` subclass that delegates `loss` to built-in mean squared error and `configure_optimizers` to built-in SGD. Because standard mean squared error omits the $\frac{1}{2}$ factor of our scratch objective, both the loss (plateauing at $\sigma^2 = 10^{-4}$) and its gradient double, so halving the learning rate to $\eta = 0.015$ reproduces the parameter updates of our scratch implementation. Because `LinearRegression` satisfies the `d2l.Module` contract, `d2l.Trainer` fits it without a single change to the data pipeline or training loop.

2.5.6 Exercises

1. code **A layer without a bias.** Build `LinearRegression` with a layer that has no bias (every framework's dense layer has an argument that turns the bias off), and train it as in this section.
    1. Before training, predict the value at which the validation loss levels off, using the expected squared error formula from Section 2.3 (accounting for the factor of 2 in the mean squared error). Then check your prediction.
    2. Predict what the weights converge to and how far they end from $\mathbf{w}^*$. Since the features have mean zero, the missing bias acts on the weights like label noise of standard deviation $b^* = 4.2$; it enters both the estimation error of Section 2.2.3

and the noise ball of minibatch SGD (Section 2.1). Compare your prediction with the errors of the trained weights.

2. code **Several outputs.** Change the output width of the layer in `LinearRegression` from 1 to 2. After constructing `data`, append to `data.y` a second label column $y_2 = \mathbf{x}^\top \mathbf{w}_2^* + b_2^* + \epsilon_2$, computed from `data.X` with generating parameters of your choice and noise of the same size.
    1. Show that the squared loss summed over the two outputs splits into two independent linear regressions, one per output. Conclude that the parameters learned for one output do not depend on the labels of the other.
    2. The framework losses average over every entry of the $|\mathcal{B}| \times 2$ matrix of errors of a minibatch $\mathcal{B}$, not only over its examples. State by which factor the gradient of each output's parameters shrinks relative to the single-output model, and which learning rate reproduces the single-output training run. Confirm this by training both models from the same initial parameters on the same minibatches.
3. code **Feature scale and step size.** Multiply the first column of `data.X` by 100 after constructing `data`, leaving `data.y` unchanged, so that the generating weight of that feature becomes 0.02. Recall from exercise 1 that gradient descent diverges once η exceeds $2/\mu_{\max}$, where $\mu_{\max}$ is the largest eigenvalue of the Hessian $\mathbf{H}$ of the loss.
    1. Compute $\mathbf{H}$ for `LinearRegression.loss` and estimate its eigenvalues before and after the rescaling.
    2. Predict the largest learning rate for which training does not diverge, before and after the rescaling, and check both predictions by training at learning rates around them.
    3. At a learning rate just below the new limit, predict how many updates the second weight needs to reduce its error by a factor of 100, and compare the prediction with a measurement. Then standardize the rescaled feature as in Section 1.2, and show that `lr=0.015` works again.
4. code **Optimization versus estimation error.** ● Train `LinearRegression` with `lr=0.015` and `max_epochs=10` on `SyntheticRegressionData` with `num_train` set to each of $5, 50, 500, 5{,}000$ and `batch_size=min(32, num_train)`. For each run, compare the distance $\|\hat{\mathbf{w}} - \mathbf{w}^*\|_2$ of the trained weights from the generating weights $\mathbf{w}^*$, stored in `data.w`, with the same distance for the least-squares solution on the same training data, computed as in the recovery subsection (Section 2.2.3).
    1. Count the parameter updates in each run, and explain the ratio of the two errors at each `num_train`.
    2. Find the number of epochs after which the run with `num_train=5` comes within a factor of two of its least-squares error.

2.6 Generalization

Imagine two college students, Ellie and Irene, preparing for a statistics final exam by practicing on past exams. Ellie memorizes every past question and its exact numerical answer word for word; scored on those practice exams, she gets 100%. Irene instead works to understand the underlying concepts and formulas; she gets 92% on the practice questions because of occasional arithmetic slips, yet when the real final exam arrives with brand-new problems, Irene scores 90% while Ellie flounders.

Machine learning models face the same divide. Our ultimate goal is never simply to score well on the finite *training set* we already have; it is to make accurate predictions on unseen examples drawn from the broader population, a property called *generalization*. In our syn-

thetic linear regression earlier, we fitted 3 parameters to 1,000 examples, so memorization was impossible and training loss tracked validation loss closely. Once a model class becomes rich enough to bend around individual data points, however, driving training error to zero can make predictions on new data *worse*. In this section, we formalize the distinction between training error and generalization error, demonstrate *underfitting* and *overfitting* through polynomial curve fitting, unpack the *bias–variance decomposition*, and introduce *validation sets* and *K-fold cross-validation* for principled model selection.

2.6.1 Training Error and Generalization Error

In supervised learning, we assume that both our training examples and the future test examples we will encounter at deployment are drawn independently from the same underlying probability distribution $P(\mathbf{x}, y)$—the *independent and identically distributed* (i.i.d.) assumption.

Given a finite training dataset $\mathcal{D} = \{(\mathbf{x}^{(1)}, y^{(1)}), \dots, (\mathbf{x}^{(n)}, y^{(n)})\}$ of n examples drawn i.i.d. from P, the *training error* (or *empirical risk*) of a predictor f under loss l is the sample average over $\mathcal{D}$:

$$R_{\text{emp}}[f] = \frac{1}{n} \sum_{i=1}^{n} l\left(\mathbf{x}^{(i)}, y^{(i)}, f(\mathbf{x}^{(i)})\right). \tag{2.23}$$

By contrast, the *generalization error* (or *true risk* / *population risk*) is the expected loss on a fresh example $(\mathbf{x}, y)$ drawn from P:

$$R[f] = \mathbb{E}_{(\mathbf{x},y)\sim P}\Big[l(\mathbf{x}, y, f(\mathbf{x}))\Big] = \int l(\mathbf{x}, y, f(\mathbf{x}))\, dP(\mathbf{x}, y). \tag{2.24}$$

We want to find a function f that minimizes the true risk $R[f]$, yet we never have direct access to the distribution P: we only observe the finite sample $\mathcal{D}$, so our training algorithm minimizes $R_{\text{emp}}[f]$ instead (*empirical risk minimization*).

Why can $R_{\text{emp}}[\hat{f}]$ underestimate $R[\hat{f}]$ so severely? If we picked a *fixed* function f in advance, before looking at $\mathcal{D}$, then by the law of large numbers $R_{\text{emp}}[f]$ would be an unbiased estimate of $R[f]$ with fluctuations of order $1/\sqrt{n}$. However, our trained model $\hat{f}$ is *not* fixed independent of $\mathcal{D}$: it was specifically selected from an entire model class $\mathcal{F}$ to minimize error on $\mathcal{D}$. Whenever $\mathcal{F}$ is flexible enough to latch onto accidental noise quirks in $\mathcal{D}$, the training error $R_{\text{emp}}[\hat{f}]$ becomes optimistically biased downward relative to the true risk $R[\hat{f}]$ (Exercise 4). To measure $R[\hat{f}]$ honestly, we must evaluate $\hat{f}$ on an independent held-out *test set* (or *validation set*) $\mathcal{D}'$ that played no role in fitting its parameters.

2.6.2 Underfitting and Overfitting

Comparing training error against validation error diagnoses two opposite failure modes:

- **Underfitting (high bias):** The model is too rigid to capture the true underlying trend in the data, so *both* training error and validation error remain high and close together.

- **Overfitting (high variance):** The model is so flexible relative to the dataset size that it fits the random noise in the training examples in addition to the signal; training error becomes tiny while validation error climbs much higher, opening a large *generalization gap* $R[\hat{f}] - R_{\text{emp}}[\hat{f}]$.

2.6.2.1 Polynomial Curve Fitting

We can watch underfitting, balanced fitting, and catastrophic overfitting unfold without leaving linear regression. Suppose a scalar input x is drawn uniformly from $[-1, 1]$ and the true label is generated by a smooth curve plus Gaussian noise,

$$y = f^*(x) + \epsilon, \qquad f^*(x) = 5 + 1.2x - 3.4x^2 + 5.6x^3, \qquad \epsilon \sim \mathcal{N}(0, \sigma^2), \tag{2.25}$$

with $\sigma = 0.1$. Note that the irreducible population noise floor for the squared error $(\hat{y} - y)^2$ is $\sigma^2 = 0.01$: even the true function $f^*(x)$ incurs an expected squared error of 0.01 because of the unpredictable noise ϵ.

To fit a curved function using linear regression, we map the scalar input x to a vector of monomial *polynomial features* $(1, x, x^2, \dots, x^k)^\top \in \mathbb{R}^{k+1}$. A degree-$k$ polynomial

$$\hat{y} = \sum_{j=0}^{k} w_j x^j \tag{2.26}$$

is nonlinear in x, yet **linear in the weights** $\mathbf{w} = (w_0, \dots, w_k)^\top$. Thus we can fit any degree-k polynomial on n training points by constructing the Vandermonde design matrix $\mathbf{X} \in \mathbb{R}^{n \times (k+1)}$ with entries $X_{ij} = (x^{(i)})^j$ and calling `lstsq` (just as we did in Section 2.1.2.1). Let us generate $n = 20$ training points and $1{,}000$ test points (enough to estimate test error accurately), and then compare three degrees: $k = 1$ (a straight line), $k = 3$ (a cubic), and $k = 19$ (a degree-19 polynomial with 20 parameters for 20 points):

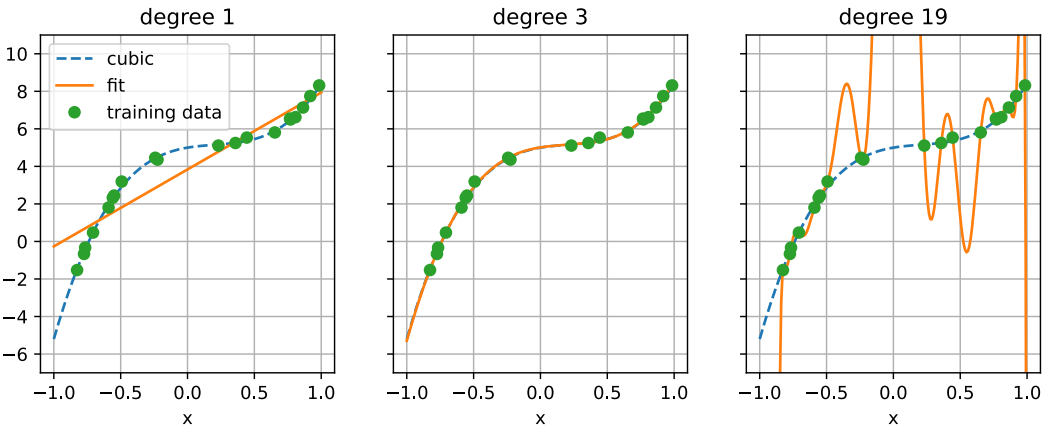
```
def target(x):
    """The cubic f* that generates the labels."""
    return 5 + 1.2 * x - 3.4 * x**2 + 5.6 * x**3

def make_data(rng, n, sigma):
    """Draw n inputs uniformly from [-1, 1] and label them by the noisy cubic."""
    x = rng.uniform(-1, 1, n)
    return x, target(x) + rng.normal(0, sigma, n)

sigma = 0.1                                 # standard deviation of the noise
rng = np.random.default_rng()               # a new random draw on every run
x_train, y_train = make_data(rng, 20, sigma)  # few, so high degrees overfit
x_test, y_test = make_data(rng, 1000, sigma)
```

Below, `features(x, d)` builds the design matrix $[1, x, \dots, x^d]$, `fit_degree` finds the exact least-squares weights, and `mse` computes the plain mean squared error (omitting the factor of $\frac{1}{2}$ so the noise floor sits neatly at $\sigma^2 = 0.01$):

```
def features(x, d):
    """Design matrix whose columns are the powers x^0, x^1, ..., x^d."""
    return x.reshape(-1, 1) ** np.arange(d + 1)

def fit_degree(x, y, d):
    """Least-squares weights of the degree-d polynomial fitted to (x, y)."""
    return np.linalg.lstsq(features(x, d), y, rcond=None)[0]

def mse(w, x, y):
    """Mean squared error of the polynomial with weights w on (x, y)."""
    return ((features(x, len(w) - 1) @ w - y) ** 2).mean()
```

Let us plot the fits of degree 1, 3, and 19 against the 20 noisy training points and the true cubic:

```
xs = np.linspace(-1, 1, 200)
_, axes = d2l.plt.subplots(1, 3, figsize=(10, 3.4), sharey=True)
for ax, d in zip(axes, (1, 3, 19)):
    w = fit_degree(x_train, y_train, d)
    d2l.plot([xs, xs, x_train], [target(xs), features(xs, d) @ w, y_train],
             xlabel='x', ylim=(-7, 11), fmts=('--', '-', 'o'),
             legend=['cubic', 'fit', 'training data'] if d == 1 else None,
             axes=ax)
    ax.set_title(f'degree {d}')
```



Those three panels tell the whole story of the two classic failure modes of supervised learning:

- **Degree $k = 1$ (underfitting):** A straight line cannot bend to follow the S-shaped cubic; it misses training and test points alike.
- **Degree $k = 3$ (well matched):** A cubic has exactly the flexibility of f^*; its fit is nearly indistinguishable from the true curve.
- **Degree $k = 19$ (overfitting):** With $k + 1 = 20$ parameters and $n = 20$ distinct training points, the polynomial has enough degrees of freedom to interpolate every single training point (Exercise 2). Between the data points, however, the curve oscillates wildly, sending the test error soaring by orders of magnitude.

What happens to the learned weights w_j themselves when the polynomial overfits? Let us print the training error, the test error, and the largest weight magnitude $\max_j |\hat{w}_j|$ for each of the three degrees:

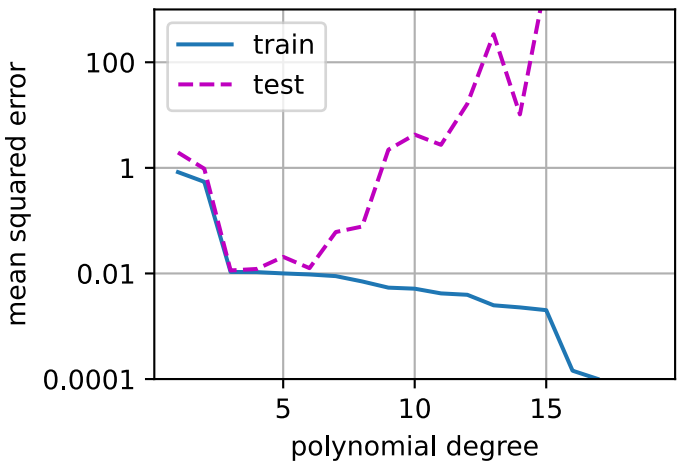
```
for d in (1, 3, 19):
    w = fit_degree(x_train, y_train, d)
    print(f'degree {d:2d}: train {mse(w, x_train, y_train):.2g}, '
          f'test {mse(w, x_test, y_test):.2g}, '
          f'largest |weight| {np.abs(w).max():.2g}')
```

```
degree  1: train 0.83, test 2, largest |weight| 4.1
degree  3: train 0.011, test 0.011, largest |weight| 5.8
degree 19: train 2.4e-19, test 2.9e+06, largest |weight| 9.4e+06
```

Although exact numbers vary slightly with each random draw, the contrast is dramatic. Degree 1 suffers large errors on both splits (a test MSE around 2, roughly $200\times$ the $\sigma^2 = 0.01$ noise floor). Degree 3 pushes both training and test errors right down to the $\sigma^2 = 0.01$ floor, with its largest weight sitting near the true leading coefficient 5.6. Degree 19 drives training error far below the noise floor—essentially to zero—while its test error explodes by many orders of magnitude. Look at its weights to see the mechanism: degree 19 relies on huge positive and negative coefficients that delicately cancel at the 20 training inputs and blow up everywhere else.

Sweeping across every degree from 1 to 19 maps out the full transition on a logarithmic scale:

```
degrees = list(range(1, 20))
weights = [fit_degree(x_train, y_train, d) for d in degrees]
train_err = [mse(w, x_train, y_train) for w in weights]
test_err = [mse(w, x_test, y_test) for w in weights]
d2l.plot(degrees, [train_err, test_err], xlabel='polynomial degree',
         ylabel='mean squared error', legend=['train', 'test'],
         yscale='log', ylim=(1e-4, 1e3))
```



As k increases, the training error decreases monotonically (since adding x^{k+1} only enlarges

the subspace of functions available to least squares), plunging far below σ^2 by $k = 19$. The test error, by contrast, traces a classic **U-shaped curve**: it drops steeply to its minimum at $k = 3$ (or a nearby degree) as underfitting disappears, and then climbs by orders of magnitude as extra powers of x fit the random noise in our 20 training points. Because x^{17} and x^{19} look almost identical on $[-1, 1]$, the high-degree fit cancels giant positive multiples of one against giant negative multiples of the other to thread the 20 noisy points, an instability that we cure directly with weight decay in Section 2.7.

2.6.2.2 Bias and Variance

Why does model complexity produce a U-shaped generalization curve? Let us decompose the expected squared error of a model $\hat{f}_{\mathcal{D}}$ trained on a random dataset $\mathcal{D}$ at a test point x where $y = f^*(x) + \epsilon$ (with $\mathbb{E}[\epsilon] = 0$, $\mathrm{Var}(\epsilon) = \sigma^2$, and ϵ independent of $\mathcal{D}$). Write $\bar{f}(x) = \mathbb{E}_{\mathcal{D}}[\hat{f}_{\mathcal{D}}(x)]$ for the average predictor across all possible training sets of size n. Adding and subtracting $f^*(x)$ and $\bar{f}(x)$ inside $\mathbb{E}_{\mathcal{D},\epsilon}[(\hat{f}_{\mathcal{D}}(x) - y)^2]$ and noting that the cross terms vanish yields the famous *bias–variance decomposition*:

$$\mathbb{E}_{\mathcal{D},\epsilon}\left[\left(\hat{f}_{\mathcal{D}}(x) - y\right)^2\right] = \underbrace{\left(\bar{f}(x) - f^*(x)\right)^2}_{\mathrm{bias}^2(x)} + \underbrace{\mathbb{E}_{\mathcal{D}}\left[\left(\hat{f}_{\mathcal{D}}(x) - \bar{f}(x)\right)^2\right]}_{\mathrm{variance}(x)} + \underbrace{\sigma^2}_{\mathrm{noise}}. \tag{2.27}$$

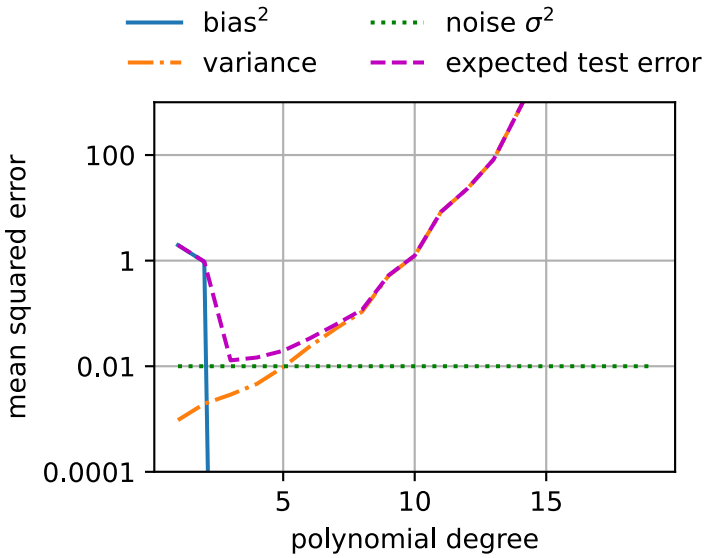
Every model's expected test error is the sum of three nonnegative pieces:

1. **Squared bias** $(\bar{f}(x) - f^*(x))^2$**:** How far the *average* fitted model (even averaged over infinitely many training datasets) sits from the true function $f^*(x)$. Rigid models (like a straight line $k = 1$ on a cubic) have high bias.
2. **Variance** $\mathbb{E}_{\mathcal{D}}[(\hat{f}_{\mathcal{D}}(x) - \bar{f}(x))^2]$**:** How sensitively the fitted model fluctuates from one random training sample $\mathcal{D}$ to the next. Overly flexible models ($k = 12$) swing wildly with every new draw of 20 points, suffering huge variance.
3. **Irreducible noise** σ^2**:** The variance of the label y around $f^*(x)$ itself, which no predictor can ever beat.

In our polynomial experiment we can measure all three terms directly. Let us keep the 20 training inputs fixed and redraw only their label noise. Because the least-squares weights depend linearly on $\mathbf{y}$ and the noise has mean zero, the average fit $\bar{f}$ is simply the fit to the noiseless labels $f^*(x^{(i)})$, which gives the squared bias exactly; refitting across 200 independent draws of the training noise estimates the variance at every test point:

```
noise = rng.normal(0, sigma, (len(x_train), 200))   # 200 draws of the noise
bias2, var = [], []
for d in degrees:
    w_avg = fit_degree(x_train, target(x_train), d)              # average fit
    W = fit_degree(x_train, target(x_train)[:, None] + noise, d)  # fit per draw
    bias2.append(mse(w_avg, x_test, target(x_test)))
    var.append((features(x_test, d) @ W).var(axis=1).mean())
d2l.plot(degrees, [bias2, var, [sigma**2] * len(degrees),
```

```
                    np.add(bias2, var) + sigma**2],
          xlabel='polynomial degree', ylabel='mean squared error',
          fmts=('-', 'C1-.', 'g:', 'm--'), yscale='log', ylim=(1e-4, 1e3))
d2l.plt.legend(['bias$^2$', 'variance', r'noise $\sigma^2$',
                'expected test error'], ncol=2, frameon=False,
               loc='lower center', bbox_to_anchor=(0.5, 1));  # above the plot
```



Look how cleanly the curves explain the U-shape of test error: squared bias is large at degrees 1 and 2, then collapses to zero at degree 3 (the first family rich enough to contain the cubic). Variance, meanwhile, climbs with every added degree, starting around $\sigma^2(k+1)/n$ (the noise variance scaled by parameters per example; see exercise 4) and eventually rising orders of magnitude past the noise floor. Underfitting is error dominated by *bias*; overfitting is error dominated by *variance*. The idealized trade-off shown in Figure 2.11 recurs across classical statistics and machine learning: training error drops steadily with complexity, while test error reaches its minimum at an intermediate sweet spot (never dropping below σ^2).

2.6.2.3 Model Complexity

As Figure 2.11 summarizes schematically, increasing model complexity (capacity) reduces bias while increasing variance. The optimal generalization performance sits at the sweet spot where the sum of squared bias and variance reaches its minimum. In classical parametric models, complexity is governed by the number of parameters p and the magnitude of the weights $\|\mathbf{w}\|$; in deep learning (Section 4.5), we will see how regularization, architecture design, and dataset scale reshape this curve.

2.6.2.4 Dataset Size

Besides restricting model complexity, the most direct way to crush variance is to collect *more training data*. For a fixed polynomial degree k, fitting to $n = 1{,}000$ points instead of

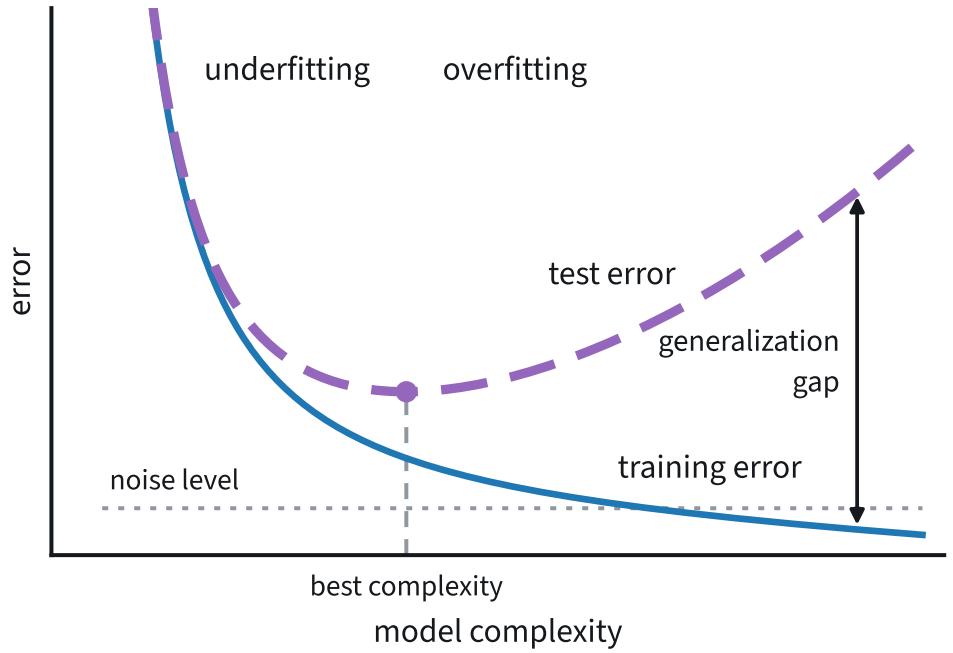


Figure 2.11: Training and test error against model complexity. To the left of the best complexity the model underfits and both errors are high; to the right it overfits, its training error falls below the noise level, and the generalization gap widens.

$n = 20$ averages out the random noise and pins down even high-degree coefficients cleanly (with estimation variance decaying as $(k+1)\sigma^2/n$). Let us watch how degrees 1, 3, and 19 respond as we grow a fresh training set from $n = 20$ to $n = 1{,}000$ examples:

```
for n in (20, 50, 100, 1000):
    x, y = make_data(rng, n, sigma)          # a fresh training set of size n
    errs = [mse(fit_degree(x, y, d), x_test, y_test) for d in (1, 3, 19)]
    print(f'n = {n:4d}: test error of degrees 1, 3, 19:',
          ', '.join(f'{e:.2g}' for e in errs))
```

```
n =   20: test error of degrees 1, 3, 19: 1.8, 0.013, 1.4e+04
n =   50: test error of degrees 1, 3, 19: 1.9, 0.011, 0.37
n =  100: test error of degrees 1, 3, 19: 1.9, 0.011, 0.014
n = 1000: test error of degrees 1, 3, 19: 1.9, 0.01, 0.011
```

Look at the contrast. Growing the dataset rescues the overfitting degree-19 polynomial dramatically: its test error plunges by orders of magnitude by $n = 100$, and by $n = 1{,}000$ it matches the true cubic. Meanwhile, the underfitting straight line barely budges (its test error remains stuck around 2): collecting more data shrinks *variance*, but it can never cure *bias*. A model is never "too complex" in isolation; it is only too complex *relative to the signal and data available to constrain it*. That scaling principle is one major reason deep learning surged once massive digital datasets became commonplace, though deep networks will add important twists to the classical curve in Section 4.5.

2.6.3 Model Selection

2.6.3.1 Validation Sets

In our synthetic experiment we knew $f^*(x)$ and generated 1,000 test points at will. In a real problem, how should we choose hyperparameters, such as the polynomial degree k, the learning rate η, or the weight decay strength λ?

We can never choose hyperparameters by minimizing training error, because training error always favors the most complex, memorization-prone model ($k = 19$). Nor should we tune hyperparameters by repeatedly consulting the final *test set*, which would leak test-set noise into our design choices (Chapter 3.5). Instead, we partition our available data into three disjoint splits:

1. **Training set:** Used to fit the model parameters $(\mathbf{w}, b)$ for each candidate hyperparameter setting.
2. **Validation set:** Used to compare candidate hyperparameter settings and select the winner (*model selection*).
3. **Test set:** Held in a vault and evaluated *once* at the very end to report an unbiased estimate of the selected model's generalization error.

2.6.3.2 Cross-Validation

When a dataset is small (say, $n = 20$ or $n = 100$ clinical records), carving off a fixed 20% validation set leaves too few examples to train on and too few to validate reliably. *K-fold cross-validation* (Figure 2.12) makes every example do double duty. We partition the dataset randomly into K equal-sized disjoint subsets (*folds*) $\mathcal{D}_1, \dots, \mathcal{D}_K$ (typically $K = 5$ or $K = 10$). In round $i \in \{1, \dots, K\}$, we train the model from scratch on the $K - 1$ folds $\mathcal{D} \setminus \mathcal{D}_i$ and evaluate its validation loss on the held-out fold $\mathcal{D}_i$. Averaging the K held-out losses gives the K-fold cross-validation score, and once the winning hyperparameter is chosen, we retrain the model on all n examples. (In the extreme case $K = n$, called *leave-one-out cross-validation*, each fold is a single data point; for linear least squares, a remarkable algebraic identity computes all n leave-one-out residuals from a *single* fit without retraining n times—see Exercise 6.)

Folds must come from a *random* permutation, because real datasets often arrive sorted, and with folds of only four points a single split is noisy. So `kfold_splits` below draws ten independent random partitions into $K = 5$ folds (*repeated* cross-validation), and `cv_error` averages the validation error of a polynomial fit over all of their folds:

```
def kfold_splits(rng, n, K=5, repeats=10):
    """Random partitions of the indices 0, ..., n-1 into K folds."""
    return [np.array_split(rng.permutation(n), K) for _ in range(repeats)]

def cv_error(x, y, d, splits):
    """Validation error of the degree-d fit, averaged over all folds."""
    return np.mean([mse(fit_degree(np.delete(x, f), np.delete(y, f), d),
                        x[f], y[f]) for folds in splits for f in folds])
```

Let us select the polynomial degree using strictly the 20 training examples, evaluating every degree on the same folds for a fair comparison, and check how closely that choice matches the true test-set optimum (which cross-validation never sees):

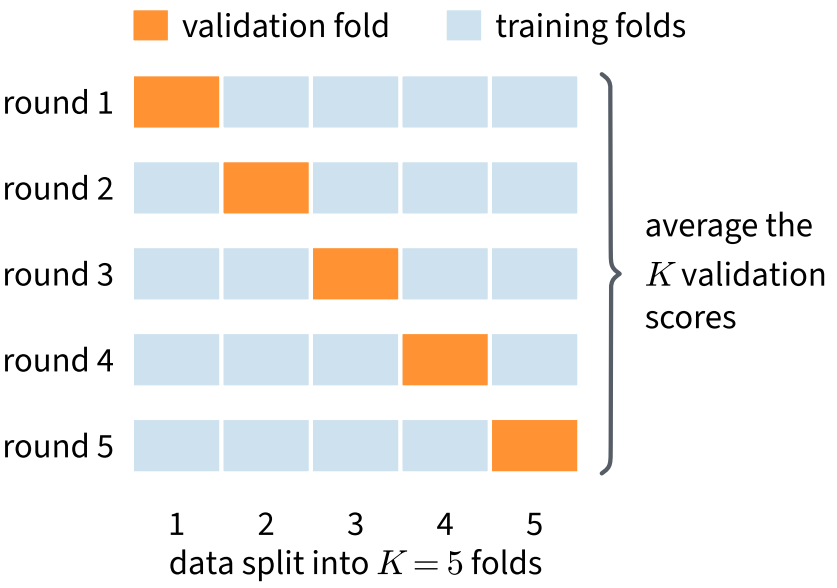


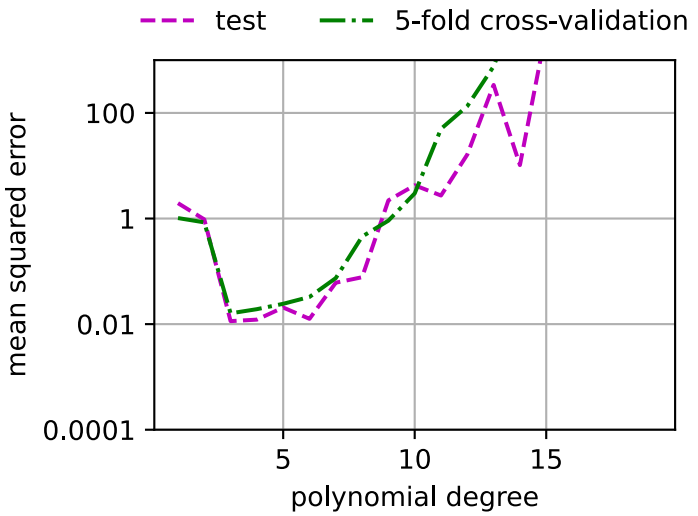
Figure 2.12: In K-fold cross-validation ($K = 5$ shown), the dataset is partitioned into K folds; each round holds out one fold for validation (orange) and trains on the remaining $K - 1$ folds.

```
splits = kfold_splits(rng, len(x_train))  # the same folds for every degree
cv_err = [cv_error(x_train, y_train, d, splits) for d in degrees]
print('cross-validation picks degree:', degrees[np.argmin(cv_err)])
print('lowest test error at degree:  ', degrees[np.argmin(test_err)])
```

```
cross-validation picks degree: 3
lowest test error at degree:   3
```

Let us plot the 5-fold cross-validation curve right alongside the test-error curve across all 19 degrees:

```
d2l.plot(degrees, [test_err, cv_err], xlabel='polynomial degree',
         ylabel='mean squared error', fmts=('m--', 'g-.'), yscale='log',
         ylim=(1e-4, 1e3))
d2l.plt.legend(['test', '5-fold cross-validation'], ncol=2, frameon=False,
               loc='lower center', bbox_to_anchor=(0.5, 1));  # above the plot
```



Without ever glancing at the test set, repeated 5-fold cross-validation reliably locates the bottom of the test-error bowl, usually selecting degree 3 or occasionally a nearby degree whose test error is nearly indistinguishable. Near the minimum, the two curves track within a small factor, with cross-validation naturally a bit noisier since it averages over 20 points instead of 1,000. At high degrees both curves shoot up by orders of magnitude. Cross-validation has a reason to rise even earlier than the test error: each fold trains on only 16 points rather than 20, so degree 15 already interpolates all 16 training points, and any validation point near the edges of $[-1, 1]$ falls outside the fitted range where high-degree polynomials explode. With a single draw of 20 points, however, either curve may be the first to rise.

How should we pick K? Because every fold trains on only a fraction $(K-1)/K$ of the dataset, its fitted model is slightly worse than a model trained on all n examples, giving K-fold cross-validation a slight pessimistic bias (exercise 5 works it out exactly). Pushing $K = n$ (*leave-one-out* cross-validation) nearly eliminates that bias. For a general learner, however, it demands n separate fits per candidate. And for an unstable learner such as a decision tree, where removing a single example can change the whole model, its n nearly identical training sets can make the estimate noisier than that of a coarser split. This is why $K = 5$ or $K = 10$ is the usual default when each fit is expensive (Kohavi 1995). Least squares is neither expensive nor unstable: a single fit yields all n leave-one-out residuals, and the fit changes smoothly with the data.

Let us measure the bias and the noise for our cubic. On each of 500 fresh training sets of 20 points, we fit the cubic to all 20 points and record its test error, together with three estimates of that error that use the 20 points alone: leave-one-out ($K = 20$, one point per fold), 5-fold on a single random split, and 5-fold repeated over ten splits:

```
loo = kfold_splits(rng, 20, K=20, repeats=1)  # K = n: one point per fold
est = {'test error': [], 'leave-one-out': [], '5-fold, 1 split': [],
       '5-fold, 10 splits': []}
for _ in range(500):                          # 500 fresh training sets
    x, y = make_data(rng, 20, sigma)
```

```
        est['test error'].append(mse(fit_degree(x, y, 3), x_test, y_test))
        est['leave-one-out'].append(cv_error(x, y, 3, loo))
        est['5-fold, 1 split'].append(
            cv_error(x, y, 3, kfold_splits(rng, 20, repeats=1)))
        est['5-fold, 10 splits'].append(cv_error(x, y, 3, kfold_splits(rng, 20)))
    for name, e in est.items():
        print(f'{name:18s} mean {np.mean(e):.4f}   std {np.std(e):.4f}')
```

```
test error         mean 0.0135   std 0.0045
leave-one-out      mean 0.0149   std 0.0252
5-fold, 1 split    mean 0.0200   std 0.0519
5-fold, 10 splits  mean 0.0175   std 0.0320
```

On average, leave-one-out lands close to the test error, within about a tenth of it. Both 5-fold estimates sit above the leave-one-out estimate, reflecting the pessimistic bias of training on 16 points instead of 19. Among the three cross-validation estimates, leave-one-out also has the smallest standard deviation across the 500 draws, so here the folklore that it is the noisier estimate does not hold. Repeating the 5-fold split ten times narrows its spread, but not down to that of leave-one-out: much of the spread comes from the luck of which 20 points were drawn, not from how they were split. For least squares, then, leave-one-out is both cheap and the most accurate of the three; for the networks of later chapters, where a single fit takes minutes or hours, a coarse split remains the practical choice.

Selecting a discrete polynomial degree worked here because our features came in a natural hierarchy $1, x, x^2, \dots, x^k$. What should we do when a dataset arrives with hundreds of unordered features and no obvious way to drop columns one by one? In Section 2.7, we replace discrete feature selection with a continuous penalty on the weight norm.

2.6.4 Summary

Supervised learning minimizes empirical risk $R_{\text{emp}}[f]$ on a finite training sample while aiming for low population risk $R[f] = \mathbb{E}[l(f(\mathbf{x}), y)]$ on unseen data. Because optimization chooses $\hat{f}$ specifically to fit the training set, $R_{\text{emp}}[\hat{f}]$ is optimistically biased downward whenever the model class is flexible enough to absorb label noise. Expected squared test error decomposes into squared bias, estimation variance, and irreducible noise σ^2: rigid models underfit with high bias, overly flexible models on small samples overfit with high variance, and collecting more training data shrinks variance as $1/n$ without reducing bias. Because training error always favors memorization and repeated queries contaminate the final test set, hyperparameters must be chosen on a held-out validation split or via K-fold cross-validation.

2.6.5 Exercises

1. **When IID fails.** Give three examples in which dependent observations make treating a dataset as IID inadvisable. For each, state whether a random train/validation split overestimates or underestimates the generalization error, and describe a split that avoids the problem.

2. **Exact polynomial regression.** Show that when the n training inputs $x^{(1)}, \dots, x^{(n)}$ are distinct, a polynomial of degree $n - 1$ fits the n labels exactly, and explain where

distinctness is used. What does least squares return when two inputs coincide but their labels differ?

3. code **A better basis for degree 19.** The high powers x^{17} and x^{19} differ by less than 0.05 anywhere on $[-1, 1]$, so the columns of `features(x, 19)` are nearly parallel. The Chebyshev polynomials $T_0, \dots, T_{19}$ span the same functions but stay between -1 and 1 on the interval.

    1. Predict first: if we fit degree 19 to the 20 training points in the Chebyshev basis, which of the training error, the test error, and the largest weight will change, and by how much?
    2. Check your prediction over several runs: replace `features(x, 19)` by `np.polynomial.chebyshev.chebvander(x, 19)` for both the training and the test inputs, and compare the two design matrices' condition numbers (`np.linalg.cond`), the training and test errors, and the largest weights.
    3. Explain why the two fits are the same polynomial in exact arithmetic, and which of your numbers differ in float64 and why. Does the better basis rescue degree 19 on 20 points? What does your answer say about the size of the weights as a measure of complexity?

4. **Optimism of the training error.** ● Consider least squares with a fixed $n \times p$ design matrix $\mathbf{X}$ of full column rank and labels $\mathbf{y} = \mathbf{X}\mathbf{w}^* + \epsilon$, whose noise terms are independent with mean zero and variance σ^2. The fitted predictions are $\hat{\mathbf{y}} = \mathbf{H}\mathbf{y}$ with $\mathbf{H} = \mathbf{X}(\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top$.

    1. Show that the expected training error is $\mathbb{E}[\|\mathbf{y} - \hat{\mathbf{y}}\|^2]/n = \sigma^2(n-p)/n$. Hint: $\mathbf{y} - \hat{\mathbf{y}} = (\mathbf{I} - \mathbf{H})\epsilon$, and the trace of $\mathbf{I} - \mathbf{H}$ is $n - p$.
    2. Draw fresh labels $\mathbf{y}' = \mathbf{X}\mathbf{w}^* + \epsilon'$ at the same inputs. Show that the expected error of $\hat{\mathbf{y}}$ on them is $\sigma^2(n+p)/n$, so that the training error underestimates it by $2\sigma^2 p/n$. Show that $\sigma^2 p/n$ is the variance of the predictions, averaged over the n inputs.
    3. Evaluate both expressions for the cubic fit of Section 2.6.2.1 ($n = 20$, $p = 4$, $\sigma = 0.1$) and compare them with the training and test errors of degree 3 over several runs of the experiment. Explain the differences: how much does the training error of a single run spread around its expectation, and why does the formula of (b), which holds at the training inputs, not apply exactly to the test inputs?

5. K**-fold cross-validation.**

    1. Suppose that a model trained on m examples has expected generalization error $R(m) = R_\infty + c/m$ with $c > 0$. Each fold's model trains on $(K-1)n/K$ examples. Compute the bias of the K-fold estimate relative to $R(n)$, the error of the model trained on all n examples. Express the bias as a fraction of $R(n) - R_\infty$ and evaluate this fraction for $K = 2$, 5, 10, and n.
    2. After selecting a hyperparameter by cross-validation, the final model is retrained on all n examples. Explain why a hyperparameter whose best value depends on the training-set size complicates this step, and give an example.

6. **Leave-one-out shortcut.** ● Leave-one-out cross-validation requires n model fits for a general learner. Consider least squares with the design matrix $\mathbf{X}$, whose rows are $\mathbf{x}^{(i)\top}$. Let $\mathbf{A} = \mathbf{X}^\top\mathbf{X}$ and assume that it remains invertible when any one row of $\mathbf{X}$ is removed. Let $\hat{y}^{(i)}$ be the prediction at $\mathbf{x}^{(i)}$ of the fit on all n examples, let $\hat{y}^{(-i)}$ be the prediction at $\mathbf{x}^{(i)}$ of the fit that omits example i, and let $h_i = \mathbf{x}^{(i)\top}\mathbf{A}^{-1}\mathbf{x}^{(i)}$. Use the

Sherman–Morrison identity (Section 33.2)

$$(\mathbf{A} - \mathbf{u}\mathbf{u}^\top)^{-1} = \mathbf{A}^{-1} + \frac{\mathbf{A}^{-1}\mathbf{u}\mathbf{u}^\top\mathbf{A}^{-1}}{1 - \mathbf{u}^\top\mathbf{A}^{-1}\mathbf{u}}$$

with $\mathbf{u} = \mathbf{x}^{(i)}$ to show that

$$y^{(i)} - \hat{y}^{(-i)} = \frac{y^{(i)} - \hat{y}^{(i)}}{1 - h_i}. \tag{2.28}$$

State what leave-one-out cross-validation then costs for least squares compared with a single fit.

2.7 Weight Decay

In Section 2.6, we controlled model complexity in the bluntest way possible: by changing the discrete number of polynomial features $k + 1$. Yet discrete feature selection quickly hits a wall. In high-dimensional problems with hundreds or thousands of features, there are 2^d possible feature subsets, and worse still, we saw that when a polynomial overfits, its weights w_j explode into the millions and beyond to cancel almost-collinear features against one another.

To see the core pathology in its simplest two-feature form, imagine predicting room temperature from two noisy thermometers placed side by side, x_1 and x_2, each reading the true temperature plus a tiny independent noise wiggle. Consider two linear models whose weights sum to 2:

- **Model A** sets $\mathbf{w} = (1, 1)^\top$, averaging the two sensors and cutting the sensor noise variance in half.
- **Model B** sets $\mathbf{w} = (501, -499)^\top$. Whenever the two thermometers agree, $501x_1 - 499x_2 = 2x_1$ gives the same average, yet any tiny discrepancy $x_1 - x_2$ between the two sensors gets amplified by 500.

On a small training sample where $x_1 - x_2$ happened by chance to correlate slightly with the label noise, unregularized least squares will choose $(501, -499)^\top$ to shave a sliver off the training error, only to fail badly on new readings. How can we keep all d features in the model while preventing the weights from blowing up along poorly determined directions?

The classic remedy, known as *ridge regression* or *Tikhonov regularization* in statistics (Hoerl and Kennard 1970; Tikhonov and Arsenin 1977) and *weight decay* in deep learning, adds a smooth penalty on the squared Euclidean norm $\|\mathbf{w}\|_2^2$ directly to the loss function. In this section, we derive weight decay from four complementary perspectives (penalized optimization, per-step multiplicative decay in SGD, Bayesian Gaussian priors, and spectral shrinkage along eigenvectors of $\mathbf{X}^\top\mathbf{X}$) and implement it both from scratch and concisely in all four frameworks.

2.7.1 Norm Penalties

2.7.1.1 The Penalized Loss

Rather than restricting the number of features, we control a linear model's effective capacity by penalizing the squared ℓ_2 norm of its weight vector $\|\mathbf{w}\|_2^2 = \sum_{j=1}^{d} w_j^2$. Adding this penalty to the mean squared error loss $L(\mathbf{w}, b)$ with a nonnegative *regularization constant* $\lambda \geq 0$ yields the regularized objective:

$$L(\mathbf{w}, b) + \frac{\lambda}{2}\|\mathbf{w}\|_2^2. \tag{2.29}$$

Once again, the factor of $\frac{1}{2}$ cancels the 2 when we differentiate $\|\mathbf{w}\|_2^2$. Notice three basic properties of Equation 2.29:

1. **Trade-off via λ:** Setting $\lambda = 0$ recovers unregularized least squares; increasing λ places a steeper tax on large weights, pulling $\mathbf{w}$ toward $\mathbf{0}$ (and driving $\mathbf{w} \to \mathbf{0}$ as $\lambda \to \infty$).
2. **Why we leave the bias b unpenalized:** The bias b simply shifts the average output level to match the mean of the target y; shrinking b toward zero would bias every prediction toward zero if y were measured in, say, Kelvin instead of Celsius, without reducing the model's sensitivity to input perturbations at all.
3. **Feature scaling matters:** Because $\sum_j w_j^2$ sums raw squared coefficients across coordinates, re-expressing one feature in millimeters instead of meters divides its weight w_j by $1{,}000$ and cuts its penalty contribution by 10^6. In practice, one standardizes features to comparable scales (zero mean, unit variance) before applying an isotropic $\|\mathbf{w}\|_2^2$ penalty.

Why penalize the squared ℓ_2 norm? In our two-thermometer example, $(1, 1)^\top$ has $\|\mathbf{w}\|_2^2 = 2$, whereas $(501, -499)^\top$ has $\|\mathbf{w}\|_2^2 = 500{,}002$, so even a tiny $\lambda > 0$ strongly prefers averaging the two sensors over subtracting them.

2.7.1.2 Weight Decay in the Gradient Step

How does the penalty $\frac{\lambda}{2}\|\mathbf{w}\|_2^2$ change our minibatch SGD update rule? Since $\nabla_{\mathbf{w}} \left(\frac{\lambda}{2}\|\mathbf{w}\|_2^2\right) = \lambda\mathbf{w}$, the penalized gradient step is

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \left(\frac{\partial L}{\partial \mathbf{w}} + \lambda\mathbf{w}\right) = (1 - \eta\lambda)\mathbf{w} - \eta\frac{\partial L}{\partial \mathbf{w}}. \tag{2.30}$$

Look at the first term on the right: **before taking the usual data-driven gradient step $-\eta\, \partial L/\partial \mathbf{w}$, we multiply the weight vector by $(1 - \eta\lambda) < 1$, shrinking ("decaying") it slightly toward zero on every update.** That multiplicative shrinkage factor $(1 - \eta\lambda)$ is why deep learning practitioners call ℓ_2 regularization *weight decay*. (For plain SGD, adding $\frac{\lambda}{2}\|\mathbf{w}\|_2^2$ to the loss and decaying weights by $(1 - \eta\lambda)$ are algebraically identical; for adaptive optimizers like Adam in Chapter 9, decoupling multiplicative weight decay into *AdamW* makes an important difference.)

2.7.1.3 The ℓ_1 Penalty and Sparsity

What if we replaced the squared ℓ_2 norm $\|\mathbf{w}\|_2^2 = \sum_j w_j^2$ with the ℓ_1 norm $\|\mathbf{w}\|_1 = \sum_j |w_j|$, a method known in statistics as the *Lasso* (Tibshirani 1996)? Comparing the geometry of their constraint sets (Figure 2.13) reveals a striking difference. Minimizing the data loss $L(\mathbf{w})$ subject to $\|\mathbf{w}\|_2^2 \le r^2$ searches inside a smooth round disk (or sphere in $\mathbb{R}^d$), where the elliptical loss contours almost always touch the boundary at a point where *every* coordinate w_j is small but nonzero. Minimizing $L(\mathbf{w})$ subject to $\|\mathbf{w}\|_1 \le r$, by contrast, searches inside a sharp-cornered diamond (cross-polytope) whose vertices lie directly on the coordinate axes. Elliptical loss contours frequently collide with one of those sharp corners first, setting many coordinates w_j to *exact zero* and performing automatic feature selection (*sparsity*).

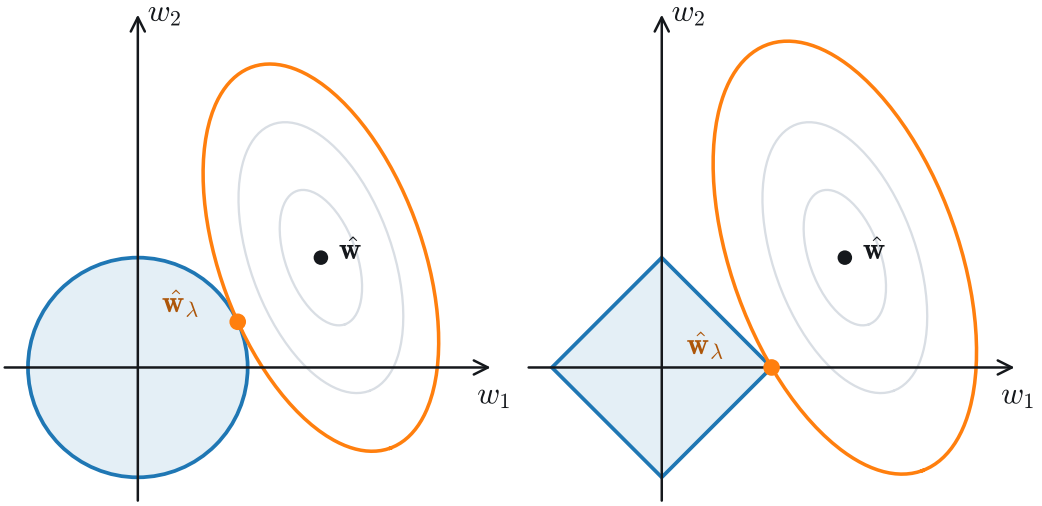


Figure 2.13: Constraint geometry of ℓ_2 ridge regression (left) vs. ℓ_1 Lasso regression (right). Elliptical contours of the data loss $L(\mathbf{w})$ expanding from the unregularized minimum $\hat{\mathbf{w}}$ typically meet the smooth ℓ_2 circle away from the axes (keeping both weights nonzero), whereas they often hit a sharp corner of the ℓ_1 diamond right on an axis (setting $w_2 = 0$).

2.7.1.4 A Gaussian Prior on the Weights

Just as maximum likelihood under Gaussian observation noise gave us the squared data loss in Section 2.1.3, Bayesian statistics gives weight decay an equally clean probabilistic interpretation. Suppose that before observing any data, we place an isotropic zero-mean Gaussian *prior distribution* on the weights, $w_j \sim \mathcal{N}(0, \tau^2)$ independently, so $p(\mathbf{w}) \propto \exp(-\|\mathbf{w}\|_2^2/(2\tau^2))$, while the labels follow $y \mid \mathbf{x}, \mathbf{w}, b \sim \mathcal{N}(\mathbf{w}^\top \mathbf{x} + b, \sigma^2)$.

By Bayes' theorem, the *posterior density* $p(\mathbf{w} \mid \mathbf{X}, \mathbf{y}) \propto p(\mathbf{y} \mid \mathbf{X}, \mathbf{w}, b)\, p(\mathbf{w})$ is maximized by minimizing its negative logarithm (*maximum a posteriori* or MAP estimation):

$$\frac{1}{2\sigma^2} \sum_{i=1}^{n} \left(y^{(i)} - \mathbf{w}^\top \mathbf{x}^{(i)} - b\right)^2 + \frac{1}{2\tau^2} \|\mathbf{w}\|_2^2. \tag{2.31}$$

Dividing through by n/σ^2 recovers our weight-decay objective Equation 2.29 with $\lambda =$

$\sigma^2/(n\tau^2)$. Notice how sensible that ratio is: stronger prior belief that weights are small (smaller τ^2) or noisier observations (larger σ^2) calls for stronger regularization λ, whereas as the sample size $n \to \infty$, the penalty $\lambda = \sigma^2/(n\tau^2) \to 0$ because abundant data eventually speak for themselves.

2.7.2 Weight Decay from Scratch

To see weight decay rescue a model with nearly as many weights as training examples, let us construct a high-dimensional synthetic regression problem where the dimension $d = 200$ comes close to the training sample size $n_{\text{train}} = 300$. Specifically, we draw inputs $\mathbf{x} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}_{200})$ and generate labels from

$$y = 0.05 + \sum_{j=1}^{200} 0.1\, x_j + \epsilon, \qquad \epsilon \sim \mathcal{N}(0, 1), \tag{2.32}$$

with 300 training examples (in minibatches of 20) and 1,000 validation examples, using `SyntheticRegressionData` from Section 2.4. The true weights $\mathbf{w}^* = (0.1, \ldots, 0.1)^\top$ have coordinate scale $\tau = 0.1$, and the noise is substantial: its standard deviation $\sigma = 1$ is comparable to that of the noise-free signal, $\|\mathbf{w}^*\| = \sqrt{2} \approx 1.4$. While 300 examples are technically enough to identify all 200 weights, we will soon see that many linear combinations of them are pinned down only weakly.

```
tau, sigma, n = 0.1, 1.0, 300   # size of the weights, noise, examples
data = d2l.SyntheticRegressionData(w=d2l.ones(200) * tau, b=0.05,
                                   noise=sigma, num_train=n, num_val=1000,
                                   batch_size=20)
```

Before training anything, let us set two benchmarks for the validation loss. For independent standard normal inputs $\mathbf{x} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}_d)$ and noise $\epsilon \sim \mathcal{N}(0, \sigma^2)$, the expected loss at $(\mathbf{w}, b^*)$ equals $\frac{1}{2}\|\mathbf{w} - \mathbf{w}^*\|_2^2 + \frac{1}{2}\sigma^2$ (Section 2.3). A trivial constant model that ignores all 200 inputs and predicts b^* everywhere therefore incurs an expected loss of $\frac{1}{2}(\|\mathbf{w}^*\|^2 + \sigma^2) = \frac{1}{2}(2+1) = 1.5$, while the oracle parameters $(\mathbf{w}^*, b^*)$ achieve $\sigma^2/2 = 0.5$. Any learned model must land comfortably below 1.5 to justify its 200 weights.

2.7.2.1 The Penalized Model

We define `l2_penalty(w)` to return $\frac{1}{2}\|\mathbf{w}\|_2^2$:

```
def l2_penalty(w):
    return d2l.reduce_sum(w**2) / 2
```

We subclass `LinearRegressionScratch` from Section 2.4 (our object-oriented version of the scratch model in Section 2.3). Only `training_step` needs to change so that it returns the penalized objective Equation 2.29 for differentiation and optimization. Crucially, `validation_step` continues to report the unpenalized squared loss, ensuring that validation losses across different values of λ remain directly comparable measures of predictive accuracy.

```
class WeightDecayScratch(d2l.LinearRegressionScratch):
    def __init__(self, num_inputs, lambd, lr, sigma=0.01):
        super().__init__(num_inputs, lr, sigma)
        self.save_hyperparameters()

    def training_step(self, batch):
        loss = super().training_step(batch)
        return loss + self.lambd * l2_penalty(self.w)
```

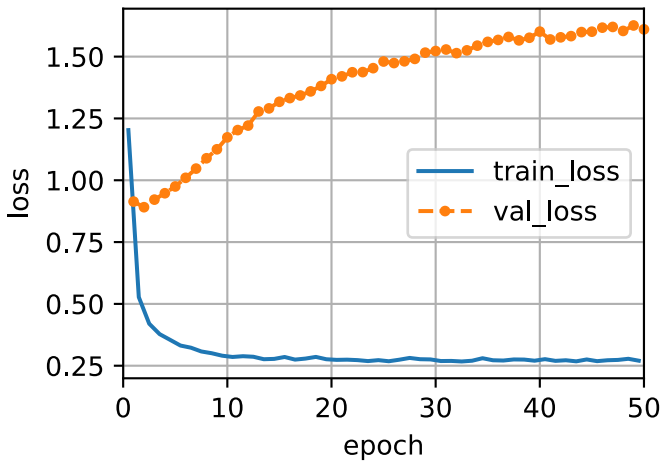
2.7.2.2 Training with and without the Penalty

Let us write `train_scratch(lambd, data)` to train `WeightDecayScratch` for 50 epochs at learning rate $\eta = 0.05$ (750 SGD steps) while plotting the training and validation curves, and to return the fitted model, the per-epoch validation losses, and the learned weight norm $\|\mathbf{w}\|_2$ (for comparison, the true weight vector $\mathbf{w}^* = (0.1, \ldots, 0.1)^\top \in \mathbb{R}^{200}$ has $\|\mathbf{w}^*\|_2 = \sqrt{200 \times 0.1^2} \approx 1.4$):

```
def train_scratch(lambd, data):
    trainer = d2l.Trainer(max_epochs=50)
    model = WeightDecayScratch(num_inputs=200, lambd=lambd, lr=0.05)
    trainer.fit(model, data)
    norm = float(torch.linalg.vector_norm(model.w.detach()))
    return model, [float(v) for v in trainer.history['val_loss']], norm
```

We begin with `lambd = 0`, disabling regularization altogether.

```
model_0, val_losses, norm = train_scratch(0, data)
```



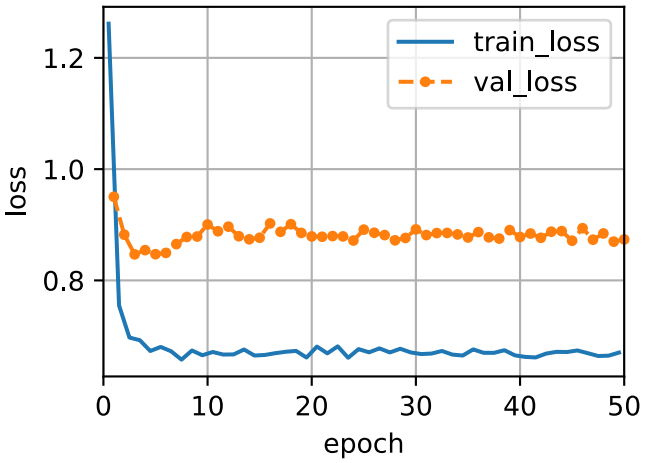

```
print(f'validation loss {val_losses[-1]:.3f} (lowest {min(val_losses):.3f}, '
      f'epoch {val_losses.index(min(val_losses)) + 1}), norm of w {norm:.2f}')
```

```
validation loss 1.610 (lowest 0.891, epoch 2), norm of w 2.03
```

The training loss plunges to roughly 0.2–0.3, far below the irreducible noise floor of 0.5 achieved by the true parameters, confirming that the model is memorizing noise in its 300 training labels. Meanwhile, the validation loss hits a minimum of 0.8–1.0 after just two or three epochs and then climbs steadily, typically finishing between 1.1 and 1.9. After 50 epochs, this 200-weight model has given back much of its early gain and often does no better than (and sometimes worse than) the trivial constant b^* at 1.5: a textbook display of overfitting. Why does validation loss bottom out so early, after only 30 to 45 gradient steps? We will uncover the reason once we examine which directions in weight space gradient descent learns first.

Next, we turn on weight decay. Rather than guessing λ blindly, we consult the Gaussian prior formula: for $\sigma = 1$, $n = 300$, and true weight scale $\tau = 0.1$, MAP estimation prescribes $\lambda = \sigma^2/(n\tau^2) = \frac{1}{3}$.

```
lambd = sigma**2 / (n * tau**2)   # the value the prior suggests, 1/3
model_lambd, val_losses, norm = train_scratch(lambd, data)
```




```
print(f'validation loss {val_losses[-1]:.3f}, norm of w {norm:.2f}')
```

```
validation loss 0.874, norm of w 1.17
```

Now the training curve tracks the penalized objective, which stays safely above 0.5: the penalty keeps the model from chasing noise in the training labels. Validation loss settles at roughly 0.75–1.05 and stays flat, well below the 1.5 baseline of the constant predictor, while the weight norm shrinks from roughly 2 to about 1.2, actually *shorter* than the true norm $\|\mathbf{w}^*\| \approx 1.4$. Even though the regularized weights are shrunk below the true length, they generalize dramatically better than the unregularized weights.

Let us be honest about what we gained here. Compared with the unpenalized model trained for all 50 epochs, weight decay slashes the validation loss by a quarter to a half. Compared with that same unpenalized model stopped at its best epoch, however, the gain is small: both reach roughly 0.8–1.0. The practical difference is stability. Because weight decay changes the objective itself, its solution stays good no matter how long we train, whereas the unpenalized run must be halted at a sweet spot that the training loss alone gives no hint

of. To understand *why* both regularization and early stopping behave this way, let us look inside the weight vector and ask which components the penalty actually removed.

2.7.3 Shrinkage Along Eigenvectors

Why does adding $\frac{\lambda}{2}\|\mathbf{w}\|_2^2$ improve validation error so dramatically instead of just scaling every weight down blindly? We can answer that question in closed form by looking at the exact minimizer of the ridge objective in the eigenbasis of the feature covariance matrix.

2.7.3.1 The Ridge Estimator

Suppose the features and labels are centered so $b = 0$, and write the ridge objective Equation 2.29 in matrix form:

$$L_\lambda(\mathbf{w}) = \frac{1}{2n}\|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 + \frac{\lambda}{2}\|\mathbf{w}\|_2^2. \tag{2.33}$$

Setting the gradient $\nabla_{\mathbf{w}} L_\lambda(\mathbf{w}) = \frac{1}{n}\mathbf{X}^\top(\mathbf{X}\mathbf{w} - \mathbf{y}) + \lambda\mathbf{w} = \mathbf{0}$ gives the regularized normal equations:

$$\begin{aligned}\left(\frac{1}{n}\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I}\right)\hat{\mathbf{w}}_\lambda &= \frac{1}{n}\mathbf{X}^\top\mathbf{y}, \\ \hat{\mathbf{w}}_\lambda &= \left(\mathbf{X}^\top\mathbf{X} + n\lambda\mathbf{I}\right)^{-1}\mathbf{X}^\top\mathbf{y}.\end{aligned} \tag{2.34}$$

Notice the immediate algebraic consequence: even when $\mathbf{X}^\top\mathbf{X}$ is singular (as it is whenever there are fewer examples than features: with $n = 20$ and $d = 200$, $\mathbf{X}^\top\mathbf{X}$ has 180 zero eigenvalues, exercise 6), adding $n\lambda\mathbf{I}$ for any $\lambda > 0$ shifts every eigenvalue upward by $n\lambda > 0$, making $\mathbf{X}^\top\mathbf{X} + n\lambda\mathbf{I}$ strictly positive definite and invertible.

2.7.3.2 Shrinkage Factors

To see how Equation 2.34 acts along each direction in feature space, let $\mathbf{S} = \frac{1}{n}\mathbf{X}^\top\mathbf{X}$ be the $d \times d$ sample second-moment (covariance) matrix of the features. Since $\mathbf{S}$ is symmetric and positive semidefinite (Section 1.3), it admits an orthonormal eigendecomposition $\mathbf{S} = \sum_{j=1}^{d} \mu_j \mathbf{v}_j \mathbf{v}_j^\top$ with eigenvalues $\mu_1 \geq \mu_2 \geq \cdots \geq \mu_d \geq 0$ and orthonormal eigenvectors $\mathbf{v}_1, \ldots, \mathbf{v}_d$. Here $\mu_j = \frac{1}{n}\|\mathbf{X}\mathbf{v}_j\|_2^2$ measures the empirical variance (spread) of the training inputs along direction $\mathbf{v}_j$.

In this eigenbasis, $\mathbf{S}$ acts on $\mathbf{v}_j$ by multiplying it by μ_j, while $(\mathbf{S} + \lambda\mathbf{I})^{-1}$ acts on $\mathbf{v}_j$ by dividing by $\mu_j + \lambda$. Whenever $\mu_j > 0$, comparing the ridge component $\mathbf{v}_j^\top\hat{\mathbf{w}}_\lambda$ against the unregularized least-squares component $\mathbf{v}_j^\top\hat{\mathbf{w}}_0$ yields

$$\begin{aligned}\mathbf{v}_j^\top\hat{\mathbf{w}}_\lambda &= \frac{\mu_j}{\mu_j + \lambda}\left(\mathbf{v}_j^\top\hat{\mathbf{w}}_0\right), \\ \mathbf{v}_j^\top\hat{\mathbf{w}}_\lambda &= 0 \quad \text{whenever } \mu_j = 0.\end{aligned}$$

Each eigendirection $\mathbf{v}_j$ is multiplied by its own *shrinkage factor* (Figure 2.14):

$$\frac{\mu_j}{\mu_j + \lambda} \in [0, 1). \tag{2.35}$$

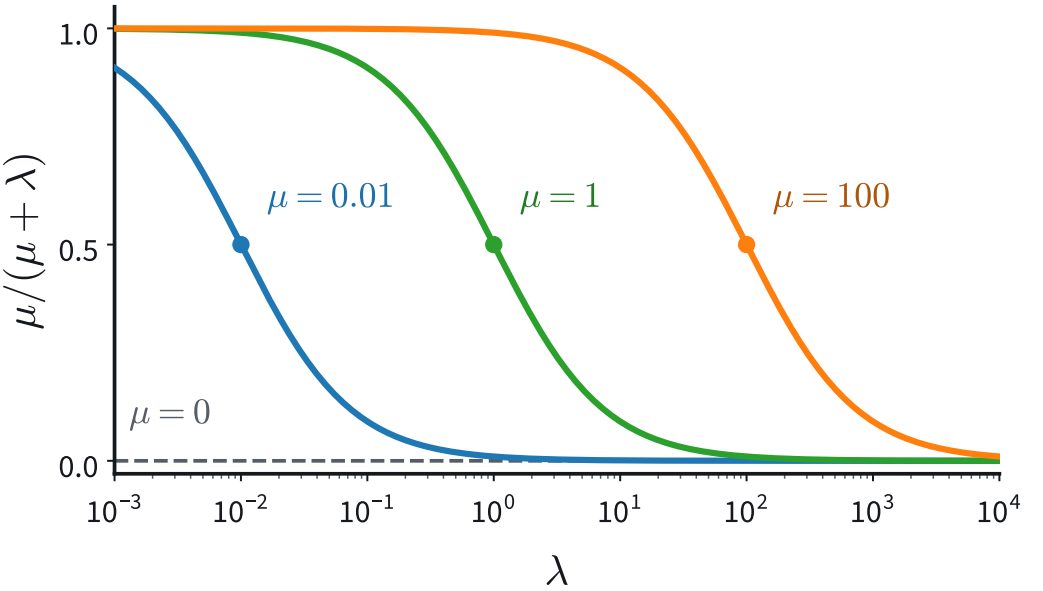


Figure 2.14: The ridge shrinkage factor $\mu/(\mu + \lambda)$ along an eigenvector with eigenvalue μ, against the regularization constant λ (log scale). A direction well determined by the data is kept nearly untouched while $\lambda \ll \mu$ and is strongly shrunk toward zero once $\lambda \gg \mu$; the factor is $\frac{1}{2}$ at $\lambda = \mu$ (dots). Directions with $\mu = 0$ are set to zero for every $\lambda > 0$ (dashed).

Here lies the secret of ridge regression: **weight decay is not uniform shrinkage; it is variance-adaptive directional filtering.**

- Along **well-determined directions** where the data vary richly ($\mu_j \gg \lambda$), the slope is tightly pinned down by observations, and the shrinkage factor $\mu_j/(\mu_j+\lambda) \approx 1$ leaves the least-squares estimate nearly untouched.
- Along **poorly determined directions** where the data barely vary ($\mu_j \ll \lambda$), such as the difference $x_1 - x_2$ between our two almost-identical thermometers, unregularized least squares divides noise by a tiny μ_j and blows up (with coefficient variance $\sigma^2/(n\mu_j)$), whereas ridge regression multiplies by $\mu_j/(\mu_j + \lambda) \approx \mu_j/\lambda \ll 1$ and damps the noisy component.
- Along **unobserved null-space directions** ($\mu_j = 0$), ridge regression sets the weight component identically to 0.

2.7.3.3 The Directions Our Data Determine

Let us verify this spectral picture on our $d = 200$ dataset `data`. First, we compute the eigenvalues μ_j of $\mathbf{S} = \frac{1}{n}\mathbf{X}^\top\mathbf{X}$ for the 300 training inputs (uncentered, which scarcely alters the spectrum) along with the shrinkage factors at $\lambda = \frac{1}{3}$. As in Section 1.3, `eigh` returns the eigenvalues in ascending order with the eigenvectors as the columns of its second output.

We also count how many eigenvalues fall below λ, marking the poorly determined directions where ridge removes more than half of the least-squares coordinate:

```
X = data.X[:n]                                # the 300 training inputs
mu, V = torch.linalg.eigh(X.T @ X / n)        # increasing eigenvalues
factors = mu / (mu + lambd)                   # shrinkage at lambda = 1/3
k = int((mu < lambd).sum())                   # the poorly determined ones
print(f'mu from {float(mu[0]):.2f} to {float(mu[-1]):.1f}, '
      f'{k} of {len(mu)} below lambda')
print(f'factors from {float(factors[0]):.2f} to {float(factors[-1]):.2f}, '
      f'df = {float(factors.sum()):.0f}')
```

```
mu from 0.04 to 3.2, 54 of 200 below lambda
factors from 0.10 to 0.91, df = 127
```

None of the eigenvalues is zero: 300 examples are enough to span all 200 dimensions of weight space. Yet the data constrain some directions far more tightly than others. The eigenvalues span nearly two orders of magnitude, from a few hundredths up to about 3.2, with roughly a quarter of them (50 to 60 directions) lying below $\lambda = \frac{1}{3}$. The corresponding shrinkage factors range from about 0.1 to 0.9. Their sum defines the *effective number of parameters* $\mathrm{df}(\lambda) = \sum_j \mu_j/(\mu_j + \lambda)$, and here $\mathrm{df}(\frac{1}{3}) \approx 127$: out of 200 raw weights, the regularized model behaves as though it has roughly 127 effective parameters.

Just how much damage does a tiny eigenvalue inflict on least squares? Along eigenvector $\mathbf{v}_j$, label noise injects $\frac{1}{n}(\mathbf{X}\mathbf{v}_j)^\top \epsilon$ into $\frac{1}{n}\mathbf{v}_j^\top \mathbf{X}^\top \mathbf{y}$, a random perturbation with standard deviation $\sigma\sqrt{\mu_j/n}$ (because $\|\mathbf{X}\mathbf{v}_j\|^2 = n\mu_j$). Least squares divides by μ_j, which shifts its coordinate $\mathbf{v}_j^\top \hat{\mathbf{w}}_0$ by roughly $\sigma/\sqrt{n\mu_j}$: about 0.06 when $\mu_j = 1$, but a whopping 0.3 when $\mu_j = 0.04$. Since the true coordinates are only around $\|\mathbf{w}^*\|/\sqrt{200} = 0.1$ in magnitude, least squares along the smallest-eigenvalue directions is swamped by noise. Meanwhile, validation inputs are isotropic standard normal, so orthogonal invariance of $\|\mathbf{w} - \mathbf{w}^*\|_2^2 = \sum_j c_j^2$ means that an error c_j along $\mathbf{v}_j$ inflates the expected validation loss by $c_j^2/2$, regardless of μ_j. Let us verify this directly by projecting the parameter error $\mathbf{w} - \mathbf{w}^*$ of both trained models onto the eigenvectors (`V.T @ (w - w_star)`) and comparing the total squared error accumulated in the k poorly determined directions versus the remaining ones:

```
w_star = d2l.reshape(data.w, (-1, 1))
for name, model in (('lambda = 0', model_0), ('lambda = 1/3', model_lambd)):
    c = V.T @ (model.w.detach() - w_star)     # error along each eigenvector
    print(f'{name:>12}: squared error {float((c[:k] ** 2).sum()):.2f} in the '
          f'{k} poorly determined directions, '
          f'{float((c[k:] ** 2).sum()):.2f} in the rest')
```

```
  lambda = 0: squared error 1.49 in the 54 poorly determined directions, 0.52 in the rest
lambda = 1/3: squared error 0.30 in the 54 poorly determined directions, 0.40 in the rest
```

The breakdown tells the whole story. Without regularization, roughly one quarter of the eigenvectors account for more than half of the total squared parameter error: typically between 0.8 and 2.2, versus only about 0.5 across all the remaining directions combined. Turning on weight decay slashes that poorly determined error by a factor of two to six while trimming the rest modestly, leaving roughly 0.3 to 0.5 in each group. Even along the better-

determined directions, label noise is nontrivial relative to the true coordinates, so shrinking them by 0.5–0.9 still discards more noise than signal. As a quick sanity check, half the total squared parameter error plus $\sigma^2/2 = 0.5$ roughly matches the validation loss we observed for both runs. In effect, weight decay performed across some fifty ill-conditioned directions simultaneously the same selective damping it performs on the difference direction $x_1 - x_2$ of our two thermometers.

This spectral lens also solves the mystery we left hanging earlier: why did the unpenalized run hit its lowest validation loss after just two or three epochs? On the unpenalized loss, each gradient step contracts the gap between the current coordinate along $\mathbf{v}_j$ and its least-squares solution by a factor of $1 - \eta\mu_j$. After a single epoch of 15 steps at $\eta = 0.05$, a large-eigenvalue direction ($\mu_j = 3.2$) has already closed $1 - 0.84^{15} \approx 93\%$ of its gap, whereas a small-eigenvalue direction ($\mu_j = 0.04$) has closed a mere $1 - 0.998^{15} \approx 3\%$. After two epochs, gradient descent has essentially converged along the well-determined directions while barely moving along the noisy ones—closely mimicking the ridge estimator. The resemblance can be made quantitative. Starting from weights near zero, t steps build up the fraction $1 - (1 - \eta\mu_j)^t$ of the least-squares coordinate along $\mathbf{v}_j$, and this is close to the ridge factor $\mu_j/(\mu_j + \lambda)$ with $\lambda \approx 1/(\eta t)$: both are nearly 1 when $\mu_j \gg 1/(\eta t)$ and nearly $\eta t \mu_j$ when $\mu_j \ll 1/(\eta t)$. Our best stopping point of 30 to 45 steps at $\eta = 0.05$ gives $1/(\eta t)$ between 0.45 and 0.67, the same order as the $\lambda = \frac{1}{3}$ that the prior prescribed. Keep training, however, and those poorly determined directions steadily soak up label noise, driving validation loss back up. *Early stopping* is therefore a genuine form of regularization in its own right, and in this experiment it matches weight decay in accuracy. It is trickier to deploy, however, because the optimal stopping point shifts with the learning rate and minibatch noise and must be monitored on validation data throughout every run (exercise 5).

2.7.4 Concise Implementation

Because weight decay is ubiquitous in neural network training, every major deep learning framework lets the optimizer apply the update Equation 2.30, so you need not modify the loss function. Beyond convenience, this integration is computationally clean: the optimizer already visits every parameter once per step, so applying the decay factor adds virtually no overhead. Frameworks also support per-parameter update rules, allowing us to decay weights while leaving biases untouched.

We simply subclass `d2l.LinearRegression` into `WeightDecay` and pass the weight decay hyperparameter `wd` to the framework's SGD optimizer in `configure_optimizers` (excluding the bias parameter from decay so that only the weights $\mathbf{w}$ are penalized, matching Equation 2.29).

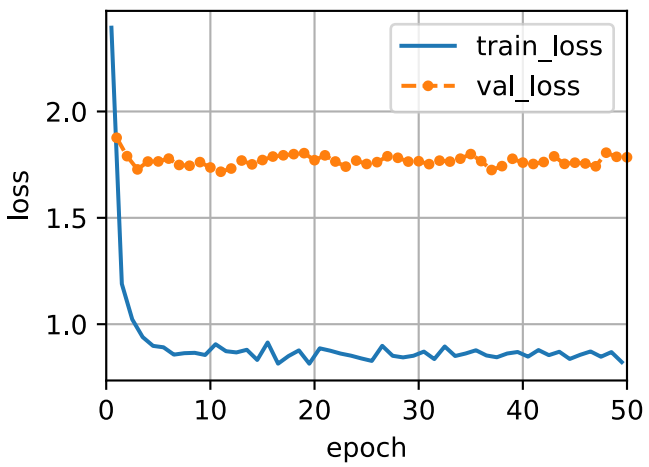
PyTorch optimizers accept a `weight_decay` argument. By splitting the parameters into two parameter groups, we apply `weight_decay` only to `net.weight` while leaving `net.bias` unregularized. Just as in Section 2.5, we initialize the weights from $\mathcal{N}(0, 0.01^2)$ and the bias to zero by applying `weight_init` after the first forward pass materializes the lazy layer's parameters.

```
class WeightDecay(d2l.LinearRegression):
    def __init__(self, wd, lr):
        super().__init__(lr)
        self.save_hyperparameters()
```

```
    def configure_optimizers(self):
        return torch.optim.SGD([
            {'params': self.net.weight, 'weight_decay': self.wd},
            {'params': self.net.bias}], lr=self.lr)
```

To match `train_scratch(lambd, data)` exactly, we must rescale both hyperparameters. Because the built-in squared-error loss omits our factor of $\frac{1}{2}$, we halve the learning rate to 0.025 just as in Section 2.5. However, the decay term $\lambda \mathbf{w}$ is injected inside the optimizer rather than multiplied by the loss gradient. With the learning rate cut in half, the optimizer's weight decay must therefore be doubled to $2\lambda = \frac{2}{3}$ so that every step retains the exact same contraction factor $1 - 0.025 \cdot \frac{2}{3} = 1 - \frac{1}{60}$.

```
trainer = d2l.Trainer(max_epochs=50)
model = WeightDecay(wd=2 * lambd, lr=0.025)
model(data.X[:1])  # Creates the parameters of the lazy layer
model.net.apply(model.weight_init)
trainer.fit(model, data)
norm = float(torch.linalg.vector_norm(model.get_w_b()[0]))
```




```
print(f'validation loss {trainer.history["val_loss"][-1]:.3f}, '
      f'norm of w {norm:.2f}')
```

```
validation loss 1.785, norm of w 1.17
```

The resulting weight norm (around 1.2) reproduces our from-scratch run at $\lambda = \frac{1}{3}$, and the validation loss (about twice the from-scratch value) agrees too once we account for the factor of 2 in the framework's mean squared error. Note that the plotted training curve now tracks plain mean squared error rather than the penalized loss, because the decay lives inside the optimizer rather than the loss function.

In everyday practice, enabling weight decay is as simple as passing this single optimizer hyperparameter. Outside synthetic benchmarks we rarely know the noise level σ or prior scale τ in advance, so λ is typically tuned on validation data across candidate runs. Keep in

mind that our norm-based story applies directly to linear models, whose weights are the slopes of the function itself. For nonlinear functions, measuring complexity rigorously is subtler: frameworks such as reproducing kernel Hilbert spaces (RKHS) extend norm penalties to nonlinear functions, though classical kernel methods scale poorly to massive datasets. Even so, applying weight decay to the weights of neural network layers remains one of the most reliable regularizers in modern deep learning.

2.7.5 Summary

Adding a squared ℓ_2 penalty $\frac{\lambda}{2}\|\mathbf{w}\|_2^2$ to the training objective (while leaving the bias b unpenalized) keeps high-dimensional models from amplifying noise along nearly collinear feature combinations. In minibatch SGD, this penalty multiplies $\mathbf{w}$ by $(1-\eta\lambda)$ before each gradient step—hence the name *weight decay*—and corresponds to maximum a posteriori estimation under an isotropic Gaussian prior $w_j \sim \mathcal{N}(0,\tau^2)$ with $\lambda = \sigma^2/(n\tau^2)$. Whereas the sharp corners of an ℓ_1 penalty drive weights to exact zero, the smooth ℓ_2 penalty acts as a directional filter in the eigenbasis of $\frac{1}{n}\mathbf{X}^\top\mathbf{X}$: multiplying each least-squares component by $\mu_j/(\mu_j+\lambda)$ preserves directions well determined by the data ($\mu_j \gg \lambda$), strongly damps noisy directions ($\mu_j \ll \lambda$), and zeroes unobserved null-space directions ($\mu_j = 0$).

2.7.6 Exercises

1. **The ℓ_1 update.** Replace the penalty $\frac{\lambda}{2}\|\mathbf{w}\|^2$ by $\lambda\sum_j |w_j|$.

    1. Derive the minibatch SGD update for $\mathbf{w}$, using the subgradient $\operatorname{sign}(w_j)$ of exercise 2 in place of the derivative of $|w_j|$.
    2. Compare the change that the penalty alone makes to w_j in one step of this update with the change it makes in the update Equation 2.30, once for large and once for small $|w_j|$.
    3. The update of (a) almost never sets w_j exactly to zero. Split it into a gradient step on L followed by the penalty step, and modify the penalty step so that it sets w_j to zero whenever it would change the sign of w_j. Relate the result to the feature selection that lasso performs. The modified step is *soft-thresholding*, the proximal step of the ℓ_1 norm (Section 35.6.4.1).

2. **Shrinkage versus penalty.** Weight decay can be applied by shrinking the weights before the gradient step or by adding $\frac{\lambda}{2}\|\mathbf{w}\|^2$ to the loss L.

    1. Show that the update $\mathbf{w} \leftarrow (1-\eta\lambda)\mathbf{w} - \eta\nabla L$ is identical to a gradient step on the penalized loss $L + \frac{\lambda}{2}\|\mathbf{w}\|^2$.
    2. Suppose that the gradient is rescaled coordinate-wise by fixed constants $a_j > 0$ before the step. Shrinking and then stepping gives the first update below, and stepping on the penalized gradient gives the second:

$$w_j \leftarrow (1-\eta\lambda)\, w_j - \eta a_j\, \partial_{w_j} L \qquad \text{and} \qquad w_j \leftarrow w_j - \eta a_j \left(\partial_{w_j} L + \lambda w_j\right). \tag{2.36}$$

    Determine for which a_j the two coincide, and the factor by which the second update shrinks w_j in a step with $\partial_{w_j} L = 0$. Many optimizers used in deep learning

rescale the gradient in this way. Which of the two updates keeps λ a rate of decay that is the same for every weight?

Adapted from Simon Prince, Understanding Deep Learning, Problem 9.5.

3. code **Ridge as augmented least squares.** Center the 300 training inputs and labels so that the bias drops out, and compute the ridge estimator for $\lambda = \frac{1}{3}$ in two ways.
   1. Solve the linear system Equation 2.34 directly, for instance with `np.linalg.solve`.
   2. Run ordinary least squares with `lstsq` on $\mathbf{X}$ stacked on top of $\sqrt{n\lambda}\,\mathbf{I}$, with the labels $\mathbf{y}$ followed by d zeros.

   Confirm that the two agree up to rounding errors, and explain why. Then compare the norm of the result with the norm that `train_scratch(lambd, data)` returned, and name the differences between the two procedures that could account for any gap.

   Adapted from Hastie, Tibshirani, and Friedman, The Elements of Statistical Learning, Exercise 3.12.

4. code **The λ sweep.** Train `WeightDecayScratch` on `data` for $\lambda \in \{0.01, 0.1, 0.3, 1, 3, 10, 30\}$.
   1. Plot the final validation loss and the norm of $\mathbf{w}$ against λ on a logarithmic axis, with the values of `train_scratch(0, data)` as horizontal lines, and report the λ with the smallest validation loss.
   2. Compute $\mathrm{df}(\lambda)$ for the same values from the eigenvalues of Section 2.7.3, and plot the validation loss against $\mathrm{df}(\lambda)$. Compare the shape with Figure 2.11.
   3. Is the value $\lambda = \frac{1}{3}$ that the prior suggests close to the best value of the sweep? Generate data with noise $\sigma = 0.5$ and with $\sigma = 2$, recompute the prior's λ for each, and check whether it follows the best value of a sweep.

5. code **Stopping early versus decay.** The unpenalized run of this section reaches its lowest validation loss after two or three epochs, about as low as the run with $\lambda = \frac{1}{3}$ ends.
   1. From weights near zero, gradient descent on the unpenalized loss multiplies the gap between the coordinate along $\mathbf{v}_j$ and its least-squares value by $1 - \eta\mu_j$ per step, so after t steps it has built up the fraction $1 - (1 - \eta\mu_j)^t$ of that coordinate. Plot this factor against μ_j for the eigenvalues of Section 2.7.3, with t equal to one, two, and fifty epochs of 15 steps at $\eta = 0.05$, next to the ridge factor $\mu_j/(\mu_j + \lambda)$ at $\lambda = \frac{1}{3}$. Which t comes closest to ridge?
   2. Train `WeightDecayScratch` with $\lambda = 0$ for two epochs and split its error along the eigenvectors as Section 2.7.3 does. Compare the squared error in the k poorly determined directions and in the rest with the values printed there for 50 epochs at $\lambda = 0$ and at $\lambda = \frac{1}{3}$.
   3. Repeat the unpenalized run at $\eta = 0.01$ and at $\eta = 0.1$ and read the best epoch from `trainer.history`. Explain how it moves, and state what early stopping requires in practice that the penalty does not.

6. code **More weights than examples.** ● Generate data with 200 features and only 20 training examples, true weights 0.01, bias 0.05, and noise $\sigma = 0.01$, in minibatches of 5.
   1. Compute the eigenvalues of $\frac{1}{n}\mathbf{X}^\top\mathbf{X}$ for the 20 training inputs. How many are zero,

and why does gradient descent never change the weights along their eigenvectors?

2. Train with $\lambda = 0$ and with $\lambda = 3$ for 30 epochs at learning rate 0.01, once from the random initialization of `WeightDecayScratch` and once from $\mathbf{w} = \mathbf{0}$ (pass `sigma=0`). Before you run, predict the four validation losses from the norm of the initial weights in the undetermined directions; then compare.
3. Compare all four losses with that of a model that predicts b^* everywhere. What did weight decay achieve here, and what could no value of λ achieve?

Chapter 3

Linear Classification

In Chapter 2, our goal was to predict a continuous quantity such as the sale price of a house. Many tasks ask for a discrete category instead: an online catalog photo depicts a shirt, a coat, or a sandal, never "5.5" of anything. Simply numbering the categories 1 through 10 and running linear regression fails badly, because it imposes an arbitrary ordering as though a pullover lay halfway between a pair of trousers and a dress, while leaving the model unable to express calibrated uncertainty ("most likely a shirt, possibly a pullover").

Happily, almost all of our regression training machinery—minibatch data loaders, vectorized forward passes, automatic differentiation, and minibatch SGD—carries straight over to *classification*. Only three ingredients change: we represent unordered categories via *one-hot vectors*, compute one linear score (*logit*) per category and normalize them via the *softmax* function into a probability distribution, and replace squared error with the *cross-entropy loss* (derived from maximum likelihood under a categorical label model, just as squared error arose from Gaussian noise). Nearly every neural classifier in this book, right up to large language models, ends in this softmax layer and trains under this loss.

We first formulate softmax regression and the cross-entropy loss, derive its residual gradient $\hat{\mathbf{y}} - \mathbf{y}$, and connect it to Shannon's information theory (Section 3.1). We then build a reusable `FashionMNIST` image pipeline (Section 3.2) and implement softmax regression both from scratch and concisely (Section 3.3), diagnosing errors with confusion matrices and learned 28×28 templates while curing floating-point overflow and underflow with a fused log-sum-exp loss.

Where classification departs sharply from regression is in *evaluation* and *deployment*, which occupy the second half of the chapter. Raw accuracy misleads whenever positive examples are rare or mistakes carry unequal costs, motivating precision, recall, cost-sensitive thresholds, and calibration diagnostics (Section 3.4). Even on balanced classes, a finite test set carries statistical error bars that erode under repeated reuse, leading us to Hoeffding bounds, adaptive overfitting, and Vapnik–Chervonenkis (VC) theory (Chapter 3.5). Finally, Section 3.6 asks what happens when deployment data drift away from the training distribution, developing importance-weighted covariate-shift correction, confusion-matrix label-shift estimation, and defenses against runaway feedback loops.

3.1 Softmax Regression

Regression is the tool we reach for when answering *how much?* or *how many?* questions. Many supervised learning problems, however, ask *which category?* instead: a spam filter sorts email into two classes, a clinical tool chooses among candidate diagnoses, and a language model predicts the next token from a vocabulary of tens of thousands of words.

Suppose an online retailer wants to sort product photos into ten apparel categories: T-shirt, trouser, pullover, dress, coat, sandal, shirt, sneaker, bag, and ankle boot. Why not number those categories 1 through 10 and apply linear regression directly? Because integer codes for unordered categories are arbitrary names, not numerical quantities: a prediction of 5.5 would place a photo halfway between a coat and a sandal, and squared error would penalize mistaking a T-shirt (1) for an ankle boot (10) 81 times more heavily than mistaking it for a trouser (2). Moreover, even when we ultimately need a *hard assignment* (picking a single category), we almost always want a model that outputs *soft assignments* (a calibrated probability for every candidate class) so downstream decisions can weigh confidence against the cost of a mistake.

Softmax regression (also called *multinomial logistic regression*) solves both problems cleanly: it computes one affine score per class and maps those scores into a valid probability distribution via the softmax operator. Following the maximum-likelihood recipe of Section 2.1, minimizing the negative log-likelihood of the observed categories yields the *cross-entropy loss*.

3.1.1 The Model

3.1.1.1 Class Labels

Let us start with a concrete 2×2 grayscale image classification problem with four scalar pixel features x_1, x_2, x_3, x_4 and three categories: cat (1), chicken (2), and dog (3). In general, with q classes, an example's label is an integer $y \in \{1, \dots, q\}$.

To represent unordered categories without imposing a false numerical ordering, statisticians use a *one-hot encoding*: a vector $\mathbf{y} \in \{0, 1\}^q$ with a 1 in the coordinate of the true class ($y_y = 1$) and 0 everywhere else. In our three-class example, a cat is $(1, 0, 0)$, a chicken is $(0, 1, 0)$, and a dog is $(0, 0, 1)$. Every pair of distinct classes now sits at the same Euclidean distance $\sqrt{2}$, and $\mathbf{y}$ is itself a valid probability distribution—the degenerate distribution placing all of its mass on the true class—which we can compare directly against our model's predicted distribution $\hat{\mathbf{y}}$. (In equations we use 1-based class index y and one-hot vector $\mathbf{y}$; in Python code we store the 0-based integer index directly.)

3.1.1.2 Linear Scores

To score all q classes simultaneously, softmax regression assigns each class $j \in \{1, \dots, q\}$ its own affine function of the input features. Writing w_{kj} for the weight connecting feature k to class j and b_j for class j's bias, our 2×2 image classifier computes three unnormalized scores (*logits*):

$$
\begin{aligned}
o_1 &= x_1 w_{11} + x_2 w_{21} + x_3 w_{31} + x_4 w_{41} + b_1, \\
o_2 &= x_1 w_{12} + x_2 w_{22} + x_3 w_{32} + x_4 w_{42} + b_2, \\
o_3 &= x_1 w_{13} + x_2 w_{23} + x_3 w_{33} + x_4 w_{43} + b_3.
\end{aligned}
$$

Viewed as a neural network (Figure 3.1), softmax regression is a single-layer fully connected network with d inputs and q output units ($d \cdot q$ weights and q biases).

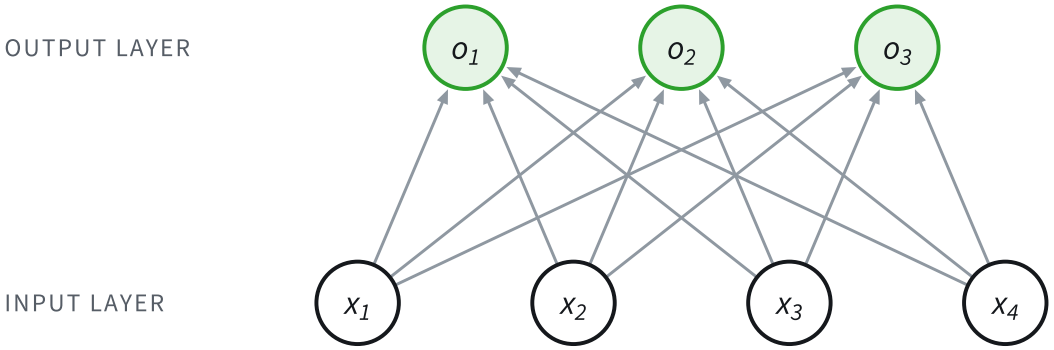


Figure 3.1: Softmax regression as a single-layer fully connected neural network. Each input x_k connects to each output logit o_j through weight w_{kj}.

Collecting the weights into a matrix $\mathbf{W} \in \mathbb{R}^{d \times q}$ (one row per feature, one column $\mathbf{w}_j \in \mathbb{R}^d$ per class) and the biases into $\mathbf{b} \in \mathbb{R}^q$, the logit vector for an input $\mathbf{x} \in \mathbb{R}^d$ is compactly written

$$\mathbf{o} = \mathbf{W}^\top \mathbf{x} + \mathbf{b}, \tag{3.1}$$

where $o_j = \mathbf{w}_j^\top \mathbf{x} + b_j$. On the 28×28 Fashion-MNIST clothing images we classify in Section 3.2, $d = 784$ and $q = 10$, giving $7{,}840$ weights and 10 biases.

3.1.1.3 The Softmax

Suppose our linear layer outputs the logits $\mathbf{o} = (1.0, 2.2, 0.3)$ for cat, chicken, and dog. The chicken has the highest score, yet raw logits are not probabilities: they sum to 3.5 rather than 1, and nothing stops an affine score from turning negative or exceeding 1. To convert any real vector $\mathbf{o} \in \mathbb{R}^q$ into a valid probability distribution $\hat{\mathbf{y}}$ (q positive numbers summing to 1) while preserving the ranking of the scores, the *softmax* function exponentiates each logit and divides by their sum:

$$\hat{\mathbf{y}} = \mathrm{softmax}(\mathbf{o}) \quad \text{where} \quad \hat{y}_j = \frac{\exp(o_j)}{\sum_{k=1}^{q} \exp(o_k)}. \tag{3.2}$$

For $\mathbf{o} = (1.0, 2.2, 0.3)$, the exponentials are $(2.72, 9.03, 1.35)$ with sum 13.09, giving $\hat{\mathbf{y}} = (0.21, 0.69, 0.10)$: a 69% predicted probability for chicken. Because exp is strictly increasing, softmax preserves the order of the logits, so the most probable class is always the class with the largest logit:

$$\operatorname*{argmax}_j \hat{y}_j = \operatorname*{argmax}_j o_j. \tag{3.3}$$

At inference time, if we only need the top predicted category, we can skip the exponentials altogether and take $\operatorname{argmax}_j o_j$.

Why use the exponential rather than some other positive increasing function? Because exponentials turn addition into multiplication, endowing the softmax with three foundational properties:

1. **Shift invariance (only differences between logits matter):** Adding any scalar constant c to every entry of $\mathbf{o}$ multiplies both numerator and denominator of Equation 3.2 by $\exp(c)$, leaving the probabilities unchanged: $\text{softmax}(\mathbf{o} + c\mathbf{1}) = \text{softmax}(\mathbf{o})$. Equivalently, the difference between any two logits is the log-odds ratio of their predicted probabilities:
$$o_i - o_j = \log \frac{\hat{y}_i}{\hat{y}_j}. \tag{3.4}$$
In our example, $o_2 - o_1 = 1.2$, so chicken is $e^{1.2} \approx 3.3$ times as probable as cat ($0.69/0.21$). This shift invariance also gives us a critical numerical safeguard (Section 3.3): subtracting $o_{\text{max}} = \max_k o_k$ before exponentiating prevents floating-point overflow without changing the result.
2. **Reduction to logistic regression for $q = 2$:** With two classes, the probabilities depend solely on the single difference $o = o_1 - o_2$:
$$\hat{y}_1 = \frac{\exp(o_1)}{\exp(o_1) + \exp(o_2)} = \frac{1}{1 + \exp(-o)} = \sigma(o), \tag{3.5}$$
where $\sigma(o)$ is the *logistic sigmoid*. Two-class softmax regression is thus identical to classic *logistic regression*, and $o = \log(\hat{y}_1/\hat{y}_2)$ is the log-odds—the historical origin of the word *logit*.
3. **Clean gradients under maximum likelihood:** As we show below, the exponential in the softmax pairs with the logarithm in the likelihood so that the gradient of the loss collapses to the residual $\hat{\mathbf{y}} - \mathbf{y}$.

3.1.1.4 Minibatches

To vectorize over a minibatch of n examples with d features, we stack the inputs into $\mathbf{X} \in \mathbb{R}^{n \times d}$, weights into $\mathbf{W} \in \mathbb{R}^{d \times q}$, and biases into $\mathbf{b} \in \mathbb{R}^{1 \times q}$:

$$\begin{aligned} \mathbf{O} &= \mathbf{X}\mathbf{W} + \mathbf{b}, \\ \hat{\mathbf{Y}} &= \text{softmax}(\mathbf{O}), \end{aligned} \tag{3.6}$$

where $\mathbf{b}$ broadcasts across the n rows of $\mathbf{O} \in \mathbb{R}^{n \times q}$ and softmax is applied *rowwise* (exponentiating $\mathbf{O}$ and dividing each row by its own row sum).

3.1.1.5 Decision Regions

Because softmax preserves the ordering of the affine logits $o_j = \mathbf{w}_j^\top \mathbf{x} + b_j$, softmax regression partitions the input space $\mathbb{R}^d$ into flat-sided convex polyhedra (Figure 3.2). Class j wins wherever $o_j \geq o_k$ for all $k \neq j$, which is the intersection of $q - 1$ halfspaces $(\mathbf{w}_j - \mathbf{w}_k)^\top \mathbf{x} + (b_j - b_k) \geq 0$, and neighboring classes meet along the hyperplane where their two logits tie. Just as in linear regression, boundaries between classes are always *flat* hyperplanes; carving curved or non-convex decision regions requires hidden layers (Chapter 4).

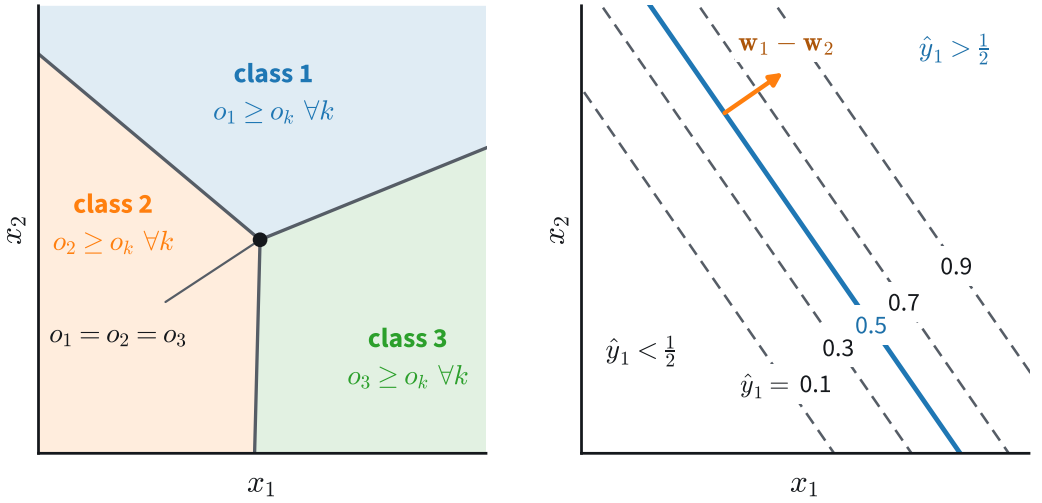


Figure 3.2: Decision regions of softmax regression in $\mathbb{R}^2$. Left: with three classes, the plane splits into three convex regions meeting along the lines where pairs of logits tie. Right: with two classes, contours of constant probability $\hat{y}_1 = \sigma(o)$ are parallel lines perpendicular to $\mathbf{w}_1 - \mathbf{w}_2$, with the decision boundary at $\hat{y}_1 = 1/2$.

Keep in mind our core structural assumption: each example belongs to *exactly one* class from an *unordered* set. When multiple tags can apply to the same input simultaneously (*multi-label classification*), one uses an independent binary sigmoid per tag rather than a competing softmax (Tsoumakas and Katakis 2007); when classes follow a natural rank order (*ordinal regression*), specialized threshold models exploit that ordering (Moon et al. 2010).

3.1.2 The Cross-Entropy Loss

3.1.2.1 Maximum Likelihood

How should we measure the discrepancy between our predicted probability distribution $\hat{\mathbf{y}}$ and the observed label y? Following the maximum-likelihood principle of Section 2.1, we interpret $\hat{y}_j = P(y = j \mid \mathbf{x})$ as the conditional probability of class j. Using the one-hot vector $\mathbf{y} \in \{0, 1\}^q$, we can write the probability assigned to the observed class y as

$$P(y \mid \mathbf{x}) = \hat{y}_y = \prod_{j=1}^{q} \hat{y}_j^{y_j}.$$

Across n independent training examples, the negative log-likelihood is the sum of per-example losses:

$$\begin{aligned} -\log \prod_{i=1}^{n} P(y^{(i)} \mid \mathbf{x}^{(i)}) &= \sum_{i=1}^{n} l(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}), \\ \text{where} \quad l(\mathbf{y}, \hat{\mathbf{y}}) &= -\sum_{j=1}^{q} y_j \log \hat{y}_j = -\log \hat{y}_y. \end{aligned} \tag{3.7}$$

This objective is the *cross-entropy loss*. Notice how it behaves:

- Because $\hat{y}_y \in (0, 1)$, $l(\mathbf{y}, \hat{\mathbf{y}}) > 0$ always. It approaches 0 as $\hat{y}_y \to 1$ and diverges to $+\infty$ as $\hat{y}_y \to 0$ (heavily penalizing confident mistakes).
- On our image with logits $(1.0, 2.2, 0.3)$ and probabilities $(0.21, 0.69, 0.10)$, if the true label is chicken ($y = 2$), the loss is $-\log 0.69 \approx 0.37$ nats; if the image is actually a dog ($y = 3$), the loss jumps sixfold to $-\log 0.10 \approx 2.30$ nats.

3.1.2.2 The Gradient

Let us differentiate the cross-entropy loss with respect to the logits $\mathbf{o}$. Substituting Equation 3.2 into Equation 3.7 and using $\sum_{j=1}^{q} y_j = 1$ expresses the loss directly in terms of $\mathbf{o}$:

$$\begin{aligned} l(\mathbf{y}, \hat{\mathbf{y}}) &= -\sum_{j=1}^{q} y_j \log \frac{\exp(o_j)}{\sum_{k=1}^{q} \exp(o_k)} \\ &= \sum_{j=1}^{q} y_j \log \sum_{k=1}^{q} \exp(o_k) - \sum_{j=1}^{q} y_j o_j \\ &= \log \sum_{k=1}^{q} \exp(o_k) - o_y. \end{aligned}$$

The first term, $\log \sum_{k=1}^{q} \exp(o_k)$, is the *log-sum-exp* function—a smooth convex upper bound on $\max_k o_k$ that exceeds $\max_k o_k$ by at most $\log q$ (Exercise 4). Differentiating with respect to o_j gives

$$\begin{aligned} \frac{\partial l}{\partial o_j} &= \frac{\exp(o_j)}{\sum_{k=1}^{q} \exp(o_k)} - y_j = \operatorname{softmax}(\mathbf{o})_j - y_j, \\ \text{or} \quad \nabla_{\mathbf{o}} l &= \hat{\mathbf{y}} - \mathbf{y}. \end{aligned}$$

Just as in linear regression under squared loss, **the gradient with respect to the natural parameters o is simply prediction minus observation, $\hat{\mathbf{y}} - \mathbf{y}$** (a universal property of exponential-family generalized linear models; see Section 36.4.3.3). On our chicken image ($\mathbf{y} = (0, 1, 0)$), $\nabla_{\mathbf{o}} l = (0.21, -0.31, 0.10)$: one gradient descent step boosts the chicken's logit while pushing down the cat and dog logits in exact proportion to the probability mass each rival wrongly claimed. By the chain rule through $\mathbf{O} = \mathbf{X}\mathbf{W} + \mathbf{b}$, the minibatch gradient with respect to $\mathbf{W}$ is $\frac{1}{n}\mathbf{X}^\top(\hat{\mathbf{Y}} - \mathbf{Y})$, and the loss is globally *convex* in $(\mathbf{W}, \mathbf{b})$ (its Hessian with respect to $\mathbf{o}$ is the covariance matrix of a categorical draw from $\hat{\mathbf{y}}$; see Exercise 2).

	Linear regression	Softmax regression
Target	continuous $y \in \mathbb{R}$	class $y \in \{1, \dots, q\}$, one-hot $\mathbf{y}$
Prediction	$\hat{y} = \mathbf{w}^\top \mathbf{x} + b$	$\hat{\mathbf{y}} = \operatorname{softmax}(\mathbf{W}^\top \mathbf{x} + \mathbf{b})$
Probabilistic model	Gaussian $\mathcal{N}(\hat{y}, \sigma^2)$	categorical $\operatorname{Cat}(\hat{\mathbf{y}})$
Loss (NLL)	$\frac{1}{2}(\hat{y} - y)^2$	$-\sum_{j=1}^{q} y_j \log \hat{y}_j = -\log \hat{y}_y$

	Linear regression	Softmax regression
Gradient	$\widehat{y} - y$ (w.r.t. $\widehat{y}$)	$\widehat{\mathbf{y}} - \mathbf{y}$ (w.r.t. $\mathbf{o}$)

3.1.2.3 Entropy and Cross-Entropy

The name *cross-entropy* comes from Claude Shannon's information theory (Section 37.1). When an outcome with probability p occurs, Shannon defined its *surprisal* (information content) as $-\log p$ (measured in *nats* when using natural logarithms, or *bits* when using $\log_2$, with $1 \text{ nat} = 1/\log 2 \approx 1.44$ bits). For a true distribution $\mathbf{p} = (p_1, \dots, p_q)$ over q classes, the expected surprisal is its *entropy*:

$$H(\mathbf{p}) = -\sum_{j=1}^{q} p_j \log p_j, \tag{3.8}$$

which ranges from 0 (when one outcome is certain) to $\log q$ (when all q outcomes are equally likely). If an observer instead encodes outcomes drawn from $\mathbf{p}$ using a code optimized for a model distribution $\widehat{\mathbf{p}}$, their average code length (expected surprisal) is the *cross-entropy*:

$$H(\mathbf{p}, \widehat{\mathbf{p}}) = -\sum_{j=1}^{q} p_j \log \widehat{p}_j. \tag{3.9}$$

By *Gibbs' inequality*, $H(\mathbf{p}, \widehat{\mathbf{p}}) \geq H(\mathbf{p})$ with equality if and only if $\widehat{\mathbf{p}} = \mathbf{p}$ (their gap is the Kullback–Leibler divergence $D_{\mathrm{KL}}(\mathbf{p} \,\|\, \widehat{\mathbf{p}}) = H(\mathbf{p}, \widehat{\mathbf{p}}) - H(\mathbf{p}) \geq 0$). Comparing $H(\mathbf{p}, \widehat{\mathbf{p}})$ with Equation 3.7, our per-example loss $l(\mathbf{y}, \widehat{\mathbf{y}}) = H(\mathbf{y}, \widehat{\mathbf{y}})$ is literally the cross-entropy between the empirical label distribution $\mathbf{y}$ and the predicted distribution $\widehat{\mathbf{y}}$. Better still, when an input is inherently ambiguous, say a blurry photo that is 60% likely to be a shirt and 40% likely to be a T-shirt, Gibbs' inequality guarantees that the expected cross-entropy loss is uniquely minimized when the model reports the true calibrated probabilities $(0.6, 0.4)$ rather than an overconfident guess.

3.1.3 Accuracy and Confidence

3.1.3.1 The Error Rate

Although we evaluate classifiers by *accuracy* (or *error rate* $= 1 -$ accuracy), we cannot train by differentiating the zero-one error directly. Define the *margin* of an example as its true class's logit minus the highest competing logit:

$$m = o_y - \max_{j \neq y} o_j. \tag{3.10}$$

The classifier makes an error ($m < 0$) or a correct prediction ($m > 0$) according to a discontinuous step function of m (Figure 3.3) whose derivative is zero everywhere it exists, starving gradient descent of any learning signal. Cross-entropy replaces that flat step with a smooth, strictly decreasing convex surrogate ($\log(1+e^{-m})$ for $q = 2$) that upper-bounds the zero-one error (when measured in bits by dividing by $\log 2$) and supplies a nonzero gradient for every example.

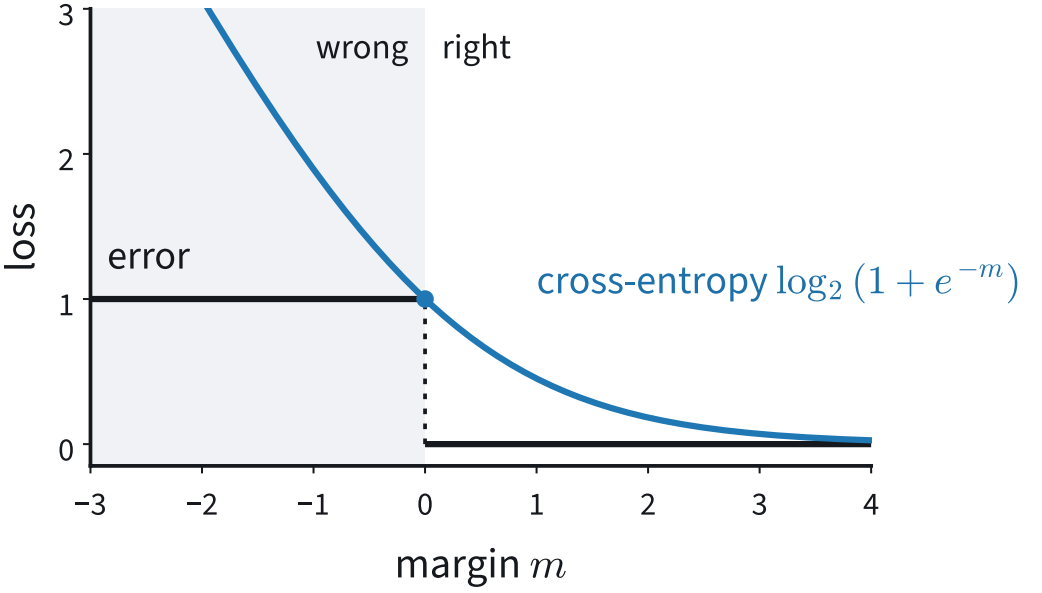


Figure 3.3: Zero-one classification error and two-class cross-entropy loss (in bits, $\log_2(1+e^{-m})$) as functions of the margin $m = o_y - \max_{j \neq y} o_j$.

3.1.3.2 Calibration and Temperature

Notice one consequence of the smooth curve in Figure 3.3: cross-entropy keeps decreasing even after $m > 0$ and the classification decision is already correct. If a training dataset is *linearly separable*, meaning some $(\mathbf{W}, \mathbf{b})$ classifies every training example correctly ($m^{(i)} > 0$ for all i), then multiplying $(\mathbf{W}, \mathbf{b})$ by any scale factor $c > 1$ multiplies every margin by c and drives the training loss toward zero without bound (Exercise 6), pushing all predicted probabilities toward 0 and 1 without moving the decision boundary at all.

More generally, scaling all logits by the inverse of a positive *temperature* $T > 0$ before applying the softmax,

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{o}/T), \tag{3.11}$$

sharpens the distribution toward a one-hot argmax as $T \to 0$ and flattens it toward the uniform distribution $(1/q, \ldots, 1/q)$ as $T \to \infty$ (Figure 3.4), while leaving the ranking of the logits (and therefore the classifier's accuracy) unchanged. Whether a trained model's predicted probabilities match true empirical accuracy (*calibration*) is an empirical question we test on held-out data in Section 3.4.

3.1.4 Summary

Softmax regression predicts unordered discrete categories by computing one affine logit per class, $\mathbf{o} = \mathbf{W}^\top \mathbf{x} + \mathbf{b}$, and normalizing them into probabilities $\hat{y}_j = \exp(o_j)/\sum_k \exp(o_k)$, partitioning input space into convex polyhedral decision regions bounded by hyperplanes. Because the softmax operator depends only on pairwise logit differences $o_i - o_j =$

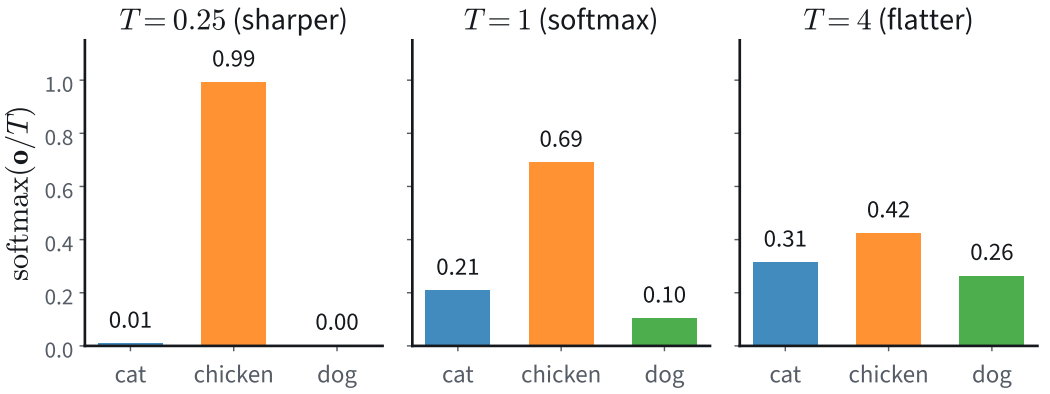


Figure 3.4: Softmax probabilities for logits $\mathbf{o} = (1.0, 2.2, 0.3)$ at three temperatures: $T = 0.25$ (peaked), $T = 1$ (standard softmax), and $T = 4$ (nearly uniform).

$\log(\hat{y}_i/\hat{y}_j)$, it is invariant to uniform shifts of $\mathbf{o}$ and reduces to binary logistic regression when $q = 2$. Maximizing the categorical likelihood of one-hot targets $\mathbf{y}$ yields the convex cross-entropy loss $l(\mathbf{y}, \hat{\mathbf{y}}) = -\log \hat{y}_y = \log \sum_k \exp(o_k) - o_y$, a smooth upper bound on zero-one classification error whose gradient with respect to the logits collapses to the residual $\hat{\mathbf{y}} - \mathbf{y}$. Dividing the logits by a temperature $T > 0$ tunes prediction confidence between a deterministic argmax and a uniform distribution without altering class rankings or accuracy.

3.1.5 Exercises

1. **Softmax and sigmoid.** By Equation 3.5, logistic regression is softmax regression with two classes and one redundant logit removed. A colleague hands you a fitted logistic model $P(y = 1 \mid x) = \sigma(\hat{\beta}_0 + \hat{\beta}_1 x)$ with $\hat{\beta}_0 = 2$ and $\hat{\beta}_1 = -1$.
    1. Characterize all two-class softmax coefficients (w_1, b_1) and (w_2, b_2) whose predictions coincide with the logistic model for every x, and identify the pair with $o_2 \equiv 0$.
    2. Show that gradient descent on the cross-entropy loss of the two-class softmax model never changes $w_1 + w_2$ or $b_1 + b_2$. Suppose that gradient descent starts from $w_1 = w_2 = b_1 = b_2 = 0$ on the colleague's training data and converges to coefficients that reproduce the logistic model. Determine which pair it reaches.

    Adapted from James, Witten, Hastie, and Tibshirani, An Introduction to Statistical Learning, Exercise 4.12.

2. **Softmax Hessian.** As in Section 3.1.2.2, write the cross-entropy loss as a function of the logits, $l(\mathbf{y}, \mathbf{o}) = g(\mathbf{o}) - \mathbf{y}^\top \mathbf{o}$ with $g(\mathbf{o}) = \log \sum_k \exp(o_k)$.
    1. Compute the Hessian $\partial^2_{\mathbf{o}}\, l(\mathbf{y}, \mathbf{o})$ and show that it does not depend on $\mathbf{y}$.
    2. Let $\mathbf{e}_Y$ be the one-hot vector of a class Y drawn from softmax($\mathbf{o}$). Compute the covariance matrix of $\mathbf{e}_Y$ and show that it equals the Hessian from the previous step.
    3. Conclude that g is convex and that the cross-entropy loss is convex in the logits.

3. **Bradley–Terry model.** The Bradley–Terry model (Bradley and Terry 1952) uses a logistic model to capture preferences. A user who chooses between apples and oranges picks each item with the probability given by the softmax of two scores, o_{apple} and o_{orange}. Now add a default option of choosing neither, with a fixed score $o_{\text{none}} = 0$.

   1. Determine whether adding the same constant to o_{apple} and o_{orange} changes the choice probabilities, first without and then with the default option. Conclude which scores the observed choices can determine in each model.
   2. Compare the ratio $P(\text{apple})/P(\text{orange})$ with and without the default option. Describe a choice situation in which real users would not keep this ratio fixed.

4. **Log-sum-exp.** The function $\operatorname{lse}(a, b) = \log(\exp(a) + \exp(b))$ is the log-sum-exp of Section 3.1.2.2 with two arguments, so its gradient is the softmax of (a, b).

   1. Prove that
   $$\max(a, b) < \operatorname{lse}(a, b) \le \max(a, b) + \log 2, \tag{3.12}$$
   with equality on the right exactly when $a = b$, and show that the gap to $\max(a, b)$ can be made arbitrarily small. Hint: without loss of generality set $b = 0$ and $a \ge 0$.
   2. Prove that for every $\lambda > 0$
   $$\max(a, b) < \lambda^{-1}\operatorname{lse}(\lambda a, \lambda b) \le \max(a, b) + \lambda^{-1}\log 2, \tag{3.13}$$
   and conclude that $\lambda^{-1}\operatorname{lse}(\lambda a, \lambda b) \to \max(a, b)$ as $\lambda \to \infty$.
   3. Construct an analogous smooth minimum.
   4. Extend the definition and both bounds to q numbers, showing that the gap to $\max_i a_i$ is at most $\log q$.

5. **Temperature.** Let $P = \operatorname{softmax}(\mathbf{o})$, and let the distribution at temperature T be $\operatorname{softmax}(\mathbf{o}/T)$. Suppose we pick another distribution Q with $Q(i) \propto P(i)^{\alpha}$ for $\alpha > 0$.

   1. Which choice of α corresponds to doubling the temperature? Which choice corresponds to halving it?
   2. Suppose that exactly s of the q logits attain the largest value. Find the limit of $\operatorname{softmax}(\mathbf{o}/T)$ as $T \to 0$.
   3. Let $\bar{o} = \frac{1}{q}\sum_j o_j$ be the mean logit. Show that for large T
   $$\operatorname{softmax}(\mathbf{o}/T)_i = \frac{1}{q} + \frac{o_i - \bar{o}}{qT} + O(T^{-2}). \tag{3.14}$$
   4. Temperature scaling fits a single T by minimizing the cross-entropy on a held-out set with the logits of the classifier held fixed. Show that this cross-entropy is a convex function of $\beta = 1/T$, so that the fit is a one-dimensional convex problem. Hint: use the convexity of g from exercise 2.

6. **Separable data.** ● Suppose that some parameters $(\mathbf{W}, \mathbf{b})$ classify every training example correctly and without ties: on each example, the logit of the true class strictly exceeds every other logit. Consider the scaled parameters $(c\mathbf{W}, c\mathbf{b})$ for $c > 0$.

   1. Show that the cross-entropy loss of every training example strictly decreases as c grows and tends to 0 as $c \to \infty$.
   2. Conclude that the training loss has no minimizer and that every sequence of parameters whose training loss tends to 0 has unbounded norm.

3. Add the weight decay penalty of Section 2.7, applied to both weights and biases, $\frac{\lambda}{2}(\|\mathbf{W}\|^2 + \|\mathbf{b}\|^2)$ with $\lambda > 0$, to the training loss. Show that the penalized loss has a minimizer.

7. **Squared error on probabilities.** Instead of cross-entropy, we could fit the predicted probabilities to the one-hot label with the squared loss $\frac{1}{2}\|\hat{\mathbf{y}} - \mathbf{y}\|^2$.
   1. Show that the derivatives of the softmax are $\partial \hat{y}_k / \partial o_j = \hat{y}_k(\mathbf{1}(j = k) - \hat{y}_j)$, and use them to compute the gradient of the squared loss with respect to the logits.
   2. For two classes, write both losses as functions of the margin m and compute their derivatives with respect to m. Show that as $m \to -\infty$, that is, for a confident mistake, the derivative of the squared loss tends to 0 while that of the cross-entropy tends to -1.
   3. Explain what this difference means for a model that starts training with confident wrong predictions.

8. code **Gumbel-max trick.** ● Suppose that a label is generated by perturbing every logit with independent noise and reporting the class with the largest perturbed score, $y = \operatorname{argmax}_i (o_i + \gamma_i)$. Gaussian noise gives the probit model, whose class probabilities have no closed form; one other noise distribution reproduces the softmax probabilities exactly. Let $\gamma_1, \ldots, \gamma_q$ be i.i.d. draws from the standard Gumbel distribution, whose CDF is $P(\gamma \le z) = \exp(-\exp(-z))$.
   1. Show that $P(o_k + \gamma_k \le z) = \exp(-\exp(o_k)\exp(-z))$.
   2. Show that $P\left(\operatorname{argmax}_i (o_i + \gamma_i) = k\right) = \operatorname{softmax}(\mathbf{o})_k$. Hint: condition on the value of $o_k + \gamma_k$ and use the substitution $u = \exp(-z)$.
   3. Conclude that scaling the noise, $\gamma_i \mapsto T\gamma_i$, samples from $\operatorname{softmax}(\mathbf{o}/T)$.
   4. Verify your result numerically: for $\mathbf{o} = (1.0, 2.2, 0.3)$ and $T \in \{0.25, 1, 4\}$, draw 10^5 samples of $\operatorname{argmax}_i(o_i + T\gamma_i)$ and confirm that the label frequencies match $\operatorname{softmax}(\mathbf{o}/T)$ up to Monte Carlo error.

3.2 Image Classification Data

To train and test softmax regression, we need an image classification benchmark that is rich enough to separate weak models from strong ones yet compact enough to train in seconds inside an interactive notebook. Classic MNIST handwritten digits (LeCun, Léon Bottou, et al. 1998) ($70{,}000$ grayscale 28×28 images across 10 digit classes) was a landmark benchmark in the 1990s, but today even trivial models clear 97% accuracy and convolutional networks exceed 99%, leaving almost no headroom to see architectural improvements work. At the opposite extreme, ImageNet (Deng et al. 2009) (1.2 million color photos across $1{,}000$ categories) requires hours of GPU compute per run.

Fashion-MNIST (Xiao et al. 2017) hits the sweet spot: it keeps MNIST's footprint ($60{,}000$ training and $10{,}000$ test grayscale images of size 28×28 across 10 balanced classes) while replacing digits with articles of clothing (T-shirt, trouser, pullover, dress, coat, sandal, shirt, sneaker, bag, and ankle boot). On Fashion-MNIST, linear softmax regression reaches roughly 83% accuracy while convolutional networks climb past 90%. In this section, we build the `FashionMNIST(d2l.DataModule)` class that feeds our image classifiers across the next several chapters.

```
import time
import torchvision
from torchvision import transforms
```

```
d2l.use_svg_display()
```

3.2.1 Loading the Dataset

Fashion-MNIST contains $6{,}000$ training images and $1{,}000$ test images for each of its 10 garment categories ($60{,}000$ and $10{,}000$ in total), stored as 8-bit unsigned integers in $[0, 255]$ (0 = dark background). Following our object-oriented design (Section 2.4), `FashionMNIST` subclasses `d2l.DataModule` and accepts `batch_size` and an optional spatial `resize` tuple (defaulting to $(28, 28)$):

`torchvision.datasets.FashionMNIST` downloads the dataset and keeps all images of a split in one `uint8` tensor. The data loader that we define below reads this tensor and converts whole minibatches at a time. The `transforms.Compose` pipeline resizes one image and scales its pixel intensities to a float32 tensor in $[0, 1]$. It applies whenever we index the dataset directly, as in `data.train[i]`, or pass `data.train` to a loader of our own.

```
class FashionMNIST(d2l.DataModule):
    """The Fashion-MNIST dataset."""
    def __init__(self, batch_size=64, resize=(28, 28)):
        super().__init__()
        self.save_hyperparameters()
        trans = transforms.Compose([transforms.Resize(resize),
                                    transforms.ToTensor()])
        self.train = torchvision.datasets.FashionMNIST(
            root=self.root, train=True, transform=trans, download=True)
        self.val = torchvision.datasets.FashionMNIST(
            root=self.root, train=False, transform=trans, download=True)
```

Let us instantiate the dataset and verify the size of each split.

```
data = FashionMNIST()
len(data.train), len(data.val)
```

```
(60000, 10000)
```

Because Fashion-MNIST ships with only two splits, we use the $10{,}000$-image test split as our validation set during training (we analyze the statistical effect of repeated test-set evaluation in Chapter 3.5). Let us also attach `text_labels` to translate integer class indices $0, \ldots, 9$ into human-readable garment names:

```
@d2l.add_to_class(FashionMNIST)
def text_labels(self, indices):
    """Return text labels."""
    labels = ['T-shirt', 'trouser', 'pullover', 'dress', 'coat',
              'sandal', 'shirt', 'sneaker', 'bag', 'ankle boot']
```

```
    return [labels[int(i)] for i in indices]
```

3.2.2 Reading Minibatches

In deep learning frameworks, each image in a minibatch of size n with c channels ($c = 1$ for grayscale, $c = 3$ for RGB color), height h, and width w is stored either in *channel-first* layout (n, c, h, w) or *channel-last* layout (n, h, w, c), depending on the framework.

PyTorch stores images in channel-first layout (n, c, h, w).

Now we attach `get_dataloader(self, train)` to `FashionMNIST`, wrapping the framework's native data loader so it reshuffles the training split on every epoch and retains the final partial minibatch ($60{,}000 = 937 \times 64 + 32$) so every image is visited once per epoch:

A `batchify` helper scales each minibatch to float32 in $[0, 1]$ and resizes it as the loader reads it. One operation on a whole minibatch is much cheaper than a separate operation for each image, which is why the loader does not go through the per-image transform.

```
@d2l.add_to_class(FashionMNIST)
def get_dataloader(self, train):
    data = self.train if train else self.val
    # Serve the 8-bit image tensor directly, not one PIL image at a time
    X = data.data.unsqueeze(1)  # Add the channel axis
    return torch.utils.data.DataLoader(
        torch.utils.data.TensorDataset(X, data.targets), self.batch_size,
        shuffle=train, collate_fn=self.batchify)

@d2l.add_to_class(FashionMNIST)
def batchify(self, examples):
    """Stack examples into a minibatch, scale to [0, 1], resize if asked."""
    X, y = torch.utils.data.default_collate(examples)
    X = X.float() / 255
    if X.shape[2:] != tuple(self.resize):
        X = F.interpolate(X, size=self.resize, mode='bilinear', antialias=True)
    return X, y
```

Let us fetch one minibatch `(X, y)` from `train_dataloader()` and inspect both the single-image layout and the full minibatch tensor shapes and dtypes:

```
X, y = next(iter(data.train_dataloader()))
X[0].shape  # Channel-first: (channels, height, width)
```

```
torch.Size([1, 28, 28])
```

Stacking examples into a minibatch prepends a leading batch axis to the images, while the labels arrive as a one-dimensional vector with one entry per image.

```
print(X.shape, X.dtype, y.shape, y.dtype)
```

```
torch.Size([64, 1, 28, 28]) torch.float32 torch.Size([64]) torch.int64
```

At $64 \times 28 \times 28 \times 4 \approx 200$ KB per minibatch (and ~ 190 MB for all $60{,}000$ float32 training images), the entire dataset fits comfortably in RAM. Still, benchmarking the data loader by itself—timing one complete pass over `train_dataloader()` with no model attached—is a vital habit, because no training epoch can run faster than its data pipeline.

```
tic = time.time()
for X, y in data.train_dataloader():
    continue
f'{time.time() - tic:.2f} sec'
```

```
'0.42 sec'
```

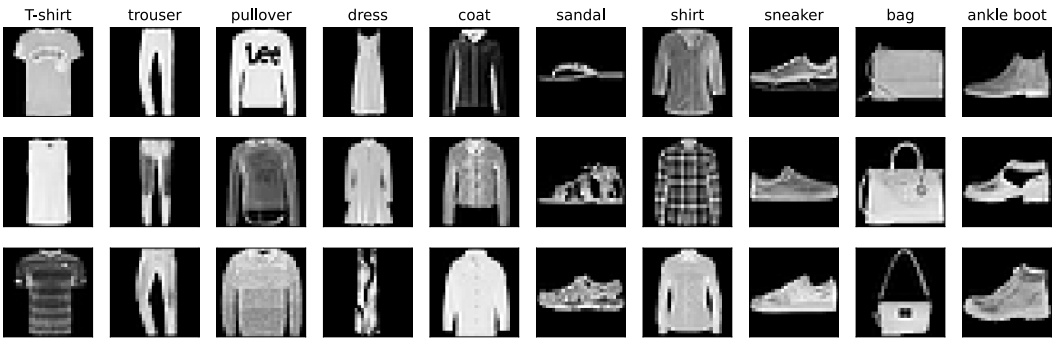
3.2.3 Inspecting the Images

Always inspect a sample of labeled images by eye before training any model: misaligned labels, inverted intensities, or a scrambled reshape will ruin training without raising a runtime error. We attach a `visualize` method to `FashionMNIST` that drops the singleton channel axis and plots a grid of grayscale images titled with their text labels:

```
@d2l.add_to_class(FashionMNIST)
def visualize(self, batch, nrows=1, ncols=8, labels=None):
    """Show the images of a batch in grayscale, titled by their labels."""
    X, y = batch
    if labels is None:
        labels = self.text_labels(y)
    X = d2l.reshape(X, (-1, *self.resize))  # Drop the channel axis
    d2l.show_images(X, nrows, ncols, titles=labels, cmap='gray')
```

Let us display the first three validation images of every class, one column per category, taking advantage of the fact that the validation loader always returns images in deterministic order:

```
X, y = next(iter(data.val_dataloader()))
labels = d2l.numpy(y)
idx = [int(np.flatnonzero(labels == c)[r]) for r in range(3) for c in range(10)]
titles = data.text_labels(range(10)) + [''] * 20
data.visualize([d2l.stack([X[i] for i in idx]), None], nrows=3, ncols=10,
               labels=titles)
```



Even a quick visual scan previews what our linear classifier is up against: trousers and bags have unmistakable silhouettes, sandals/sneakers/ankle boots form a distinct footwear cluster, and the four upper-body garments (T-shirt, pullover, coat, and shirt) share almost the same boxy torso outline, differing only in subtle collar, sleeve, and button details that 28×28 resolution barely resolves.

3.2.4 Summary

Fashion-MNIST provides $60{,}000$ training and $10{,}000$ test grayscale 28×28 images across 10 balanced clothing categories, making it rich enough to separate linear models ($\sim 83\%$) from convolutional networks ($> 90\%$) while small enough to train in seconds. Our `FashionMNIST(d2l.DataModule)` pipeline rescales raw `uint8` pixels to `float32` in $[0, 1]$, supports optional spatial resizing, and delivers reshuffled minibatches alongside integer labels in each framework's native channel layout. Benchmarking the standalone loader sets a lower bound on epoch time, and inspecting a labeled gallery catches silent shape or label mismatches before training starts.

3.2.5 Exercises

1. code **Class means.** Compute the mean image of each class over the training set, and show the ten means as a grid of images.
   1. Compute the Euclidean distance between every pair of class means and present the distances as a 10×10 table. Which pairs are closest, and which class is farthest from all others?
   2. Predict from the table which classes a linear classifier will confuse most often and which it will almost never confuse. Keep your prediction; Section 3.3 computes the answer.
   3. Name a pair of classes that a person tells apart easily although their means are close, and say which pixels carry the difference.
2. code **Loader throughput.** The time of one pass over the training set depends on the minibatch size and on the size of the images that the loader delivers.
   1. Time one full pass of `train_dataloader` for `FashionMNIST` constructed with `batch_size` of 1, 16, 64, 256, and 1024, and plot the throughput in images per second against `batch_size`. Explain the shape of the curve in terms of per-batch overhead and per-image work.
   2. `batchify` resizes each minibatch as the loader reads it. Time one pass of `train_dataloader` for `FashionMNIST(resize=(s, s))` with s = 28, 56, 112, and 224, and plot the time against the number of pixels per image. Which part of the time grows with the number of pixels, and which does not?
3. code **Upsampling and linear models.** ● Treat the resizing from 28×28 to 32×32 as a linear map $\mathbf{x}' = \mathbf{A}\mathbf{x}$ from the 784 original pixels to the 1024 resized ones, where the columns of $\mathbf{A} \in \mathbb{R}^{1024 \times 784}$ are linearly independent.
   1. Show that softmax regression on $\mathbf{x}'$ can represent exactly the classifiers that softmax regression on $\mathbf{x}$ can.
   2. After Section 3.3, train `SoftmaxRegressionScratch` on data from `FashionMNIST()` and from `FashionMNIST(resize=(32, 32))`, with `num_inputs` set to the number of pixels per image. Compare the validation accuracies and explain any difference in light of the previous part.

3.3 Softmax Regression Implementation

Just as we implemented linear regression both from scratch and concisely in Chapter 2, we now build softmax regression on Fashion-MNIST both ways. We start from raw tensors—implementing the rowwise softmax, vectorized cross-entropy loss, and `SoftmaxRegressionScratch` parameters ($\mathbf{W} \in \mathbb{R}^{784\times 10}, \mathbf{b} \in \mathbb{R}^{10}$) alongside the reusable `Classifier(d2l.Module)` base class—train to roughly 83% validation accuracy, and inspect what the linear model learned through its 10×10 confusion matrix and 28×28 weight templates. In Section 3.3.5, we then replace manual tensors and probability clamping with a built-in linear layer and a shifted *log-sum-exp* cross-entropy loss computed directly from raw logits, eliminating floating-point overflow and underflow at once.

3.3.1 The Softmax

3.3.1.1 Normalizing Each Row

For a minibatch of n examples, the logits form an $n\times q$ matrix $\mathbf{O} = \mathbf{X}\mathbf{W}+\mathbf{b}$ (Section 3.1.1.4). Softmax normalizes each row independently:

$$\operatorname{softmax}(\mathbf{O})_{ij} = \frac{\exp(O_{ij})}{\sum_{k=1}^{q}\exp(O_{ik})}. \tag{3.15}$$

In code, we exponentiate `O`, sum across columns (`axis=1`) with `keepdims=True` so the row sums keep shape `(n, 1)`, and divide via broadcasting:

```
def softmax(O):
    O_exp = d2l.exp(O)
    return O_exp / d2l.reduce_sum(O_exp, 1, keepdims=True)  # Broadcast
```

Let us sanity-check `softmax` on two rows: the cat/chicken/dog logits $(1.0, 2.2, 0.3)$ from Section 3.1, and a row of three identical zeros.

```
O = d2l.tensor([[1.0, 2.2, 0.3], [0.0, 0.0, 0.0]])
Y_hat = softmax(O)
Y_hat, d2l.reduce_sum(Y_hat, 1)
```

```
(tensor([[0.2076, 0.6893, 0.1031],
         [0.3333, 0.3333, 0.3333]]),
 tensor([1., 1.]))
```

3.3.1.2 Overflow

Why should you never deploy this three-line `softmax` formula in production? Because in 32-bit floating point (`float32`), numbers top out near 3.4×10^{38}, so $\exp(o)$ overflows to `inf` as soon as $o > 88.7$. Feed `softmax` the logits $(1000, 0, 0)$ and ∞/∞ evaluates to `NaN`, even though the true probability distribution is $(1, 0, 0)$ to machine precision.

As we proved in Section 3.1, softmax is invariant to shifting every logit in a row by the same constant. Subtracting the row maximum $o_{\max} = \max_k o_k$ before exponentiating ensures

every exponent $o_j - o_{\max}$ is ≤ 0 (and the largest is 0), making overflow impossible. Every framework's built-in softmax applies this shift internally:

```
O = d2l.tensor([[1000.0, 0.0, 0.0]])
O_max = O.amax(dim=1, keepdim=True)
softmax(O), softmax(O - O_max), torch.softmax(O, dim=1)
```

```
(tensor([[nan, 0., 0.]]), tensor([[1., 0., 0.]]), tensor([[1., 0., 0.]]))
```

For our Fashion-MNIST scratch run below, pixel values lie in $[0, 1]$ and weights stay small, so `softmax` does not overflow during training; however, shifting alone still leaves very negative logits vulnerable to *underflow* ($\exp(-200) = 0.0$ in float32, turning $\log 0$ into $-\infty$). In Section 3.3.5 below, we eliminate both hazards permanently by fusing softmax and the logarithm in log space.

3.3.2 Loss and Accuracy

3.3.2.1 Two Uses of the Logits

While gradient descent optimizes the smooth cross-entropy loss, humans judge a classifier by its *accuracy*—the fraction of examples where the top-scoring class matches the true label $y^{(i)}$:

$$\text{accuracy} = \frac{1}{n}\sum_{i=1}^{n} \mathbf{1}\Big(\operatorname*{argmax}_j o_j^{(i)} = y^{(i)}\Big). \tag{3.16}$$

Both metrics flow from the exact same logit vector (Figure 3.5), and tracking both during validation is essential: if a model becomes overconfident on a few mistakes, its validation cross-entropy loss can rise even while its validation accuracy is still climbing.

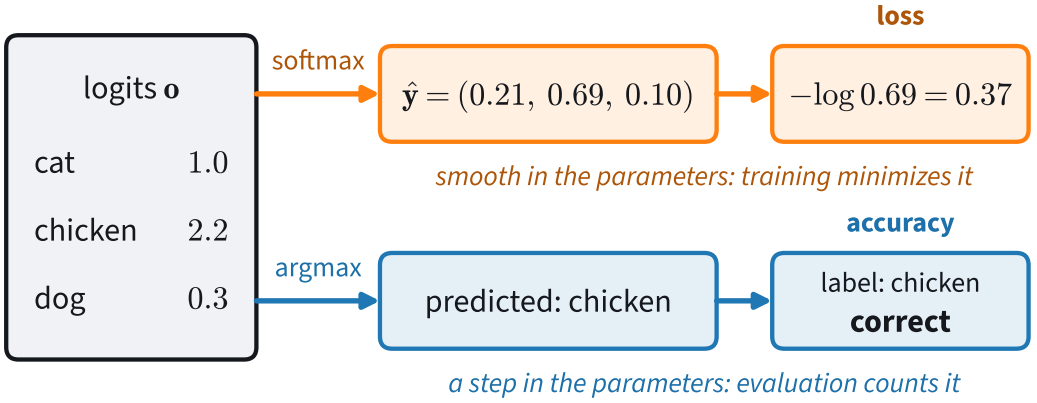


Figure 3.5: Training (top) maps logits through softmax to the smooth cross-entropy loss; evaluation (bottom) compares $\operatorname{argmax}_j o_j$ against the label y to compute discrete accuracy.

3.3.2.2 The `Classifier` Class

We package this dual-metric validation behavior into `Classifier(d2l.Module)`, the base class for every classification model in the book, starting with `validation_step` returning both `'loss'` and `'acc'` on each validation minibatch:

```
class Classifier(d2l.Module):
    """The base class of classification models."""
    def validation_step(self, batch):
        Y_hat = self(*batch[:-1])
        return {'loss': self.loss(Y_hat, batch[-1]),
                'acc': self.accuracy(Y_hat, batch[-1])}
```

Next, we attach `accuracy(self, Y_hat, Y, averaged=True)` to `Classifier`: it flattens `Y_hat` to one row of class scores per example, finds the predicted class index via `argmax` along `axis=1`, casts it to `Y.dtype`, and returns either the mean or per-example float32 equality mask `preds == Y`:

```
@d2l.add_to_class(Classifier)
def accuracy(self, Y_hat, Y, averaged=True):
    """Compute the fraction of correct predictions."""
    Y_hat = d2l.reshape(Y_hat, (-1, Y_hat.shape[-1]))
    preds = d2l.astype(d2l.argmax(Y_hat, axis=1), Y.dtype)
    compare = d2l.astype(preds == d2l.reshape(Y, (-1,)), d2l.float32)
    return d2l.reduce_mean(compare) if averaged else compare
```

Finally, we install plain minibatch SGD at learning rate `self.lr` as the default `configure_optimizers` method on `d2l.Module` so every classifier inherits a working optimizer out of the box:

```
@d2l.add_to_class(d2l.Module)
def configure_optimizers(self):
    return torch.optim.SGD(self.parameters(), lr=self.lr)
```

3.3.3 The Model and the Loss

3.3.3.1 The Parameters and Forward Pass

Flattening each 28×28 Fashion-MNIST image yields $d = 784$ input features for $q = 10$ classes, so `SoftmaxRegressionScratch` allocates $\mathbf{W} \in \mathbb{R}^{784\times 10}$ (initialized from $\mathcal{N}(0, 0.01^2)$) and $\mathbf{b} \in \mathbb{R}^{10}$ (initialized to 0), totaling $7{,}850$ parameters. Notice a quick sanity check: near-zero initial weights produce near-zero logits, so the untrained model starts close to a uniform prediction ($\hat{y}_j \approx 0.1$) with initial cross-entropy loss near $-\log(0.1) = \log 10 \approx 2.30$:

```
class SoftmaxRegressionScratch(Classifier):
    """Softmax regression implemented from scratch."""
    def __init__(self, num_inputs, num_outputs, lr, std=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.W = nn.Parameter(d2l.normal(0, std, (num_inputs, num_outputs)))
```

```
        self.b = nn.Parameter(d2l.zeros(num_outputs))
```

In `forward(self, X)`, we flatten each image in the minibatch into a row vector of length `self.num_inputs` (784) via `d2l.reshape(X, (-1, self.num_inputs))` and map the affine logits $\mathbf{XW} + \mathbf{b}$ through `softmax`:

```
@d2l.add_to_class(SoftmaxRegressionScratch)
def forward(self, X):
    X = d2l.reshape(X, (-1, self.num_inputs))  # One flattened image per row
    return softmax(d2l.matmul(X, self.W) + self.b)
```

3.3.3.2 The Cross-Entropy Loss

Recall from Section 3.1.2 that $l(\mathbf{y}, \hat{\mathbf{y}}) = -\log \hat{y}_y$. Instead of materializing one-hot matrices and taking inner products, we pluck out the true-class probability $\hat{y}^{(i)}_{y^{(i)}}$ for every row i in one vectorized indexing step:

```
y = d2l.tensor([0, 2])
y_hat = d2l.tensor([[0.1, 0.3, 0.6], [0.3, 0.2, 0.5]])
y_hat[[0, 1], y]
```

```
tensor([0.1000, 0.5000])
```

That extracts 0.1 for the first example and 0.5 for the second. Wrapping this in `cross_entropy`, we average the negative logarithms across the minibatch. Notice the one defensive clamp to 10^{-12} before taking the logarithm: it stops an underflowed zero probability from turning into $-\infty$, capping the per-example loss at 27.6, though clamping also zeroes out the gradient whenever it triggers. Later in this section (Section 3.3.5), we replace this stopgap with a fused loss computed directly from logits.

```
def cross_entropy(y_hat, y):
    # Tiny clip to keep log finite when softmax outputs underflow to 0.
    p = y_hat[list(range(len(y_hat))), y].clamp(min=1e-12)
    return -d2l.reduce_mean(d2l.log(p))

cross_entropy(y_hat, y)
```

```
tensor(1.4979)
```

That yields the mean of $-\log 0.1 \approx 2.30$ and $-\log 0.5 \approx 0.69$, or roughly 1.50. Let us attach it as the model's `loss` method:

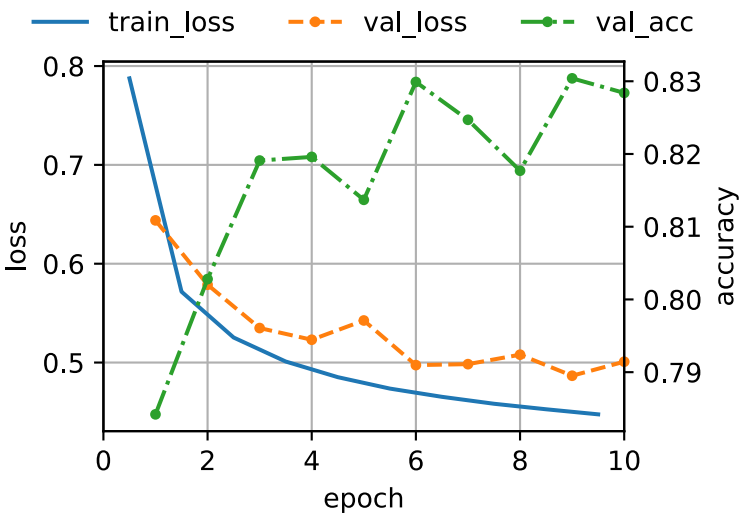
```
@d2l.add_to_class(SoftmaxRegressionScratch)
def loss(self, y_hat, y):
    return cross_entropy(y_hat, y)
```

3.3.4 Training and Evaluation

3.3.4.1 The Training Run

Let us train `SoftmaxRegressionScratch` on Fashion-MNIST for 10 epochs with minibatch size 256 and learning rate $\eta = 0.1$:

```
data = d2l.FashionMNIST(batch_size=256)
model = SoftmaxRegressionScratch(num_inputs=784, num_outputs=10, lr=0.1)
trainer = d2l.Trainer(max_epochs=10)
trainer.fit(model, data)
```



To score a trained model on an entire dataset split whenever we want, we attach an `evaluate(self, model, data, train=False)` helper to `d2l.Trainer` that binds `self.val_dataloader` to either the validation or training loader and runs `self.validate()`:

```
@d2l.add_to_class(d2l.Trainer)
def evaluate(self, model, data, train=False):
    """Metrics of `model` on the validation data of `data` (on its training
    data if `train`), averaged over examples."""
    self.train_dataloader = None
    self.val_dataloader = (data.train_dataloader() if train
                           else data.val_dataloader())
    self.prepare_model(model)
    return self.validate()
```

Averaged across all 10,000 validation images, these metrics match the final points recorded in `trainer.history['val_loss'][-1]` and `trainer.history['val_acc'][-1]`.

```
trainer.evaluate(model, data)
```

```
{'loss': 0.500844775390625, 'acc': 0.8284}
```

Our classifier gets roughly 83% of the validation images right; passing `train=True` evaluates the exact same weights across all $60{,}000$ training images:

```
trainer.evaluate(model, data, train=True)
```

```
{'loss': 0.46587086588541665, 'acc': 0.8420666666666666}
```

After 10 epochs, validation accuracy settles near 83% (varying by $\pm 1\%$ across random seeds) and training accuracy sits near 84%. That tiny 1% generalization gap reflects the restricted capacity of a linear classifier: $7{,}850$ parameters cannot memorize $60{,}000$ images (Chapter 3.5).

3.3.4.2 Misclassified Images

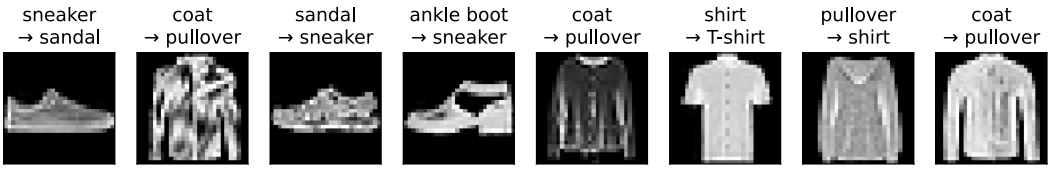
To see *how* the model errs, let us pull the first validation minibatch of 256 images and count the mistakes:

```
X, y = next(iter(data.val_dataloader()))
preds = d2l.astype(d2l.argmax(model(X), axis=1), y.dtype)
wrong = preds != y
print(f'{int(d2l.reduce_sum(d2l.astype(wrong, d2l.int32)))} of {len(y)} wrong')
```

```
35 of 256 wrong
```

Let us display the first eight misclassified images, showing the true label above the model's prediction (`true → pred`):

```
labels = [f'{true}\n→ {pred}' for true, pred in zip(
    data.text_labels(y[wrong]), data.text_labels(preds[wrong]))]
data.visualize([X[wrong], y[wrong]], labels=labels)
```



3.3.4.3 The Confusion Matrix

To summarize the error pattern across all $10{,}000$ validation images, we tally the 10×10 *confusion matrix* $\mathbf{N} \in \mathbb{R}^{q \times q}$, where rows index the predicted class $\hat{c}^{(m)} = \operatorname{argmax}_j o_j^{(m)}$ and columns index the true class $y^{(m)}$:

$$N_{ij} = \sum_{m=1}^{n} \mathbf{1}\big(\hat{c}^{(m)} = i\big)\, \mathbf{1}\big(y^{(m)} = j\big). \tag{3.17}$$

(Placing true classes in the columns and predicted classes in the rows makes column-normalized conditional probabilities multiply cleanly as $\mu = \mathbf{C}\mathbf{p}$ when we study label shift

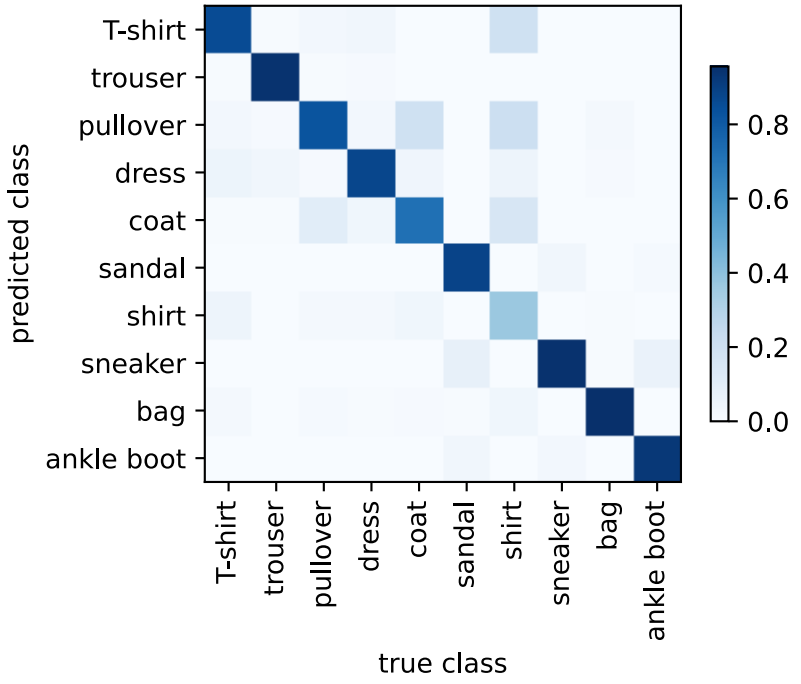
in Section 3.6.) Using one-hot matrices $\mathbf{P}, \mathbf{Y} \in \{0,1\}^{n \times q}$ for a minibatch's predicted and true classes, the product $\mathbf{P}^\top \mathbf{Y}$ accumulates all 100 counts in a single matrix multiplication per minibatch:

```
def one_hot(y, q):
    """Rows of the q x q identity matrix, one per class index in y."""
    return d2l.astype(d2l.reshape(y, (-1, 1)) == d2l.arange(q, dtype=y.dtype),
                      d2l.float32)

N = d2l.zeros((10, 10))
for X, y in data.val_dataloader():
    P = one_hot(d2l.argmax(model(X), axis=1), 10)
    N += d2l.matmul(d2l.transpose(P), one_hot(y, 10))
int(d2l.reduce_sum(N))
```

```
10000
```

All 10,000 validation images are accounted for. Because raw counts depend on class frequency, we normalize each column of $\mathbf{N}$ by its sum to obtain $C_{ij} = N_{ij} / \sum_{i'} N_{i'j}$, an empirical estimate of the conditional probability $P(\hat{c} = i \mid y = j)$. Each column shows where examples of true class j land, while the diagonal gives per-class recognition rates. Let us render $\mathbf{C}$ as a heatmap via `d2l.show_heatmaps`.

```
C = N / d2l.reduce_sum(N, 0, keepdims=True)  # Each column sums to 1
names = data.text_labels(range(10))
d2l.show_heatmaps(d2l.reshape(C, (1, 1, 10, 10)), xlabel='true class',
                  ylabel='predicted class', figsize=(4, 4), cmap='Blues',
                  xticklabels=names, yticklabels=names)
```



Let us also print the five largest off-diagonal entries to spotlight the most frequent confusions:

```
mistakes = d2l.numpy(N) * (1 - np.eye(10))  # Zero the diagonal
for k in np.argsort(mistakes, axis=None)[::-1][:5]:
    i, j = divmod(int(k), 10)
    print(f'{names[j]} taken for {names[i]}: {int(mistakes[i, j])}')
```

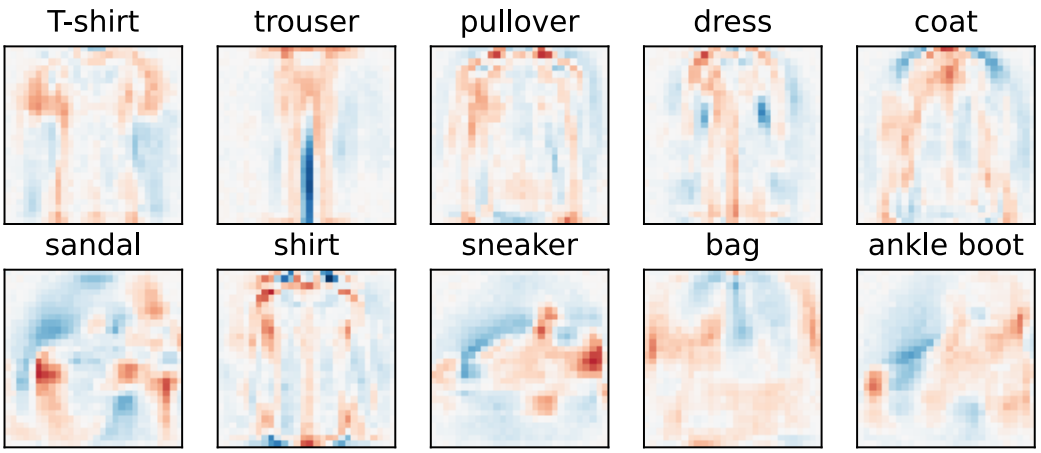
```
shirt taken for pullover: 207
coat taken for pullover: 192
shirt taken for T-shirt: 192
shirt taken for coat: 151
pullover taken for coat: 102
```

The confusion matrix confirms what our visual inspection in Section 3.2 predicted: trousers, bags, and footwear are recognized $\sim 90\%$ of the time or better, whereas the five largest off-diagonal confusions all occur among the four look-alike upper-body tops (T-shirt, pullover, coat, and shirt)—with shirts recognized only about half the time or less because their silhouette sits right in the middle of the other three.

3.3.4.4 The Learned Templates

Because each logit $o_j = \mathbf{w}_j^\top \mathbf{x} + b_j$ is an inner product between the flattened image $\mathbf{x}$ and column j of $\mathbf{W}$, reshaping each 784-element column $\mathbf{w}_j$ back to 28×28 lets us view the entire classifier as ten visual *templates* (red = positive weights, blue = negative weights):

```
W = d2l.numpy(model.W)
templates = W.T.reshape(10, 28, 28)  # Column j of W as an image
v = np.abs(templates).max()
d2l.show_images(templates, 2, 5, titles=names, cmap='RdBu_r', vmin=-v, vmax=v);
```



Each template is a recognizable ghostly silhouette of its garment—notice the blue negative channel between the trouser legs that penalizes dresses, and the similar torso shapes of the four tops that differ mainly around the collar and sleeves. Because a linear layer can only take a single fixed global dot product per class, subtle local features like collars or buttons cannot overcome overall silhouette overlap; breaking past $\sim 83\%$ requires nonlinear hidden layers (Chapter 4) and spatial convolutions (Chapter 6).

3.3.5 Concise Implementation

Now let us see how deep learning frameworks simplify softmax regression while permanently curing the overflow and underflow hazards of our naive `softmax` and clamped `cross_entropy`.

3.3.5.1 Defining the Model

In `SoftmaxRegression(d2l.Classifier)`, a flattening step and a single built-in fully connected layer with 10 outputs compute the logits $\mathbf{O} = \mathbf{X}\mathbf{W} + \mathbf{b}$ (initialized with $\mathcal{N}(0, 0.01^2)$ weights and zero biases to match our scratch baseline). Crucially, **`forward` returns the raw logits `O` directly, without applying softmax**:

```
class SoftmaxRegression(d2l.Classifier):
    """The softmax regression model."""
    def __init__(self, num_outputs, lr):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(nn.Flatten(),
                                 nn.LazyLinear(num_outputs))

    def weight_init(self, module):
```

```
        # The Trainer calls this after the lazy layer has learned its shape
        if isinstance(module, nn.Linear):
            nn.init.normal_(module.weight, std=0.01)
            nn.init.zeros_(module.bias)

    def forward(self, X):
        return self.net(X)
```

3.3.5.2 A Stable Cross-Entropy Loss

Why does `SoftmaxRegression.forward` stop at the logits instead of returning probabilities? Recall the twin floating-point hazards of computing $\hat{\mathbf{y}} = \text{softmax}(\mathbf{o})$ first and $-\log \hat{y}_y$ second in `float32`:

1. **Overflow:** $\exp(o)$ overflows to `inf` for $o > 88.7$. Subtracting $o_{\max} = \max_k o_k$ from every logit fixes overflow because every shifted exponent $o_k - o_{\max}$ is ≤ 0.
2. **Underflow:** Even after shifting by $o_{\max}$, any class whose logit sits more than 104 below $o_{\max}$ (such as class 1 in $\mathbf{o} = (0, -200, 0)$) has $\exp(o_y - o_{\max})$ underflow to exact `0.0` in float32. Taking $\log(0.0)$ then produces `-inf` (or, with our scratch `1e-12` clamp, caps the loss at 27.6 and **zeroes out the gradient**, leaving the model unable to learn from a confident blunder).

We avoid both overflow and underflow by *never materializing intermediate probabilities at all*. Combining the logarithm and the shifted softmax algebraically via the *LogSumExp trick* turns the division into a subtraction:

$$l(y, \mathbf{o}) = \log \sum_{k=1}^{q} \exp(o_k) - o_y = o_{\max} + \log \sum_{k=1}^{q} \exp(o_k - o_{\max}) - o_y. \tag{3.18}$$

Because every shifted exponent $o_k - o_{\max}$ is ≤ 0 and at least one equals 0, the sum inside the logarithm is trapped strictly inside $[1, q]$, and its logarithm lies strictly inside $[0, \log q]$. Neither overflow nor $\log 0$ can occur, and automatic differentiation yields the finite gradient $\text{softmax}(\mathbf{o}) - \mathbf{y}$.

Every framework provides this fused log-sum-exp cross-entropy. We install it once on `d2l.Classifier.loss` so every classifier in the rest of the book inherits it:

```
@d2l.add_to_class(d2l.Classifier)
def loss(self, Y_hat, Y, averaged=True):
    Y_hat = d2l.reshape(Y_hat, (-1, Y_hat.shape[-1]))
    Y = d2l.reshape(Y, (-1,))
    return F.cross_entropy(
        Y_hat, Y, reduction='mean' if averaged else 'none')
```

Testing `SoftmaxRegression.loss` on the extreme logits `O = [[1000.0, 0.0, 0.0], [1000.0, 0.0, 0.0]]` with true labels `y = [0, 1]` confirms that the fused LogSumExp implementation returns the exact per-example losses `[0., 1000.]` without overflow or underflow:

```
O = d2l.tensor([[1000.0, 0.0, 0.0], [1000.0, 0.0, 0.0]])
y = d2l.tensor([0, 1])
SoftmaxRegression(num_outputs=3, lr=0.1).loss(O, y, averaged=False)
```

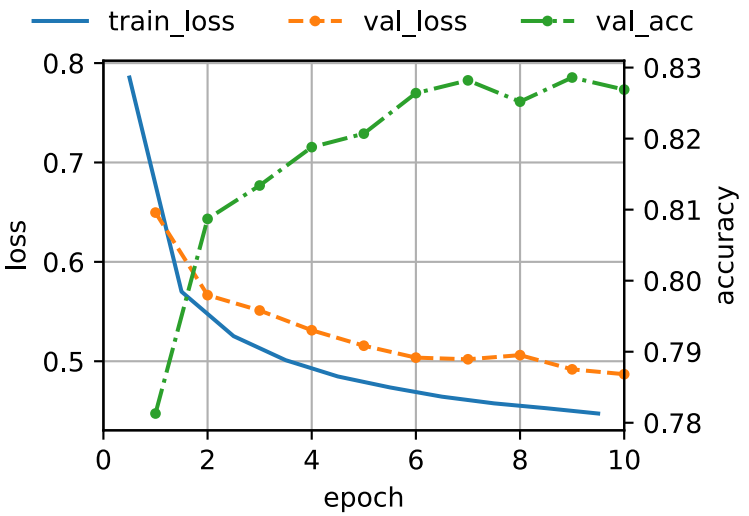
```
tensor([  -0., 1000.])
```

(Watch out for one classic pitfall: because `Classifier.loss` expects raw logits and applies log-softmax internally, accidentally placing a softmax at the end of `forward` applies softmax *twice*, trapping the training loss above a positive floor; see Exercise 4 below.)

3.3.5.3 Training

Training `SoftmaxRegression` with `d2l.Trainer` takes four lines and reproduces our $\sim 83\%$ validation accuracy with rock-solid numerical stability:

```
data = d2l.FashionMNIST(batch_size=256)
model = SoftmaxRegression(num_outputs=10, lr=0.1)
trainer = d2l.Trainer(max_epochs=10)
trainer.fit(model, data)
```



3.3.6 Summary

Implementing softmax regression from scratch introduces the reusable `Classifier` base class, which pairs smooth cross-entropy optimization during training with discrete argmax accuracy during validation, reaching roughly 83% validation accuracy on Fashion-MNIST in ten epochs. Accumulating one-hot products $\mathbf{P}^\top \mathbf{Y}$ builds the 10×10 confusion matrix and reveals that linear errors concentrate among the four upper-body garments whose learned 28×28 weight templates overlap strongly. Because computing `softmax` and `log` separately overflows for large positive logits and underflows to $\log 0$ for large negative logits, our concise implementation returns raw logits from `forward` and evaluates the shifted log-sum-exp

cross-entropy $o_{\max} + \log \sum_k \exp(o_k - o_{\max}) - o_y$, guaranteeing finite values and intact gradients across all logit magnitudes.

3.3.7 Exercises

1. code **Cross-entropy from one-hot labels.** The `cross_entropy` of this section selects the correct-class probability by indexing. Implement an alternative that follows the definition $-\sum_j y_j \log \hat{y}_j$ literally, by converting `y` to a one-hot matrix and taking row-wise inner products.
    1. Use it as the `loss` of `SoftmaxRegressionScratch` and confirm that training is unchanged.
    2. Time both versions on random probability matrices with 256 rows and $q \in \{10, 10^3, 10^5\}$ columns, and state how the memory that each version allocates grows with q.
    3. Describe a setting in which the inner-product form applies but the indexing form does not.
2. **Large vocabularies.** Suppose that softmax regression predicts the next word from a feature vector of dimension $d = 1024$. The vocabulary has $V = 10^6$ words, and each minibatch holds $n = 256$ examples. List the operations of `forward` and `cross_entropy` whose cost grows with V. Compute the multiply-adds of the matrix product, the float32 memory of `W`, and the float32 memory of the $n \times V$ matrix of probabilities, and compare each number with the Fashion-MNIST model of this section.
3. code **Templates and class means.** The templates are what minibatch SGD made of the pixels it saw.
    1. Train `SoftmaxRegressionScratch` again with `std=0`, so that every weight starts at 0 and every class starts with the same logit. Explain why training works all the same, and compare the accuracy and the templates with those of the text.
    2. Show that after one SGD step from zero weights, column j of $\mathbf{W}$ is proportional to $\sum_i \left(y_j^{(i)} - \frac{1}{10}\right) \mathbf{x}^{(i)}$ over the first minibatch: the sum of the images of class j minus a tenth of the sum of all images.
    3. Compute the mean image of each class over the training set and compare it with the trained template of that class, by eye and by the correlation of their pixels. Where do they agree, and where does a template depart from the mean? Explain why a template that merely copied the class mean would confuse the tops even more.
4. code **Double softmax.** The `loss` of this section expects logits. Suppose that a model's `forward` mistakenly ends with a softmax. The loss then receives a probability vector $\mathbf{p}$ in place of the logits and evaluates $-\log \operatorname{softmax}(\mathbf{p})_y$.
    1. Find the smallest value this loss can take over all probability vectors $\mathbf{p}$ and labels y, as a function of the number of classes q. Evaluate it for $q = 10$.
    2. Use the from-scratch `softmax` defined earlier in this section, which is not part of the `d2l` package. Subclass `SoftmaxRegression` so that `forward` applies this `softmax` to the logits. Train it with `lr=0.1`, `d2l.FashionMNIST(batch_size=256)`, and `d2l.Trainer(max_epochs=10)`. Compare its training loss and validation accuracy with those of `SoftmaxRegression`, and relate the final training loss to the value from the previous step.

5. code **Stable softplus.** For two classes with logits $(x, 0)$, the first term of the loss is $\log(1 + e^x)$, the *softplus* of x.
    1. Plot $\log(1+e^x)$ and $\max(x, 0)$ on $[-5, 5]$. Show that the first exceeds the second by at most $\log 2$, at $x = 0$, and explain in one sentence why log-sum-exp is called a smooth maximum.
    2. Show that $\log(1 + e^x) = \max(x, 0) + \log(1 + e^{-|x|})$.
    3. In single precision, evaluate the naive expression `log(1 + exp(x))`, the rewritten expression, and the rewritten expression with `log1p` in place of `log(1 + ...)`, for $x \in \{-100, -20, -5, 0, 5, 20, 100\}$. Compare each value with a double-precision reference, and report where each version overflows and where it loses all relative accuracy.
    4. Compute the derivative of the naive expression at $x = -20$ by automatic differentiation. Explain why it is accurate although the value of the expression is not, and what this implies for training and for reporting the loss of confidently classified examples.
6. code **The gradient survives.** For the logits $(0, -200, 0)$ with true label 1, the shifted softmax probability of class 1 underflows to 0 in float32, so the clamped scratch loss `cross_entropy(softmax(O - O_max), y)` hits its `1e-12` floor and has zero gradient with respect to the logits. Pass the same logits and label to the fused `loss` of `SoftmaxRegression` and compute the gradient with respect to the logits by automatic differentiation. Compare it with $\hat{\mathbf{y}} - \mathbf{y}$ from Section 3.1, and explain why the fused loss moves the logit of class 1 while the clamped scratch loss does not.
7. **Number formats.** Half-precision formats store a number in 16 bits: FP16 has 5 exponent bits and 10 mantissa bits, BFLOAT16 has 8 and 7, and FP32 has 8 and 23. Determine for each format the largest logit o for which $\exp(o)$ does not overflow. Then explain why a network whose logits reach 20 can be trained in BFLOAT16 but not in FP16 unless its softmax is computed in FP32. Finally, consider the shifted sum of Equation 3.18 for $q = 10^5$ classes, each term at most 1, accumulated in FP16: what goes wrong once the running sum exceeds 2,048, and what does the 10-bit mantissa have to do with it?

3.4 Measuring Classifiers

Our Fashion-MNIST softmax regression model classifies roughly 83% of validation images correctly. Does higher accuracy always mean a better classifier? Only when every class is equally important and every type of error carries the exact same cost. In real deployments, three everyday conditions break that symmetry:

1. **Class imbalance:** When the positive class of interest is rare (say, 1% of patients have a disease or 0.1% of transactions are fraudulent), a useless classifier that always predicts the majority class scores 99% or 99.9% accuracy while catching zero positives.
2. **Asymmetric error costs:** Missing a life-threatening illness (a false negative) is vastly costlier than ordering a follow-up test (a false positive).
3. **Probability-driven decisions:** Downstream decision rules act directly on the numerical predicted probabilities $\hat{\mathbf{y}}$, which are trustworthy only if the classifier is *calibrated*.

In this section, we develop the metrics and diagnostics for all three: *precision* and *recall*, *cost-sensitive decision thresholds*, *reliability diagrams* and *expected calibration error* (ECE), and post-hoc *temperature scaling*.

3.4.1 The Limits of Accuracy

3.4.1.1 Class Imbalance

Suppose we screen $100{,}000$ people for a disease with 1% prevalence ($1{,}000$ sick and $99{,}000$ healthy). A trivial rule that ignores every test result and always predicts *healthy* achieves 99% accuracy while detecting not a single sick patient. Now compare it against a real screening test that detects 90% of sick patients while falsely flagging 5% of healthy ones:

	sick (positive)	healthy (negative)
predicted sick	900 (TP)	4,950 (FP)
predicted healthy	100 (FN)	94,050 (TN)

In this 2×2 binary confusion matrix, the four cells are *true positives* (TP $= 900$), *false positives* (FP $= 4{,}950$), *false negatives* (FN $= 100$), and *true negatives* (TN $= 94{,}050$). Summing the diagonal gives $94{,}950/100{,}000 = 95\%$ accuracy, so raw accuracy ranks the useless "always-healthy" rule (99%) ahead of a lifesaving test (95%).

To evaluate performance on the positive class specifically, we pair two ratios: *recall* (or *sensitivity* / *true-positive rate*), the fraction of true positives caught, and *precision* (or *positive predictive value*), the fraction of positive flags that are truly positive:

$$\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}, \qquad \text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}. \tag{3.19}$$

Reading down the first column gives recall $= 900/1{,}000 = 0.90$ (versus 0 for "always healthy"). Reading across the first row gives precision $= 900/5{,}850 \approx 0.15$: only 15% of flagged people are actually sick, because 5% false alarms from the huge healthy majority ($4{,}950$) outnumber the 900 true positives (the Bayes' theorem effect from Section 1.6.5.2). In two-stage clinical screening, 90% recall with 15% precision is ideal: the cheap first test catches almost every sick patient and sends only the $5{,}850$ flagged patients to a definitive follow-up test.

Let us hold our test's intrinsic recall (0.90) and false-alarm rate (0.05) fixed while sweeping the disease prevalence across $\{0.1\%, 1\%, 10\%, 50\%\}$:

```
recall, false_alarms = 0.9, 0.05  # Properties of the test
for prevalence in [0.001, 0.01, 0.1, 0.5]:
    tp, fp = recall * prevalence, false_alarms * (1 - prevalence)
    accuracy = tp + (1 - false_alarms) * (1 - prevalence)
    print(f'{prevalence:5.1%} sick: accuracy {accuracy:.3f}, '
          f'recall {recall:.2f}, precision {tp / (tp + fp):.3f}')
```

```
 0.1% sick: accuracy 0.950, recall 0.90, precision 0.018
 1.0% sick: accuracy 0.950, recall 0.90, precision 0.154
10.0% sick: accuracy 0.945, recall 0.90, precision 0.667
50.0% sick: accuracy 0.925, recall 0.90, precision 0.947
```

Notice what changes and what stays fixed: recall and false-alarm rate are properties of the *classifier alone*, whereas precision depends heavily on the *population prevalence*.

3.4.1.2 Asymmetric Costs

Suppose missing a sick patient costs $\kappa_{\text{FN}} = 100$ whereas a false alarm costs $\kappa_{\text{FP}} = 1$ (and correct calls cost 0). Weighted by these penalties, the "always-healthy" rule incurs an expected cost of $100 \times 1{,}000/100{,}000 = 1.00$ per person, whereas our screening test costs $(100 \times 100 + 1 \times 4{,}950)/100{,}000 \approx 0.15$ per person, nearly seven times lower.

More generally, if a classifier estimates probability $p = P(\text{sick} \mid \mathbf{x})$ for an individual, predicting *healthy* has expected cost $p\,\kappa_{\text{FN}}$ and predicting *sick* has expected cost $(1-p)\,\kappa_{\text{FP}}$. Predicting *sick* minimizes expected cost whenever

$$p\,\kappa_{\text{FN}} > (1-p)\,\kappa_{\text{FP}}, \quad \text{that is,} \quad p > \frac{\kappa_{\text{FP}}}{\kappa_{\text{FP}} + \kappa_{\text{FN}}}. \tag{3.20}$$

Equal costs ($\kappa_{\text{FN}} = \kappa_{\text{FP}}$) recover the familiar $1/2$ threshold (argmax classification). With $\kappa_{\text{FN}} = 100$ and $\kappa_{\text{FP}} = 1$, however, the optimal threshold drops to $1/101 \approx 0.01$: even a 2% predicted risk warrants flagging the patient. Lowering the threshold raises recall at the expense of precision (when costs are unspecified, practitioners often summarize the trade-off via the harmonic mean of precision and recall, the *F1 score* $= 2\,\text{precision} \cdot \text{recall}/(\text{precision} + \text{recall})$, or sweep all thresholds via ROC and precision–recall curves; see Exercises 4 and 5).

3.4.2 Calibration

3.4.2.1 Reliability Diagrams

Cost-sensitive decision rules like Equation 3.20 act directly on the numerical value of p, which means they rely on the classifier being *calibrated*: among all examples where the classifier predicts confidence $\hat{p} \approx 0.80$, roughly 80% should actually be correct.

In a q-class problem, we check top-class calibration by recording each validation prediction's *confidence* $\hat{p}^{(i)} = \max_j \hat{y}_j^{(i)}$ and correctness indicator $c^{(i)} = \mathbf{1}(\text{argmax}_j\, \hat{y}_j^{(i)} = y^{(i)})$, partitioning the examples into confidence bins $[0, 0.1), [0.1, 0.2), \ldots, [0.9, 1.0]$, and comparing each bin's mean confidence $\bar{p}_b$ against its empirical accuracy a_b. Plotting a_b against $\bar{p}_b$ gives a *reliability diagram* (where a perfectly calibrated model lies along the diagonal $a_b = \bar{p}_b$), and their sample-weighted average gap defines the *expected calibration error* (ECE):

$$\text{ECE} = \sum_b \frac{n_b}{n} \left| a_b - \bar{p}_b \right|. \tag{3.21}$$

Let us train our concise `SoftmaxRegression` model on Fashion-MNIST, collect its validation logits `O` and labels `y`, and compute its per-bin reliability statistics, ECE, and reliability diagram:

```
data = d2l.FashionMNIST(batch_size=256)
model = d2l.SoftmaxRegression(num_outputs=10, lr=0.1)
trainer = d2l.Trainer(max_epochs=10)
trainer.board.display = False
trainer.fit(model, data)
O, y = [], []
```

```
for X, Y in data.val_dataloader():
    O.append(d2l.numpy(model(X)))
    y.append(d2l.numpy(Y))
O, y = np.concatenate(O), np.concatenate(y)
print(f'validation accuracy {(O.argmax(axis=1) == y).mean():.3f}')
```

```
validation accuracy 0.823
```

To compute each prediction's confidence, we apply the numerically shifted softmax (Section 3.3) to the logits in NumPy and take the maximum probability per row.

```
def softmax(O):
    """Row-wise softmax of a NumPy array of logits."""
    E = np.exp(O - O.max(axis=1, keepdims=True))
    return E / E.sum(axis=1, keepdims=True)

conf = softmax(O).max(axis=1)  # Confidence of each prediction
correct = (O.argmax(axis=1) == y).astype(float)
```

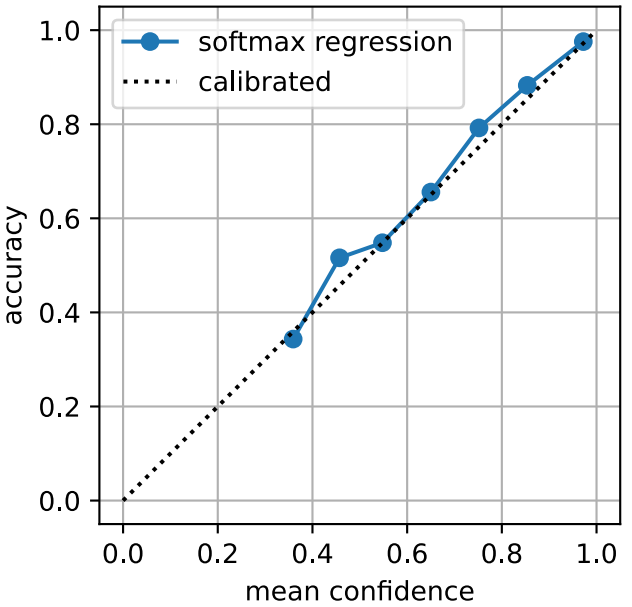
Next, `reliability(conf, correct, num_bins=10)` groups the validation predictions into ten equal-width confidence bins via `np.bincount`, returning the per-bin sample counts `n`, mean confidences `mean_conf`, empirical accuracies `acc`, and sample-weighted `ece`:

```
def reliability(conf, correct, num_bins=10):
    """Count, mean confidence, and accuracy per confidence bin; the ECE."""
    bins = np.minimum((conf * num_bins).astype(int), num_bins - 1)
    n = np.bincount(bins, minlength=num_bins)
    mean_conf = np.bincount(bins, conf, num_bins) / np.maximum(n, 1)
    acc = np.bincount(bins, correct, num_bins) / np.maximum(n, 1)
    return n, mean_conf, acc, np.sum(n * np.abs(acc - mean_conf)) / len(conf)

n, mean_conf, acc, ece = reliability(conf, correct)
for b in np.flatnonzero(n):
    print(f'confidence {b / 10:.1f} to {(b + 1) / 10:.1f}: {n[b]:4d} '
          f'predictions, mean {mean_conf[b]:.3f}, accuracy {acc[b]:.3f}')
print(f'ECE: {ece:.3f}')
```

```
confidence 0.1 to 0.2:    2 predictions, mean 0.178, accuracy 0.000
confidence 0.2 to 0.3:   39 predictions, mean 0.266, accuracy 0.231
confidence 0.3 to 0.4:  268 predictions, mean 0.359, accuracy 0.343
confidence 0.4 to 0.5:  655 predictions, mean 0.457, accuracy 0.516
confidence 0.5 to 0.6:  956 predictions, mean 0.548, accuracy 0.548
confidence 0.6 to 0.7:  953 predictions, mean 0.650, accuracy 0.656
confidence 0.7 to 0.8: 1025 predictions, mean 0.752, accuracy 0.792
confidence 0.8 to 0.9: 1372 predictions, mean 0.854, accuracy 0.883
confidence 0.9 to 1.0: 4730 predictions, mean 0.972, accuracy 0.976
ECE: 0.015
```

Filtering to well-populated bins with at least 100 predictions (`used = n >= 100`) so noisy small-sample ratios do not distort the curve, we plot `acc[used]` against `mean_conf[used]` alongside the ideal $y = x$ diagonal:

```
used = n >= 100  # Fewer predictions give too noisy an accuracy
d2l.plot([mean_conf[used], [0, 1]], [acc[used], [0, 1]], 'mean confidence',
         'accuracy', legend=['softmax regression', 'calibrated'],
         fmts=('o-', 'k:'), figsize=(3.5, 3.5))
```



Across all well-populated bins ($n_b \geq 100$), empirical accuracy tracks mean confidence within eight percentage points, sitting slightly *above* the diagonal (average confidence $\sim 81\%$ vs. accuracy $\sim 83\%$, for an ECE below 0.03). Unlike deep overparameterized networks, which frequently become severely overconfident, shallow softmax regression on Fashion-MNIST is already well calibrated (slightly underconfident).

3.4.2.2 Temperature Scaling

When a reliability diagram reveals miscalibration, *temperature scaling* (C. Guo et al. 2017) offers a one-parameter post-hoc fix: hold the trained weights fixed, divide the logits by a scalar temperature $T > 0$ (Section 3.1), and fit T to minimize cross-entropy on a held-out validation split. Because dividing by $T > 0$ preserves logit ordering, temperature scaling never alters a single class prediction or changes accuracy; it only rescales confidence ($T > 1$ damps overconfidence; $T < 1$ sharpens underconfidence). Let us fit T on the first half of our Fashion-MNIST validation set and check the ECE on the second half:

```
def nll(O, y, T):
    """Mean cross-entropy of softmax(O / T), computed from the logits."""
    Z = O / T
    m = Z.max(axis=1)
    lse = m + np.log(np.exp(Z - m[:, None]).sum(axis=1))
    return np.mean(lse - Z[np.arange(len(y)), y])
```

```
half = len(y) // 2
Ts = np.linspace(0.5, 2, 151)
T = Ts[np.argmin([nll(O[:half], y[:half], t) for t in Ts])]
ece = lambda Z: reliability(softmax(Z).max(axis=1), correct[half:])[3]
print(f'T = {T:.2f}; ECE on the second half: {ece(O[half:]):.3f} before, '
      f'{ece(O[half:] / T):.3f} after scaling')
```

```
T = 0.93; ECE on the second half: 0.015 before, 0.012 after scaling
```

Consistent with our slightly underconfident reliability curve, the fitted temperature sits just below 1 ($T \approx 0.9$), leaving the already tiny ECE essentially unchanged.

3.4.3 Summary

Raw accuracy is an adequate figure of merit only when classes are balanced and all errors cost the same. When positive examples are rare, true negatives dominate overall accuracy, so we separate positive detection into *recall* $\mathrm{TP}/(\mathrm{TP}+\mathrm{FN})$, an intrinsic property of the classifier) and *precision* $\mathrm{TP}/(\mathrm{TP}+\mathrm{FP})$, which scales with population prevalence via Bayes' theorem). When false negatives and false positives carry unequal costs κ_{FN} and κ_{FP}, expected cost is minimized by predicting positive whenever $p > \kappa_{\mathrm{FP}}/(\kappa_{\mathrm{FP}}+\kappa_{\mathrm{FN}})$, a rule that relies on the predicted probabilities being calibrated. Reliability diagrams and expected calibration error audit that agreement, and fitting a single post-hoc temperature $T > 0$ on held-out data rescales confidence without changing accuracy.

3.4.4 Exercises

1. **Optimal decisions and rejection.** In a multiclass problem, let κ_{ij} be the cost of predicting class i when the true class is j, and let $p(j \mid x)$ be the probability of class j given the input x.

    1. Write the expected cost of predicting i in terms of the costs and $p(j \mid x)$, and state the rule that selects the optimal i. Check that it reduces to Equation 3.20 for two classes.
    2. Let a correct guess cost 0 and a wrong guess cost λ_s, and allow the classifier to *abstain* instead of guessing, at a fixed cost $\lambda_r < \lambda_s$. Derive the threshold on the top-class probability above which guessing beats abstaining.
    3. In medical diagnosis, deferring to a specialist is cheap relative to a misdiagnosis, say $\lambda_r/\lambda_s = 0.05$. Compute the threshold and state what the classifier should do with a prediction at 0.9 confidence. Then describe a case in which reporting the full distribution $p(y \mid x)$ serves the physician better than either guessing or abstaining.

    Adapted from Kevin Murphy, Probabilistic Machine Learning: An Introduction, Exercise 5.1.

2. **Class-weighted loss.** In the screening example of Section 3.4.1, missing a sick patient costs much more than a false alarm. Give each class j a weight $w_j > 0$ and train f on the class-weighted average loss

$$\frac{1}{n}\sum_{i=1}^{n} w_{y^{(i)}}\, l\big(f(\mathbf{x}^{(i)}), y^{(i)}\big), \tag{3.22}$$

where l is the cross-entropy loss and $y^{(i)}$ is the class of example i, so that every example of class j counts w_j times.

1. Let the training examples be drawn with class prior $p(y)$ and class-conditional densities $p(\mathbf{x} \mid y)$. Show that the expected class-weighted loss is proportional to the expected unweighted loss under the distribution with the same class-conditionals and class prior proportional to $w_y\, p(y)$. Name the change to the training data, rather than to the loss, that has the same effect.
2. Fix an input $\mathbf{x}$ with class probabilities $p(j \mid \mathbf{x})$, and suppose the model can output any distribution $\hat{\mathbf{y}}$. Show that the expected weighted loss

$$-\sum_j w_j\, p(j \mid \mathbf{x}) \log \hat{y}_j \tag{3.23}$$

   is minimized by $\hat{y}_j \propto w_j\, p(j \mid \mathbf{x})$.
3. For the two classes healthy and sick, find the threshold on $p(\text{sick} \mid \mathbf{x})$ above which the arg max of this $\hat{\mathbf{y}}$ is sick, and choose the weights so that the threshold equals that of Equation 3.20. Explain why choosing $w_{\text{sick}} > w_{\text{healthy}}$ raises recall, and what it does to the number of false positives.

3. **Calibration and usefulness.** A ten-class problem has the same number of examples in every class.

   1. A classifier ignores its input and outputs the probability 0.1 for every class, breaking ties at random. Show that it is calibrated, and compute its accuracy.
   2. Another classifier is right on 95% of the examples and always reports confidence 1. Draw its reliability diagram and compute its expected calibration error.
   3. Explain why calibration and accuracy measure different things, and why neither can replace the other when a decision uses the threshold Equation 3.20.

4. code **ROC curves.** A binary classifier that outputs a score $s \in (0, 1)$ for the positive class can be thresholded at any $\tau \in [0, 1]$, not just $\frac{1}{2}$: predict positive whenever $s > \tau$. The *receiver operating characteristic* (ROC) curve plots the true-positive rate (recall) against the false-positive rate $\text{FP}/(\text{FP} + \text{TN})$ as τ sweeps from 1 down to 0. Simulate 1000 positive and 1000 negative examples with scores $s = \sigma(z)$, where $\sigma(z) = 1/(1+\exp(-z))$ is the sigmoid, $z \sim \mathcal{N}(\mu, 1)$ for positives, and $z \sim \mathcal{N}(0, 1)$ for negatives.

   1. Compute and plot the ROC curve for $\mu \in \{0, 1, 2\}$, and state which points of the plot the endpoints $\tau = 1$ and $\tau = 0$ give.
   2. Show that a classifier whose scores are independent of the labels traces the diagonal in expectation.
   3. For $\mu = 2$, plot the *precision-recall curve*, precision against recall as τ sweeps, once with equal numbers of positives and negatives and once with positives making up 1% of the examples (keep 1000 negatives and draw 10 positives, or reweight). Explain why the ROC curve does not change with the prevalence while the precision-recall curve drops, and which of the two describes the screening test of this section better.

5. code **AUC as ranking probability.** ● This problem uses the ROC curves and simulated scores of problem 4.

   1. Show that, when no two scores tie, the area under the ROC curve equals the probability that a randomly chosen positive example receives a higher score than

a randomly chosen negative one, which makes it a threshold-free summary of ranking quality.
2. Compute this probability in closed form for the simulated scores and compare it with the trapezoidal area under each of the curves.
3. Apply the ROC curve and the AUC to the trained softmax regression on the validation images of two classes a and b, with a as the positive class and the score $s = \sigma(o_a - o_b)$, the sigmoid of the difference of the two logits: sneaker versus sandal, and shirt versus T-shirt. Compare the ordering of the two AUCs with the entries of the confusion matrix $\mathbf{C}$ of Section 3.3 for the same pairs, and explain why the two diagnostics need not agree.

6. code **Temperature scaling.** The text fits T on the first half of the validation images.
1. Draw the reliability diagrams of the second half before and after scaling, and show that the accuracy does not change.
2. Fit T on the training images instead and compare it with the T of the text. Explain why the fit should use data that the model was not trained on.
3. Compute the expected calibration error of the unscaled model with 5, 10, and 20 bins. How much does the number depend on the binning?

3.5 Generalization in Classification

Our Fashion-MNIST classifier achieved roughly 83% accuracy on $10{,}000$ validation images. How far can we trust that number as a guarantee of performance on future garments? Two fundamental obstacles stand in the way:

1. **Finite-sample uncertainty and test-set reuse:** Even a fresh test set of $n = 10{,}000$ images is only a finite random sample, and worse still, in real development we consult the same evaluation split repeatedly as we compare models and tune hyperparameters.
2. **Memorization vs. learning:** In high dimensions where every training image is distinct, a lookup-table memorizer achieves 100% training accuracy while guessing randomly on every new image. Why does a low training error for a *linear* classifier certify low population error when a memorizer's zero training error certifies nothing?

Where Section 2.6 introduced overfitting and validation splits qualitatively for regression, this section makes generalization quantitative for classification: we derive exact and asymptotic confidence intervals for a test set of size n, quantify how selection bias and adaptive overfitting inflate scores under test-set reuse, and introduce the *Vapnik–Chervonenkis (VC) dimension* and *uniform convergence* from statistical learning theory.

3.5.1 The Test Set

3.5.1.1 Empirical and Population Error

Fix a single classifier f *before* drawing an independent test set $\mathcal{D} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{n}$ i.i.d. from the population distribution $P(X, Y)$. Its *empirical error* on $\mathcal{D}$ and its true *population error* are

$$\epsilon_{\mathcal{D}}(f) = \frac{1}{n} \sum_{i=1}^{n} \mathbf{1}\left(f(\mathbf{x}^{(i)}) \neq y^{(i)}\right), \qquad \epsilon(f) = P\left(f(X) \neq Y\right). \tag{3.24}$$

Because f was fixed in advance, each error indicator $\mathbf{1}(f(\mathbf{x}^{(i)}) \neq y^{(i)})$ is an independent Bernoulli$(\epsilon(f))$ coin flip with mean $\epsilon(f)$ and variance $\epsilon(f)(1-\epsilon(f)) \leq 1/4$. Therefore, the test error $\epsilon_{\mathcal{D}}(f)$ is an unbiased estimator ($\mathbb{E}[\epsilon_{\mathcal{D}}(f)] = \epsilon(f)$) with standard error

$$\mathrm{SE}(\epsilon_{\mathcal{D}}(f)) = \sqrt{\frac{\epsilon(f)(1-\epsilon(f))}{n}} \leq \frac{0.5}{\sqrt{n}}. \tag{3.25}$$

For our Fashion-MNIST classifier ($\epsilon(f) \approx 0.17$) on $n = 10{,}000$ test images, the standard error is $\sqrt{0.17 \times 0.83/10{,}000} \approx 0.0038$ (0.38 percentage points). Notice the $1/\sqrt{n}$ scaling law: halving our measurement uncertainty requires *four times* as many test examples.

3.5.1.2 Confidence Intervals and Hoeffding's Inequality

To convert the standard error into a 95% confidence interval $\epsilon_{\mathcal{D}}(f) \pm r$, we have two tools:

1. **Asymptotic normal interval (Central Limit Theorem):** For large n, $\epsilon_{\mathcal{D}}(f)$ is approximately normal, falling within 1.96 standard errors of $\epsilon(f)$ with probability ≈ 0.95:

$$P\left(|\epsilon_{\mathcal{D}}(f) - \epsilon(f)| \leq 1.96\sqrt{\frac{\epsilon(f)(1-\epsilon(f))}{n}}\right) \approx 0.95. \tag{3.26}$$

   At $\epsilon \approx 0.17$ and $n = 10{,}000$, the 95% normal radius is $1.96 \times 0.0038 \approx 0.007$ (± 0.7 percentage points); in the worst case ($\epsilon = 1/2$), guaranteeing a 95% radius ≤ 0.01 takes $n \geq (1.96 \times 0.5/0.01)^2 = 9{,}604$ examples. This is a useful way to read the 10,000-example test sets of MNIST, Fashion-MNIST, and CIFAR-10: they pin accuracy to about one point.
2. **Finite-sample Hoeffding interval:** Because the normal approximation can undercover badly when $\epsilon(f)$ is very close to 0 (where a small test set may observe zero mistakes and collapse the plug-in normal radius to 0), *Hoeffding's inequality* (Hoeffding 1963) uses only the fact that each error lies in $[0, 1]$ to give an exact bound valid for every n and every distribution (Section 36.9):

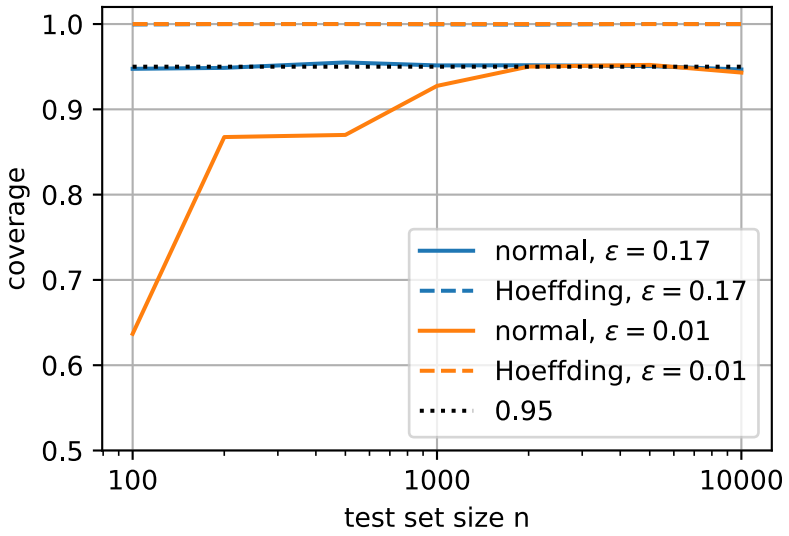
$$P\left(|\epsilon_{\mathcal{D}}(f) - \epsilon(f)| \geq t\right) \leq 2\exp\left(-2nt^2\right). \tag{3.27}$$

   Setting the right-hand side to $\delta = 0.05$ yields the 95% Hoeffding radius $r = \sqrt{\log(2/\delta)/(2n)}$ (requiring $n \geq \log(40)/(2 \times 0.01^2) \approx 18{,}445$ examples for $r \leq 0.01$, roughly twice the normal prescription: both assume the worst-case variance $1/4$, but Hoeffding's exponential tail bound holds for every n and is looser than the exact normal quantile).

Let us simulate 10,000 independent test sets across sizes $n \in [100, 10{,}000]$ for two classifiers (one with $\epsilon = 0.17$ and one with $\epsilon = 0.01$) and plot the empirical coverage of both 95% intervals:

```
rng = np.random.default_rng()
num_sets = 10000
ns = np.array([100, 200, 500, 1000, 2000, 5000, 10000])
curves, legend = [], []
for eps in [0.17, 0.01]:
    # The number of errors on a test set of size n is Binomial(n, eps)
```

```
    errs = [rng.binomial(n, eps, num_sets) / n for n in ns]
    normal = [1.96 * np.sqrt(e * (1 - e) / n) for e, n in zip(errs, ns)]
    hoeffding = [np.sqrt(np.log(2 / 0.05) / (2 * n)) for n in ns]
    for name, radii in [('normal', normal), ('Hoeffding', hoeffding)]:
        curves.append([np.mean(np.abs(e - eps) <= r)  # coverage
                       for e, r in zip(errs, radii)])
        legend.append(rf'{name}, $\epsilon = {eps}$')
d2l.plot(ns, curves + [[0.95] * len(ns)], 'test set size n', 'coverage',
         legend=legend + ['0.95'], xscale='log', ylim=(0.5, 1.02),
         fmts=('C0-', 'C0--', 'C1-', 'C1--', 'k:'), figsize=(4.5, 3))
```



At $\epsilon = 0.17$, the plug-in normal interval stays close to 95% coverage at every n; at $\epsilon = 0.01$ and small n, the normal interval drops to $\sim 63\%$ coverage whenever zero-error test sets occur, whereas Hoeffding's conservative interval exceeds 95% coverage everywhere.

3.5.2 Test Set Reuse

3.5.2.1 Selection Among Many Classifiers

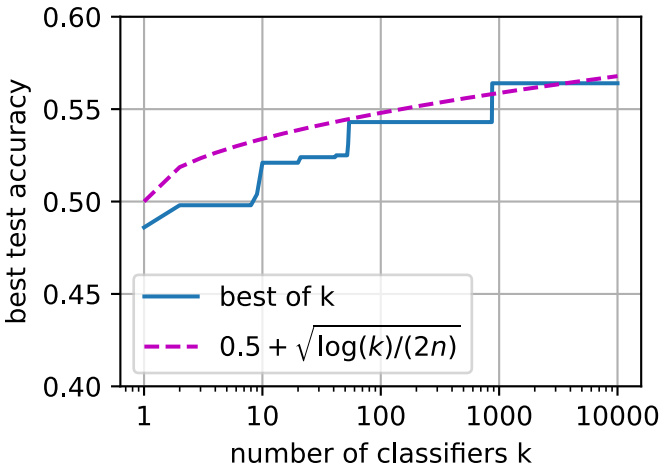
What happens when we evaluate k fixed candidate classifiers $f_1, \ldots, f_k$ on the *same* test set $\mathcal{D}$ and report the winner? Even if all k candidates were fixed before $\mathcal{D}$ was drawn, picking the best score cherry-picks whichever candidate got luckiest on $\mathcal{D}$. Applying the *union bound* (the probability that at least one of k bad events occurs is at most the sum of their individual probabilities) to Equation 3.27 gives

$$P\left(\max_{j=1,\ldots,k} |\epsilon_{\mathcal{D}}(f_j) - \epsilon(f_j)| \geq t\right) \leq 2k \exp\left(-2nt^2\right). \tag{3.28}$$

Setting the right-hand side to δ shows that simultaneous $1-\delta$ accuracy across all k models widens the confidence radius to $\sqrt{\log(2k/\delta)/(2n)}$. Notice the benign $\sqrt{\log k}$ growth: comparing $k = 1{,}000$ fixed models increases the radius by only a logarithmic factor, yet on

$n = 1{,}000$ coin-flip test labels scored by $k = 10{,}000$ purely random coin-flipping guessers (every single one having true accuracy 0.50), the luckiest guesser's test accuracy climbs steadily along $0.5 + \sqrt{\log(k)/(2n)}$ to ~ 0.56:

```
n, k = 1000, 10000
labels = rng.random(n) < 0.5            # one test set with coin-flip labels
guesses = rng.random((k, n)) < 0.5      # k classifiers that guess at random
accuracy = (guesses == labels).mean(axis=1)
ks = np.arange(1, k + 1)
d2l.plot(ks, [np.maximum.accumulate(accuracy),
              0.5 + np.sqrt(np.log(ks) / (2 * n))],
         'number of classifiers k', 'best test accuracy',
         legend=['best of k', r'$0.5 + \sqrt{\log(k) / (2n)}$'], xscale='log',
         ylim=(0.4, 0.6))
```



3.5.2.2 Adaptive Overfitting

Selection bias grows far more dangerous when new candidates are chosen *adaptively* based on previous test scores (Dwork et al. 2015). Once test feedback guides model design, the classifiers are no longer independent of $\mathcal{D}$ and the union bound Equation 3.28 breaks down completely. Following Blum and Hardt (2015), watch what happens if we take just the first $1{,}000$ random coin-flipping guessers above, select only those that happened to score above 0.50 on the test set, and take their majority vote on each test example:

```
kept = guesses[:1000][accuracy[:1000] > 0.5]  # guessers that scored above 0.5
vote = kept.mean(axis=0) > 0.5                # their majority vote
print(f'majority vote: {(vote == labels).mean():.3f}, '
      f'best single guesser: {accuracy[:1000].max():.3f}')
```

```
majority vote: 0.709, best single guesser: 0.564
```

Without ever seeing an individual test label, using only $1{,}000$ scalar accuracy queries, the majority vote of pure coin flippers jumps to $\sim$ 70% test accuracy while still having 50% population accuracy on any new data. (In fact, $O(n)$ queries can reconstruct every test label;

see Exercise 3.) This is why hyperparameter tuning and architecture exploration belong on a validation split (Exercise 4), leaving the test set for one-shot final evaluation.

How much damage has a decade of reuse done to the standard benchmarks? When Recht et al. (2019) collected fresh test sets for CIFAR-10 and ImageNet, accuracies fell by 3 to 15 points, yet the ranking of the models barely changed. The worst case above is possible, but ordinary model development had overfit these test sets far less than the theory permits. The drop itself is better explained by a small change in the data distribution, which is the subject of Section 3.6.

3.5.3 Statistical Learning Theory

3.5.3.1 Uniform Convergence

How can we guarantee *a priori* that a classifier $f_{\mathcal{S}}$ selected from a hypothesis class $\mathcal{F}$ by minimizing training error $\epsilon_{\mathcal{S}}(f)$ on a training set $\mathcal{S}$ of size n will generalize to the population? Since the learner picks $f_{\mathcal{S}} \in \mathcal{F}$ after inspecting $\mathcal{S}$, we need the training error to stay close to the population error *simultaneously for every* $f \in \mathcal{F}$ (*uniform convergence*). If $\mathcal{F}$ is finite ($|\mathcal{F}| < \infty$), applying the union bound Equation 3.28 across all $k = |\mathcal{F}|$ hypotheses on $\mathcal{S}$ yields, with probability at least $1 - \delta$,

$$|\epsilon_{\mathcal{S}}(f) - \epsilon(f)| \le \sqrt{\frac{\log(2|\mathcal{F}|/\delta)}{2n}} \quad \text{for every } f \in \mathcal{F}. \tag{3.29}$$

3.5.3.2 The VC Dimension and the VC Bound

For continuous parameter families like linear halfspaces $f(\mathbf{x}) = \operatorname{sign}(\mathbf{w}^\top \mathbf{x} + b)$ on $\mathbb{R}^d$, $|\mathcal{F}| = \infty$, making Equation 3.29 vacuous. Yet Vapnik and Chervonenkis (1971) observed that on any finite set of n points, two classifiers that produce the exact same binary ± 1 labeling are indistinguishable on $\mathcal{S}$. We say that $\mathcal{F}$ *shatters* a set of m points if functions in $\mathcal{F}$ can realize all 2^m possible binary labelings of those m points. The *Vapnik–Chervonenkis (VC) dimension* $v = \mathrm{VC}(\mathcal{F})$ is the size of the largest set of points that $\mathcal{F}$ can shatter:

- **Lines in the plane ($\mathbb{R}^2$) have VC dimension 3 (Figure 3.6):** Any three non-collinear points can be shattered in all $2^3 = 8$ ways by a straight line, whereas no four points in $\mathbb{R}^2$ can ever be shattered (splitting opposite corners of a convex quadrilateral requires crossing diagonals, and a halfplane containing the three vertices of a triangle automatically contains any point inside it).
- **Linear halfspaces in $\mathbb{R}^d$ have VC dimension $d+1$ (Exercise 5):** They shatter the origin and the d standard basis vectors, and by Radon's theorem no $d + 2$ points in $\mathbb{R}^d$ can be shattered.
- **Unconstrained lookup-table memorizers have $v = \infty$:** They shatter every finite set of distinct points.

By *Sauer's lemma*, once $n \ge v$, the number of distinct labelings that a class of VC dimension v can induce on n points grows only polynomially—at most $(en/v)^v$ rather than 2^n. Running a symmetrized union bound over those labelings (Section 36.10) yields the *VC bound*: for an absolute constant c, with probability at least $1 - \delta$ over the training draw $\mathcal{S}$,

$$\epsilon(f) \le \epsilon_{\mathcal{S}}(f) + c\sqrt{\frac{v + \log(1/\delta)}{n}} \quad \text{for every } f \in \mathcal{F}. \tag{3.30}$$

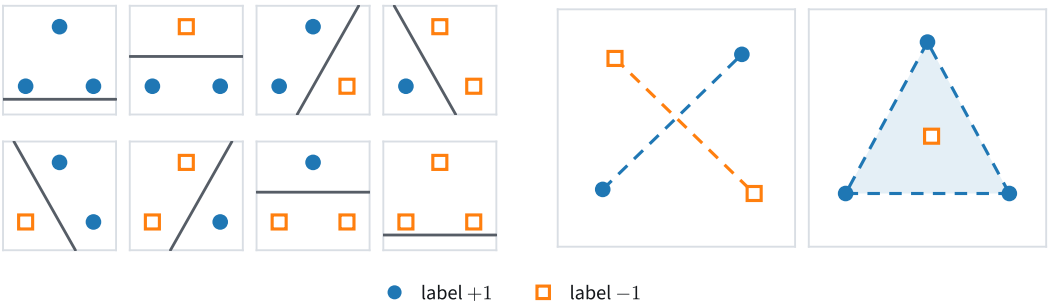


Figure 3.6: Lines in $\mathbb{R}^2$ shatter three non-collinear points (left, all $2^3 = 8$ labelings) but cannot shatter any four points (middle and right).

This theorem answers our conceptual puzzle: because linear classifiers in $\mathbb{R}^d$ have finite VC dimension $v = d + 1$, driving training error low when $n \gg d$ guarantees low population error for *any* distribution P, whereas an infinite-VC memorizer enjoys no bound at all. At the same time, because worst-case uniform convergence bounds must hold even for adversarial distributions, Equation 3.30 is numerically loose on natural images (for a single one-vs-rest halfspace $\sqrt{785/60{,}000} \approx 0.11$ before the constant c, and the capacity of all ten classes together is larger still, versus our empirical 0.01 train–test gap on Fashion-MNIST) and becomes vacuous ($v \geq n$) for overparameterized deep neural networks capable of memorizing random labels (Zhang et al. 2021). We return to that puzzle in Section 4.5 while relying on held-out test sets for sharp empirical measurement.

3.5.4 Summary

On an untouched test set of n examples, a fixed classifier's empirical error $\epsilon_{\mathcal{D}}(f)$ has standard error $\sqrt{\epsilon(1-\epsilon)/n} \leq 0.5/\sqrt{n}$, so pinning accuracy to within one percentage point at 95% confidence requires roughly $10{,}000$ test examples under the normal approximation or $18{,}500$ under Hoeffding's finite-sample bound. Reusing the same test set to select among k fixed models widens that uncertainty slowly as $\sqrt{\log(2k/\delta)/(2n)}$, whereas adaptive design guided by past test scores can overfit a test set in far fewer queries. To bound generalization from training error alone, statistical learning theory measures hypothesis-class capacity by the VC dimension v—the largest set of points the class can shatter ($v = d+1$ for linear halfspaces in $\mathbb{R}^d$)—yielding a uniform $O(\sqrt{v/n})$ bound that explains why linear models cannot memorize large datasets while remaining vacuous for overparameterized deep networks.

3.5.5 Exercises

1. **Two models, one test set.** A paper reports that model A reaches 91.2% and model B 91.6% on a test set of 10,000 images.
   1. Compute the standard error of each accuracy and the 95% radius of the normal approximation. Is the difference of 0.4 points meaningful if the two models were evaluated on two independent test sets of this size?
   2. Both models were in fact evaluated on the same 10,000 images. Explain why the question is now about the images on which the two models disagree, and describe how you would decide it if the paper reported, for every test image, which models

got it right.
   3. How large a test set would make a 0.4-point difference between two independently evaluated models lie outside the radius?
2. **No errors on the test set.** A classifier makes no error on a test set of n examples.
   1. State the interval that the normal approximation, with $\epsilon_{\mathcal{D}}(f)$ in place of $\epsilon(f)$, reports, and the Hoeffding interval.
   2. Show that a classifier with population error ϵ makes no error on n independent test examples with probability $(1-\epsilon)^n \le e^{-n\epsilon}$. Conclude that if $\epsilon \ge 3/n$, a test set without errors occurs with probability below 0.05 (the "rule of three").
   3. Compare $3/n$ with the Hoeffding radius for $n = 100$ and $n = 10{,}000$, and explain the difference.
3. **Leaking a test set.** Somebody else holds a test set $\mathcal{D}$ of n examples with binary labels and releases only the inputs. You may submit any model f, with no restriction on the model class, and you receive only its test error $\epsilon_{\mathcal{D}}(f)$.
   1. Give a strategy that recovers every test label, and hence a model with test error 0 whatever its population error, and count the submissions it uses.
   2. Each answer $\epsilon_{\mathcal{D}}(f)$ takes one of $n+1$ values. Use this to lower-bound the number of submissions that any strategy needs to distinguish all 2^n labelings.
4. code **A validation split for Fashion-MNIST.** The chapter selected models on the 10,000 test images. Hold out 10,000 of the 60,000 training images as a validation split instead, and train softmax regression on the remaining 50,000.
   1. Choose the learning rate from $\{0.01, 0.03, 0.1, 0.3, 1\}$ by its accuracy on the validation split.
   2. Evaluate the selected model once on the test images, and compare its test accuracy with its validation accuracy. Is the difference within the radius of the 95% interval for 10,000 examples?
   3. Explain why the test accuracy, measured once after the choice, is an unbiased estimate for the selected model, while the best validation accuracy is not.
5. **VC dimension of linear classifiers.** ● Consider the linear classifiers $f(\mathbf{x}) = \text{sign}(\mathbf{w}^\top \mathbf{x} + b)$ on $\mathbb{R}^d$.
   1. Show that they shatter the $d+1$ points $\mathbf{0}, \mathbf{e}_1, \dots, \mathbf{e}_d$. Given arbitrary labels $y_0, y_1, \dots, y_d \in \{\pm 1\}$, construct b and $\mathbf{w}$ that produce them. Hint: $f(\mathbf{0})$ depends on b alone, so choose b first.
   2. Let $\mathbf{x}_1, \dots, \mathbf{x}_{d+2} \in \mathbb{R}^d$. Show that the $d+2$ vectors $[\mathbf{x}_i, 1] \in \mathbb{R}^{d+1}$, each $\mathbf{x}_i$ with a 1 appended, are linearly dependent, so that $\sum_i a_i [\mathbf{x}_i, 1] = \mathbf{0}$ for some a_i that are not all zero. Let I be the indices with $a_i > 0$ and J those with $a_i < 0$. Show that neither is empty and that no linear classifier labels the points in I with $+1$ and those in J with -1. Hint: evaluate $\sum_i a_i (\mathbf{w}^\top \mathbf{x}_i + b)$ in two ways.
   3. Conclude that the VC dimension is $d+1$, and show that the convex hulls of $\{\mathbf{x}_i : i \in I\}$ and $\{\mathbf{x}_j : j \in J\}$ intersect.
6. **VC dimension of polynomials.** What is the VC dimension of the class of classifiers $f(x) = \text{sign}(p(x))$ with p a polynomial of degree at most five in $x \in \mathbb{R}$? What is it for $\mathbf{x} \in \mathbb{R}^d$? Hint: such a classifier is a linear classifier on the monomials of $\mathbf{x}$. Then say what the VC bound predicts for the training error of a degree-five polynomial classifier on Fashion-MNIST, and why nobody trains one.

3.6 Environment and Distribution Shift

Every guarantee in Chapter 3.5 rested on one quiet assumption: that deployment data are drawn from the exact same distribution as our training and test sets. In the real world, that assumption breaks constantly. Scanners and cameras get upgraded, seasonal flu waves alter disease prevalence overnight, regional slang evolves, and worse yet, deploying a model can *itself* cause the population to change its behavior. Suppose a bank discovers in historical records that loan applicants wearing Oxfords repay more often than applicants in sneakers, and deploys a classifier that approves Oxfords and rejects sneakers; within weeks, every applicant shows up in Oxfords while the underlying default rate never budges—a textbook example of *Goodhart's law* (when a statistical correlate becomes a decision target, it ceases to be a valid correlate).

In this section, we study how to detect and correct distribution shift between a *source distribution* $p_S(\mathbf{x}, y)$ (where we have labeled training data) and a *target distribution* $p_T(\mathbf{x}, y)$ (where we observe only unlabeled deployment inputs $\mathbf{x}$). We organize the three canonical types of passive shift by which factor of the joint distribution remains invariant, derive and implement *importance-weighted covariate shift correction* and *confusion-matrix label shift correction*, and close with concept shift and feedback loops.

3.6.1 Types of Distribution Shift

Without *some* structural assumption connecting $p_S(\mathbf{x}, y)$ and $p_T(\mathbf{x}, y)$, adapting to an unlabeled target domain is mathematically impossible: if $p_T(\mathbf{x}) = p_S(\mathbf{x})$ stays identical while the labels of cats and dogs secretly swap overnight ($p_T(y = \text{cat} \mid \mathbf{x}) = p_S(y = \text{dog} \mid \mathbf{x})$), no algorithm looking only at unlabeled target images can ever detect the swap.

Every joint distribution factors in two ways:

$$p(\mathbf{x}, y) = p(\mathbf{x})\, p(y \mid \mathbf{x}) = p(y)\, p(\mathbf{x} \mid y). \tag{3.31}$$

Holding one conditional factor invariant while the marginal drifts defines the three classic regimes of distribution shift (Figure 3.7):

Shift regime	What changes	What stays fixed	Natural causal setting
Covariate shift	$p(\mathbf{x})$	$p(y \mid \mathbf{x})$	features $\mathbf{x}$ cause label y
Label shift	$p(y)$	$p(\mathbf{x} \mid y)$	label y causes features $\mathbf{x}$
Concept shift	$p(y \mid \mathbf{x})$	often $p(\mathbf{x})$	label definition drifts over time or space

3.6.1.1 Covariate Shift, Label Shift, and Concept Shift

1. **Covariate shift ($p(y \mid \mathbf{x})$ invariant):** Natural whenever $\mathbf{x}$ causes y. Suppose we train a cat-versus-dog classifier on real photographs (Figure 3.8) and deploy it on cartoons (Figure 3.9). A cartoon cat is still a cat ($p(y \mid \mathbf{x})$ is unchanged), but the marginal input distribution $p(\mathbf{x})$ has shifted into a new region of pixel space. Even though the true Bayes boundary has not moved, a misspecified or capacity-limited model fitted to source photos will focus its accuracy where $p_S(\mathbf{x})$ is dense and can fail badly where $p_T(\mathbf{x})$ lives.

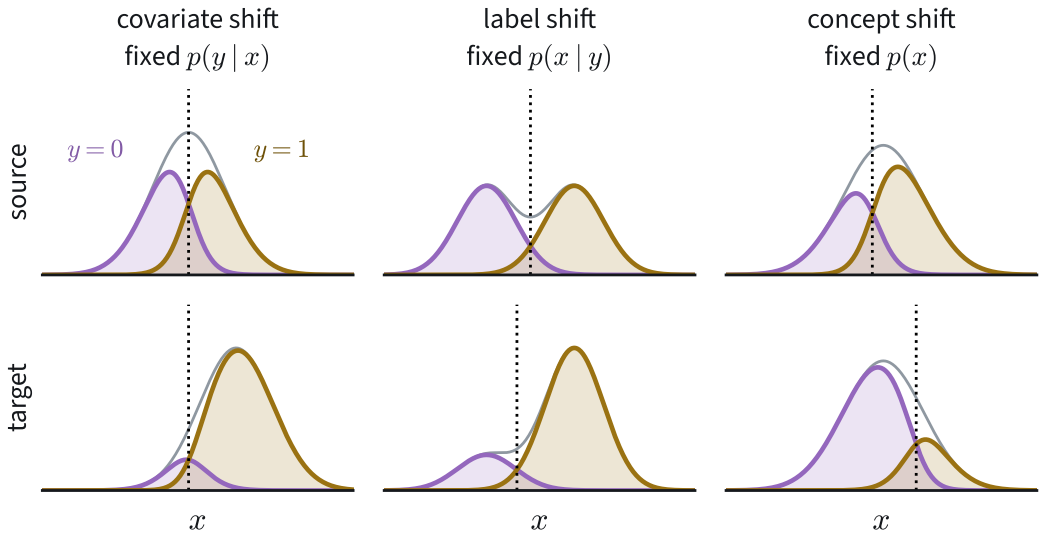


Figure 3.7: Three regimes of distribution shift for a 1D input x and two classes (top: source p_S; bottom: target p_T; dotted line: optimal decision boundary). Covariate shift moves $p(x)$ while leaving $p(y \mid x)$ and the true boundary fixed; label shift changes the class priors $p(y)$ while keeping $p(x \mid y)$ fixed (shifting the Bayes boundary toward the rarer class); concept shift alters $p(y \mid x)$ directly.

2. **Label shift ($p(\mathbf{x} \mid y)$ invariant):** Natural whenever y causes $\mathbf{x}$: for instance, diseases cause symptoms, so during a flu epidemic the prevalence $p(y)$ of influenza surges while the symptom distribution $p(\mathbf{x} \mid y)$ of a flu patient stays unchanged. By Bayes' theorem, $p(y \mid \mathbf{x}) \propto p(\mathbf{x} \mid y)\, p(y)$, so a shift in class priors shifts the optimal decision boundary. Correcting label shift requires estimating only q class-prior ratios rather than a high-dimensional density ratio over $\mathbf{x}$.
3. **Concept shift ($p(y \mid \mathbf{x})$ changes):** The meaning of the label itself changes across geography or time, whether in regional names for soft drinks across U.S. counties (Figure 3.10), evolving diagnostic criteria, or seasonal shopping intent.

3.6.2 Covariate Shift Correction

3.6.2.1 Importance Weights

At deployment we want to minimize the *target risk* $R_T[f] = \mathbb{E}_{(\mathbf{x},y)\sim p_T}[l(f(\mathbf{x}), y)]$. Under covariate shift ($p_T(y \mid \mathbf{x}) = p_S(y \mid \mathbf{x}) = p(y \mid \mathbf{x})$), whenever the source support covers the target ($p_S(\mathbf{x}) > 0$ wherever $p_T(\mathbf{x}) > 0$), multiplying and dividing by $p_S(\mathbf{x})$ rewrites the target risk as an expectation over the *source* distribution (Shimodaira 2000):

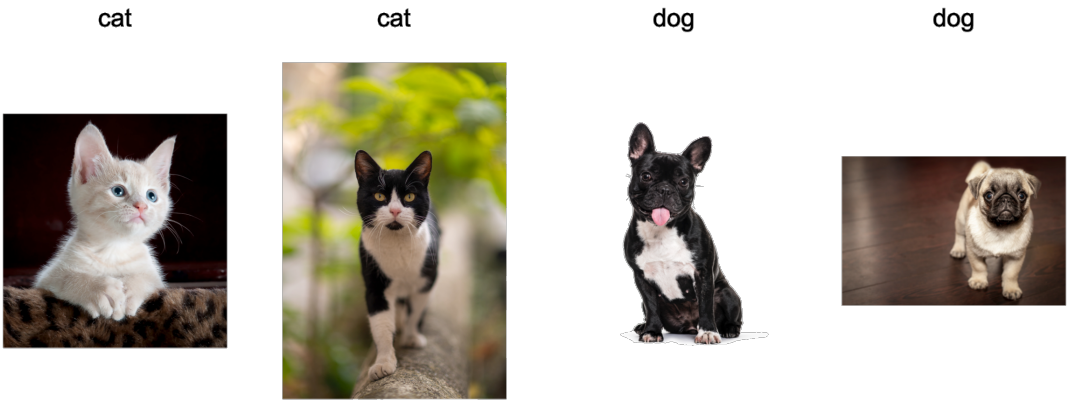


Figure 3.8: Source training images for cat vs. dog classification: photographs (illustrations: Lafeez Hossain / 500px / Getty Images; ilkermetinkursova / iStock / Getty Images Plus; GlobalP / iStock / Getty Images Plus; Musthafa Aboobakuru / 500px / Getty Images).

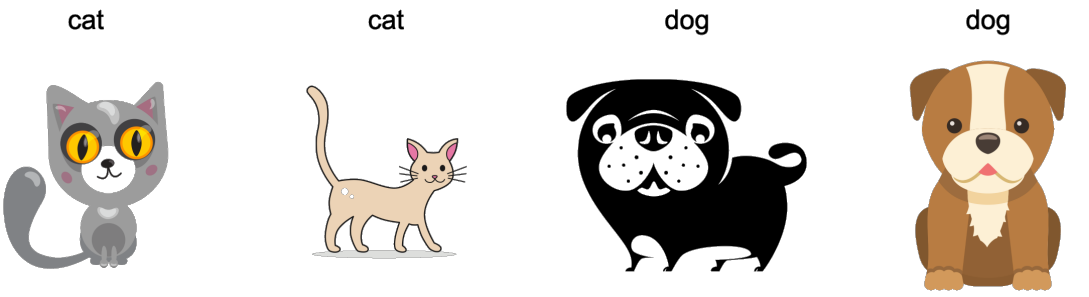


Figure 3.9: Target deployment images for cat vs. dog classification: cartoons (illustrations: SIBAS_minich / iStock / Getty Images Plus; Ghrzuzudu / iStock / Getty Images Plus; id-work / DigitalVision Vectors / Getty Images; Yime / iStock / Getty Images Plus).

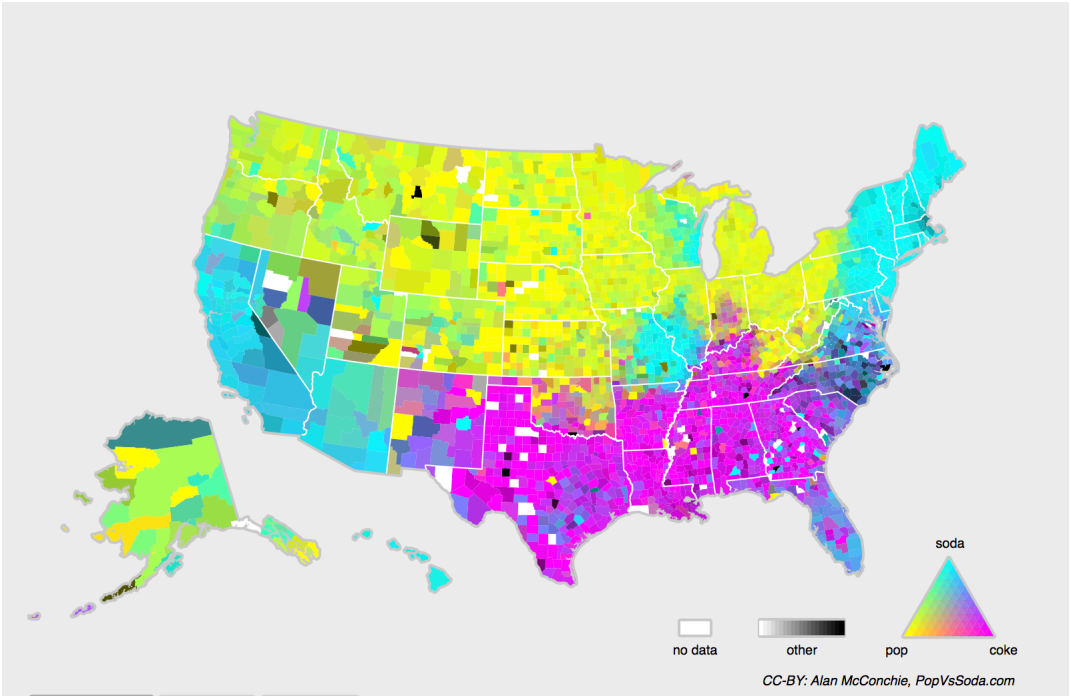


Figure 3.10: Geographic concept shift: the most common word for a sweetened carbonated beverage (“pop”, “soda”, or “coke”) across U.S. counties (CC-BY: Alan McConchie, PopVsSoda.com).

$$\begin{aligned} R_T[f] &= \sum_y \int l(f(\mathbf{x}), y)\, p(y \mid \mathbf{x})\, p_T(\mathbf{x})\, d\mathbf{x} \\ &= \sum_y \int l(f(\mathbf{x}), y)\, p(y \mid \mathbf{x})\, p_S(\mathbf{x})\, \frac{p_T(\mathbf{x})}{p_S(\mathbf{x})}\, d\mathbf{x} \\ &= \mathbb{E}_{(\mathbf{x},y)\sim p_S}\left[\beta(\mathbf{x})\, l(f(\mathbf{x}), y)\right], \end{aligned} \tag{3.32}$$

where $\beta(\mathbf{x}) = p_T(\mathbf{x})/p_S(\mathbf{x})$ is the *importance weight* of input $\mathbf{x}$. Approximating Equation 3.32 on our n labeled source examples gives *weighted empirical risk minimization*:

$$\begin{aligned} &\underset{f}{\text{minimize}}\ \frac{1}{n}\sum_{i=1}^{n} \beta_i\, l(f(\mathbf{x}^{(i)}), y^{(i)}), \\ &\text{where } \beta_i = \frac{p_T(\mathbf{x}^{(i)})}{p_S(\mathbf{x}^{(i)})}. \end{aligned} \tag{3.33}$$

Notice why *support overlap* ($p_S(\mathbf{x}) > 0$ wherever $p_T(\mathbf{x}) > 0$) is non-negotiable: if a medical startup collects diseased blood samples from elderly patients and healthy control samples exclusively from 20-year-old university students, no reweighting can ever manufacture healthy elderly controls where $p_S(\mathbf{x}) = 0$. Even when support overlaps, extreme right-tail ratios (Figure 3.11) can cause a handful of points to dominate the sum, collapsing the *effective sample size* $n_{\text{eff}} = (\sum_i \beta_i)^2 / \sum_i \beta_i^2$; clipping weights at a cap $\min(\beta_i, c)$ trades a sliver of bias for a much larger n_{eff}.

3.6.2.2 Estimating Weights with a Domain Classifier

How can we estimate $\beta(\mathbf{x}) = p_T(\mathbf{x})/p_S(\mathbf{x})$ without estimating high-dimensional densities p_S and p_T separately? Pool the n source inputs (assigned domain label $z = 0$) with m unlabeled target inputs (assigned domain label $z = 1$) and train a binary logistic regression *domain classifier* $P(z = 1 \mid \mathbf{x}) = \sigma(h(\mathbf{x}))$ with logit $h(\mathbf{x})$. By Bayes' theorem,

$$\begin{aligned} \frac{P(z=1 \mid \mathbf{x})}{P(z=0 \mid \mathbf{x})} &= \frac{m}{n}\,\frac{p_T(\mathbf{x})}{p_S(\mathbf{x})} = \exp(h(\mathbf{x})), \\ \text{so}\quad \beta(\mathbf{x}) &= \frac{n}{m}\exp(h(\mathbf{x})). \end{aligned}$$

Dropping the constant factor n/m (which scales the objective without moving its minimizer), we simply weight each source example by $\beta_i = \exp(h(\mathbf{x}^{(i)}))$ (or $\min(\exp(h(\mathbf{x}^{(i)})), c)$).

The domain classifier has a second use: it tells us whether there is any shift to correct. If $p_T(\mathbf{x}) = p_S(\mathbf{x})$, nothing separates source inputs from target inputs, and on pooled inputs that it was not trained on the domain classifier stays at chance ($1/2$ when $n = m$). An accuracy clearly above chance is therefore evidence that the inputs have shifted. This check is known as a *classifier two-sample test* (Lopez-Paz and Oquab 2017). Label shift has an even cheaper check: run the deployed classifier on the unlabeled target inputs and compare how often it predicts each class with the same frequencies on source validation data. A deployed system should watch both numbers, since a shift has to be noticed before it can be corrected.

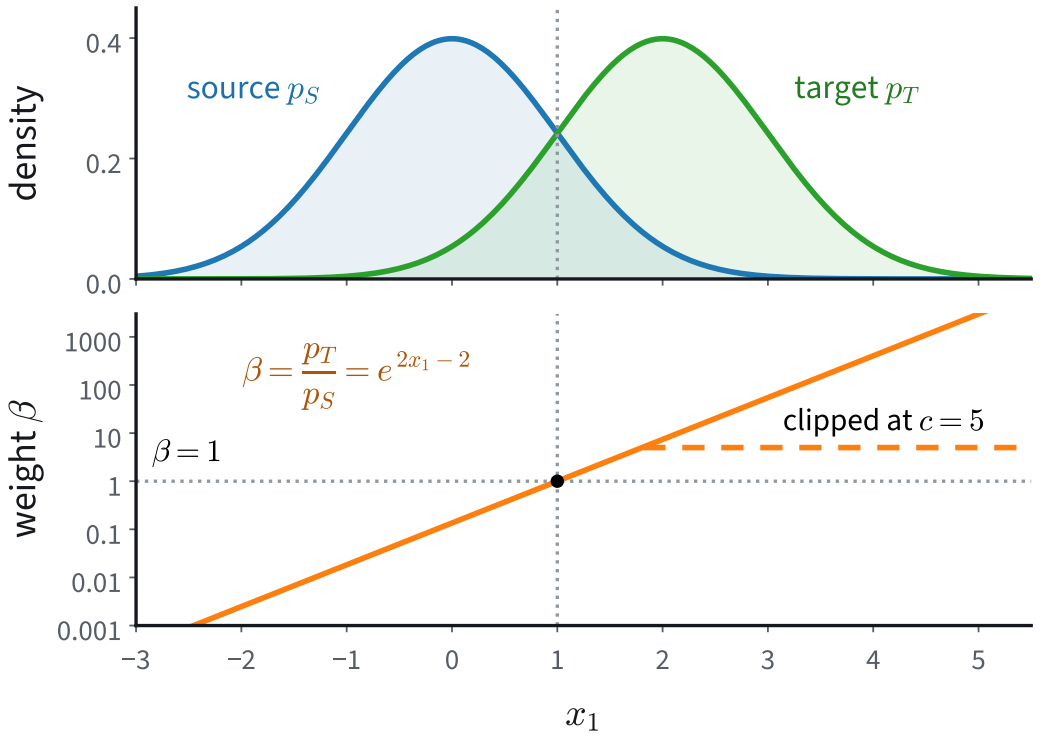


Figure 3.11: Source density $p_S = \mathcal{N}(0, 1)$, shifted target density $p_T = \mathcal{N}(2, 1)$ (top), and the resulting importance weight $\beta(x_1) = p_T(x_1)/p_S(x_1) = e^{2x_1 - 2}$ on a log scale with clipping threshold $c = 5$ (bottom).

3.6.2.3 A Two-Dimensional Example

Let us watch domain-classifier reweighting rescue a linear classifier on a 2D problem where $p_S = \mathcal{N}(\mathbf{0}, \mathbf{I}_2)$, $p_T = \mathcal{N}((2,0)^\top, \mathbf{I}_2)$, and the true label is $y = \mathbf{1}(x_2 > \frac{1}{2}x_1^2 - 1)$. Because the true parabolic boundary curves upward, the best linear boundary near $(0,0)$ is nearly horizontal and fails miserably near $(2,0)$. We generate $n = 1{,}000$ source and $1{,}000$ target points, fit the logistic domain classifier $h(\mathbf{x})$ to estimate $\beta_i = \exp(h(\mathbf{x}^{(i)}))$, and compare unweighted, importance-weighted, and clipped ($c = 5$) logistic regression:

```
rng = np.random.default_rng()
n = 1000
X_src = rng.normal(0, 1, (n, 2))            # source p_S: centered at (0, 0)
X_tgt = rng.normal(0, 1, (n, 2)) + [2, 0]   # target p_T: centered at (2, 0)
label = lambda X: (X[:, 1] > 0.5 * X[:, 0]**2 - 1).astype(float)
y_src, y_tgt = label(X_src), label(X_tgt)   # one labeling rule for both
```

Our sole learning primitive is weighted logistic regression trained by gradient descent. By Section 3.1.2.2, the cross-entropy gradient with respect to the logit is the residual $\sigma(\mathbf{w}^\top \mathbf{x} + b) - y$, so weighting example i by β_i simply multiplies its residual by β_i.

```
def fit_logreg(X, y, weights=None, lr=0.1, steps=2000):
    """Logistic regression by gradient descent on the weighted mean loss."""
    w, b = np.zeros(X.shape[1]), 0.0
    v = np.ones(len(y)) if weights is None else weights / weights.mean()
    for _ in range(steps):
        r = v * (1 / (1 + np.exp(-(X @ w + b))) - y)   # weighted error
        w -= lr * X.T @ r / len(y)
        b -= lr * r.mean()
    return w, b
```

First, we pool `X_src` (domain label $z = 0$) and `X_tgt` ($z = 1$) to train our domain classifier `(w_h, b_h)` via `fit_logreg`, compare the learned logit $h(\mathbf{x})$ against the analytical Gaussian log-ratio $2x_1 - 2$, and inspect the resulting source importance weights `beta` and their effective sample size `ess`. The last lines draw fresh source and target inputs and measure how often the domain classifier tells them apart:

```
z = np.concatenate([np.zeros(n), np.ones(n)])      # domain label: 1 = target
w_h, b_h = fit_logreg(np.concatenate([X_src, X_tgt]), z)
beta = np.exp(X_src @ w_h + b_h)                   # weights of source examples
ess = lambda v: v.sum()**2 / (v**2).sum()          # effective sample size
print(f'learned h(x) = {w_h[0]:.2f} x1 {w_h[1]:+.2f} x2 {b_h:+.2f}'
      f'   (true log-ratio: 2 x1 - 2)')
print(f'largest weight: {beta.max():.0f}; effective sample size: '
      f'{ess(beta):.0f} of {n}, {ess(np.minimum(beta, 5)):.0f} if clipped at 5')
X_out = np.concatenate([rng.normal(0, 1, (n, 2)),          # fresh source and
                        rng.normal(0, 1, (n, 2)) + [2, 0]])  # target inputs
print(f'domain classifier accuracy on held-out inputs: '
      f'{((X_out @ w_h + b_h > 0) == (z == 1)).mean():.2f} (chance: 0.50)')
```

```
learned h(x) = 1.98 x1 +0.09 x2 -1.98   (true log-ratio: 2 x1 - 2)
largest weight: 193; effective sample size: 23 of 1000, 221 if clipped at 5
```

```
domain classifier accuracy on held-out inputs: 0.84 (chance: 0.50)
```

On fresh inputs the domain classifier separates source from target with an accuracy of about 0.84, against 0.50 by chance, so this shift is easy to detect. No classifier can do better here: the two Gaussians differ only in the mean of x_1, the best rule is $x_1 > 1$, and it is right with probability $\Phi(1) \approx 0.84$, where Φ is the standard normal distribution function.

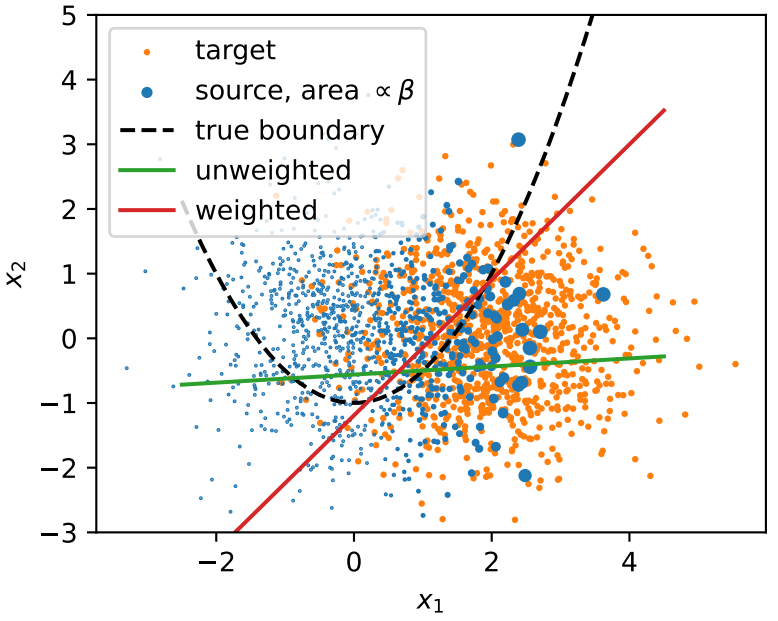
Next, we train `fit_logreg` on the labeled source data `(X_src, y_src)` under three weighting schemes (`'unweighted'`, raw importance weights `'weighted'`, and clipped weights `'clipped'` at 5) and evaluate each classifier's accuracy on `X_tgt` and `X_src`:

```
acc = lambda wb, X, y: ((X @ wb[0] + wb[1] > 0) == (y == 1)).mean()
models = {name: fit_logreg(X_src, y_src, v) for name, v in [
    ('unweighted', None), ('weighted', beta), ('clipped', np.minimum(beta, 5))]}
for name, wb in models.items():
    print(f'{name:10s}  target accuracy {acc(wb, X_tgt, y_tgt):.3f}'
          f'   source accuracy {acc(wb, X_src, y_src):.3f}')
print(f'always predicting the most common target class: '
      f'{max(y_tgt.mean(), 1 - y_tgt.mean()):.3f}')
```

```
unweighted  target accuracy 0.603   source accuracy 0.853
weighted    target accuracy 0.945   source accuracy 0.854
clipped     target accuracy 0.944   source accuracy 0.854
always predicting the most common target class: 0.711
```

To visualize why reweighting helps so dramatically, we plot the target and source point clouds (scaling each source marker's area by `beta`) together with the true parabolic curve $x_2 = \frac{1}{2}x_1^2 - 1$ and the `'unweighted'` and `'weighted'` linear decision boundaries:

```
x1 = np.linspace(-2.5, 4.5, 50)
d2l.set_figsize((4.5, 3.5))
d2l.plt.scatter(*X_tgt.T, s=2, c='C1', label='target')
d2l.plt.scatter(*X_src.T, s=np.clip(beta, 0.2, 20), c='C0',
                label=r'source, area $\propto \beta$')
d2l.plt.plot(x1, 0.5 * x1**2 - 1, 'k--', label='true boundary')
for name, fmt in [('unweighted', 'C2'), ('weighted', 'C3')]:
    (w1, w2), b = models[name]
    d2l.plt.plot(x1, -(w1 * x1 + b) / w2, fmt, label=name)
d2l.plt.xlabel('$x_1$'), d2l.plt.ylabel('$x_2$'), d2l.plt.ylim(-3, 5)
d2l.plt.legend();
```



Look at the result: on the shifted target cloud, unweighted training scores only $\sim 50\%$–60% (worse than always guessing the majority class at $\sim 70\%$), whereas importance reweighting tilts the linear decision boundary along the right arm of the parabola and boosts target accuracy to $\sim 95\%$ using *zero* target labels.

It would be wrong to conclude that we should always reweight under covariate shift. Reweighting helped here because a line cannot follow a parabola: the model is misspecified and has to choose where to be accurate. A model flexible enough to fit $p(y \mid \mathbf{x})$ wherever the source has data gains nothing, since the weighted and the unweighted objective then share the same minimizer (Shimodaira 2000). Deep networks are close to this regime. Once they are trained to a low training loss, the effect of importance weights fades (Byrd and Lipton 2019), and on the WILDS collection of real-world shifts, methods designed for distribution shift rarely beat plain empirical risk minimization (Koh et al. 2021). In practice, covariate shift is handled mostly by training on broader data and by fine-tuning on a few labeled target examples.

3.6.3 Label Shift Correction

Under label shift ($p_T(\mathbf{x} \mid y) = p_S(\mathbf{x} \mid y)$ while class proportions $p_T(y) \neq p_S(y)$ shift), the exact same importance-weighting identity weights each source example of class y by the class-prior ratio $\beta(y) = p_T(y)/p_S(y)$. Since $p_S(y)$ is known from the training labels, the only challenge is estimating the target class proportions $\mathbf{p}_T \in \mathbb{R}^q$ from *unlabeled* target data.

3.6.3.1 Estimating Target Proportions via the Confusion Matrix

Let $\mathbf{C} \in \mathbb{R}^{q \times q}$ be our source classifier's column-normalized confusion matrix from Section 3.3, with entries $C_{ij} = P_S(\hat{c} = i \mid y = j)$ estimated on labeled source validation data. Because $\hat{c}$ is a function of $\mathbf{x}$ alone and $p(\mathbf{x} \mid y)$ is invariant under label shift,

$P_T(\hat{c} = i \mid y = j) = C_{ij}$ holds on the target domain as well. By the law of total probability, the vector $\mu \in \mathbb{R}^q$ of predicted class frequencies on the unlabeled target inputs satisfies (Saerens et al. 2002; Lipton et al. 2018)

$$\mu_i = \sum_{j=1}^{q} P(\hat{c} = i \mid y = j)\, p_T(y = j) = \sum_{j=1}^{q} C_{ij}\, p_T(y = j), \quad \text{or} \quad \mathbf{C}\,\mathbf{p}_T = \mu. \tag{3.34}$$

Whenever $\mathbf{C}$ is invertible (which is guaranteed if the classifier gets more than half of each class right, $C_{jj} > 1/2$; see Exercise 3), solving the $q \times q$ linear system $\hat{\mathbf{p}}_T = \mathbf{C}^{-1}\mu$ recovers the unknown target class proportions without a single target label. Furthermore, if the source classifier's probabilities approximate the source posterior $p_S(y \mid \mathbf{x})$ (a stronger requirement than the top-class calibration of Section 3.4), Bayes' rule adapts its predictions to the target domain immediately, without retraining, by adding the log-prior shift $\log(\hat{p}_T(y)/p_S(y))$ to each class's logit.

Let us test confusion-matrix inversion and Bayes logit adjustment on Fashion-MNIST. We first train `d2l.SoftmaxRegression` on the balanced training split and collect its validation logits `O`, labels `y`, and predicted classes `pred = O.argmax(axis=1)`:

```
data = d2l.FashionMNIST(batch_size=256)
model = d2l.SoftmaxRegression(num_outputs=10, lr=0.1)
trainer = d2l.Trainer(max_epochs=10)
trainer.board.display = False
trainer.fit(model, data)
O, y = [], []
for X, Y in data.val_dataloader():
    O.append(d2l.numpy(model(X)))
    y.append(d2l.numpy(Y))
O, y = np.concatenate(O), np.concatenate(y)
pred = O.argmax(axis=1)
```

Next, we randomly partition the $10{,}000$ validation examples into a $5{,}000$-image source validation set `src` (used to estimate the column-normalized 10×10 confusion matrix `C`) and a target pool `pool` from which we draw a skewed target set `tgt` where 50% of all garments are shirts (400 shirts and 45 of each other class), then solve `p_hat = np.linalg.solve(C, mu)` from the unlabeled target prediction frequencies `mu`:

```
perm = rng.permutation(len(y))
src, pool = perm[:5000], perm[5000:]  # Source validation set; target pool
C = np.zeros((10, 10))
np.add.at(C, (pred[src], y[src]), 1)  # The confusion matrix of the source
C /= C.sum(axis=0, keepdims=True)     # Each column sums to 1
counts = np.full(10, 45)
counts[6] = 400                       # Half of the target images are shirts
p_T = counts / counts.sum()
tgt = np.concatenate([rng.choice(pool[y[pool] == j], counts[j], replace=False)
                      for j in range(10)])
mu = np.bincount(pred[tgt], minlength=10) / len(tgt)  # Needs no labels
p_hat = np.linalg.solve(C, mu)
```

```
for j, name in enumerate(data.text_labels(range(10))):
    print(f'{name:10s} true {p_T[j]:.2f}, predicted {mu[j]:.3f}, '
          f'estimated {p_hat[j]:.3f}')
```

```
T-shirt    true 0.06, predicted 0.133, estimated 0.067
trouser    true 0.06, predicted 0.055, estimated 0.055
pullover   true 0.06, predicted 0.133, estimated 0.091
dress      true 0.06, predicted 0.070, estimated 0.051
coat       true 0.06, predicted 0.082, estimated 0.050
sandal     true 0.06, predicted 0.052, estimated 0.054
shirt      true 0.50, predicted 0.294, estimated 0.468
sneaker    true 0.06, predicted 0.058, estimated 0.055
bag        true 0.06, predicted 0.066, estimated 0.054
ankle boot true 0.06, predicted 0.057, estimated 0.057
```

Finally, we adapt the classifier's predictions to the estimated target distribution by adding the log-prior shift `log_w = np.log(np.maximum(p_hat, 1e-6) / 0.1)` to the target logits `O[tgt]` and comparing target accuracy before and after adjustment:

```
log_w = np.log(np.maximum(p_hat, 1e-6) / 0.1)  # log p_T(y) - log p_S(y)
accuracy = lambda O: (O.argmax(axis=1) == y[tgt]).mean()
print(f'target accuracy: {accuracy(O[tgt]):.3f} as trained, '
      f'{accuracy(O[tgt] + log_w):.3f} adapted to the estimated proportions')
```

```
target accuracy: 0.706 as trained, 0.816 adapted to the estimated proportions
```

Even though the raw classifier predicts "shirt" on only $\sim$ 20%–40% of the target images (spilling the rest into look-alike tops), $\mathbf{C}^{-1}\mu$ unscrambles that confusion to recover a shirt proportion near the true 0.50 (typically within a few hundredths, occasionally off by almost a tenth), and adding $\log(\hat{\mathbf{p}}_T/0.1)$ to the logits lifts target accuracy by roughly 5 to 15 percentage points with zero target labels and zero gradient steps.

3.6.4 Concept Shift and Feedback

When *concept shift* alters $p(y \mid \mathbf{x})$ itself, historical labels are outdated and no importance reweighting of old data can teach the new labeling rule; continuous monitoring on a trickle of freshly labeled examples and periodic fine-tuning or retraining are essential.

Even more care is needed when a model's predictions trigger *actions that reshape future training data* (*performative prediction* and *feedback loops*). In predictive policing (Figure 3.12), dispatching extra patrols to neighborhoods with high predicted crime causes more incidents to be recorded there; feeding those biased arrest records back into the next training round reinforces the forecast in a runaway loop. Breaking or auditing such loops requires randomized exploration or independent data collection, crossing from passive supervised learning into *reinforcement learning* (Chapter 14).

3.6.5 Summary

Adapting to an unlabeled deployment environment is impossible without a structural invariance connecting source $p_S(\mathbf{x}, y)$ and target $p_T(\mathbf{x}, y)$, factored either as covariate shift ($p(y \mid \mathbf{x})$ fixed while $p(\mathbf{x})$ drifts), label shift ($p(\mathbf{x} \mid y)$ fixed while $p(y)$ drifts), or concept shift ($p(y \mid \mathbf{x})$ drifts). Under covariate shift with overlapping support, weighting source losses

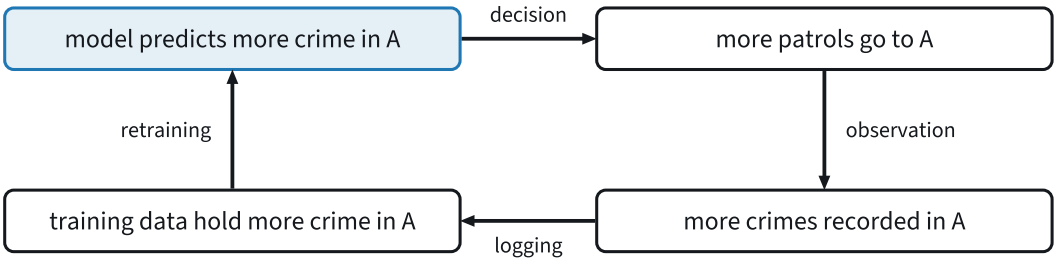


Figure 3.12: A runaway feedback loop in predictive policing: model predictions steer patrol allocation, patrol presence determines where incidents are observed, and logged incidents become the next training dataset.

by $\beta(\mathbf{x}) = p_T(\mathbf{x})/p_S(\mathbf{x}) \propto \exp(h(\mathbf{x}))$—estimated by a logistic domain classifier separating source from unlabeled target inputs—aligns misspecified models with the target risk. Under label shift, inverting the source confusion matrix via $\hat{\mathbf{p}}_T = \mathbf{C}^{-1}\mu$ recovers target class proportions from unlabeled target predictions and adapts calibrated logits directly by adding $\log(\hat{\mathbf{p}}_T/\mathbf{p}_S)$. When concept shift or decision-driven feedback loops alter the environment itself, passive reweighting is not enough: deployed systems need fresh target labels and independent exploration.

3.6.6 Exercises

1. **Missing support.** Suppose that $p_S(\mathbf{x}) = 0$ on a region $\mathcal{A}$ of inputs that has probability $\pi > 0$ under the target, and that $p_S(\mathbf{x}) > 0$ everywhere else. Define $\beta(\mathbf{x}) = p_T(\mathbf{x})/p_S(\mathbf{x})$ outside $\mathcal{A}$.
    1. Show that the weighted source risk $\mathbb{E}_{(\mathbf{x},y)\sim p_S}[\beta(\mathbf{x})\, l(f(\mathbf{x}), y)]$ equals the target risk with the contribution of $\mathcal{A}$ removed. Write the missing term as π times a conditional expectation.
    2. Explain why no choice of weights on source examples can recover the missing term. Construct two models that agree outside $\mathcal{A}$ and have the same weighted source risk but different target risks.
2. code **When reweighting fails.** The code of Section 3.6.2 (`fit_logreg`, `X_src`, `X_tgt`, `label`) places the target mean at $(2, 0)$. Sweep the target mean over $(s, 0)$ for $s \in \{0, 0.5, 1, 2, 3, 4, 6\}$, keeping $n = 1000$ and the labeling rule fixed.
    1. For each s, report the accuracy of the domain classifier h on held-out pooled data, and interpret its growth with s in terms of how detectable the shift is.
    2. For each s, record the target accuracy of the unweighted, weighted, and clipped ($c = 5$) models, and of always predicting the most common target class. Explain why target accuracy alone cannot show whether reweighting worked at large s.
3. **The label shift system.** Let $\mathbf{C}$ be the column-normalized confusion matrix of Equation 3.34.
    1. Show that $\mathbf{C}$ is invertible whenever $C_{jj} > \frac{1}{2}$ for every class j. Hint: suppose that $\mathbf{v}^\top \mathbf{C} = \mathbf{0}$ for some $\mathbf{v} \neq \mathbf{0}$, look at the column j for which $|v_j|$ is largest, and use that the column sums to 1.

2. Give an example of a classifier for which $\mathbf{C}$ is singular, and describe what μ can still reveal about $\mathbf{p}_T$ in that case.
3. For two classes, show that an error δ in the fraction μ_1 of positive predictions changes the estimated fraction of positives by $\delta/(C_{11}-C_{10})$. Using the standard error $\sqrt{\mu_1(1-\mu_1)/m}$ of a fraction estimated from m target inputs, find how many target inputs the screening example needs for a standard error of 0.01 in the estimated fraction of sick patients. Repeat for a weaker test with $C_{11}=0.6$ and $C_{10}=0.4$ on the same population.

4. code **Learned versus true weights.** ● Reuse the sweep of exercise 2 over target means $(s,0)$.
   1. Derive the true log-ratio $\log(p_T(\mathbf{x})/p_S(\mathbf{x}))$ as a function of s, and compare it with the learned h at each s. Explain how the domain classifier can reach high accuracy at large s while h is far from the true log-ratio.
   2. Show that under the source the true weights satisfy $\mathbb{E}[\beta(\mathbf{x})]=1$ and $\mathbb{E}[\beta(\mathbf{x})^2]=e^{s^2}$, so that for large n their effective sample size is about ne^{-s^2}. Evaluate this for $n=1000$ and $s=1,2,3$.
   3. For each s, compute the effective sample size of the true weights and of the learned weights on the actual source sample. Explain why at $s=2$ the sample value is usually much larger than $1000\,e^{-4}\approx 18$, and what this means for judging the reliability of a reweighted fit from the sample alone. Does clipping at $c=5$ repair the shortage?

5. **Shift assumptions from unlabeled data.** ● In a binary source domain, $p_S(y=1)=p_S(y=0)=\frac{1}{2}$, and the scalar input satisfies $x\mid y=1\sim\mathcal{N}(1,1)$ and $x\mid y=0\sim\mathcal{N}(-1,1)$. Hence $p_S(y=1\mid x)=1/(1+\exp(-2x))$. Unlabeled target inputs follow the mixture

$$p_T(x)=0.8\,\mathcal{N}(x;1,1)+0.2\,\mathcal{N}(x;-1,1).$$

   1. Assume label shift. Give $p_T(y=1)$ and the threshold on x of the Bayes-optimal target classifier.
   2. Assume instead covariate shift with the same $p_T(x)$. Compute $p_T(y=1)$ numerically and give the Bayes-optimal threshold.
   3. Use your results to explain why no statistic of the unlabeled target sample can decide which of the two corrections of this section applies.

Chapter 4

Multilayer Perceptrons

In this chapter, we introduce your first *deep* neural networks. In Chapter 3, softmax regression assigned a fixed weight to each of the 784 pixels of a Fashion-MNIST garment image and summed the evidence, so every pixel cast the same vote toward a class regardless of what surrounded it. Yet a bright pixel might belong to a sleeve, a handbag strap, or a collar, and only its neighbors can tell which. Even on four binary points, no single linear layer can compute the exclusive-OR (XOR) of two bits, the limitation Marvin Minsky and Seymour Papert highlighted in 1969 for Frank Rosenblatt's single-layer *perceptron*.

Multilayer perceptrons (MLPs) overcome this linear barrier by stacking one or more fully connected *hidden layers*—each pairing an affine transformation with a nonlinear activation function—before the output layer, and training all layers jointly via *backpropagation* (Rumelhart et al. 1986). We begin by showing why linear models fail on feature interactions like XOR, how hidden layers fold input space into exponentially many linear regions with depth, and how common activation functions (ReLU, GELU, sigmoid, and tanh) behave under differentiation (Section 4.1). Implementing a 256-unit ReLU MLP from scratch and concisely on Fashion-MNIST (Section 4.2) eliminates roughly a fifth of softmax regression's classification errors ($\sim 87\%$ vs. $\sim 84\%$ accuracy) at nearly the same GPU epoch time, replacing blurred whole-garment templates with localized contrast detectors.

Adding hidden layers also raises three foundational questions of deep learning that occupy the rest of the chapter. First, Section 4.3 derives the three matrix-calculus rules of reverse-mode differentiation through an MLP, checks them against automatic differentiation to machine precision, and accounts for training's activation-memory and FLOP budgets. Second, Section 4.4 tracks signal and gradient variance across K layers, derives *Xavier* and *He* initialization to prevent vanishing and exploding gradients in a 10-layer MLP, and shows why random weights are needed to break hidden-unit symmetry. Third, Section 4.5 and Section 4.6 examine why heavily overparameterized networks generalize—covering random-label memorization, double descent, the implicit bias of gradient descent, early stopping, and multiplicative noise regularization via *dropout*—before Section 4.7 puts the full toolkit to work on Kaggle's Ames Housing regression benchmark.

4.1 Multilayer Perceptrons

In Chapter 3, softmax regression scored each garment category by taking an affine weighted sum of the input pixels, $o_j = \sum_{i=1}^{d} x_i w_{ij} + b_j$. That linear form bakes in two severe restrictions:

1. **Monotonicity:** Increasing feature x_i always pushes o_j in the same direction (up if $w_{ij} > 0$, down if $w_{ij} < 0$). Yet many real relationships are non-monotonic: human health risk rises whenever body temperature strays from $37°$C in *either* direction.
2. **No feature interactions:** Increasing x_i by one unit changes o_j by w_{ij} regardless of the values of every other coordinate. In an image, however, a bright pixel at $(13, 17)$ means completely different things depending on whether the neighboring pixels form a collar, a sleeve, or a handbag strap.

Figure 4.1 plots the temperature example. The best linear score rises with temperature throughout, so it rates a patient with hypothermia as the safest of all.

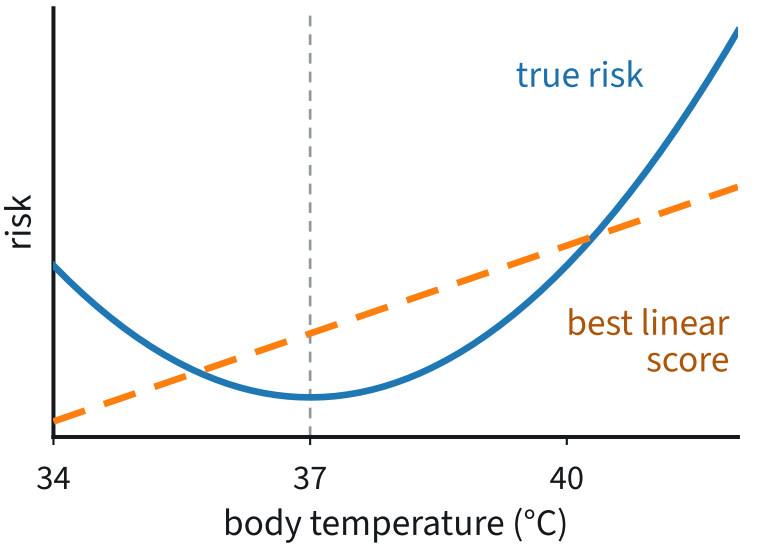


Figure 4.1: Health risk against body temperature. The risk is lowest near $37°$C and rises on both sides, whereas a linear score can only rise or fall. The dashed line is the least-squares linear fit over the plotted range.

A *multilayer perceptron* (MLP) overcomes both limitations by inserting one or more *hidden layers* that transform raw inputs into learned intermediate representations, applying an affine map followed by a nonlinear *activation function* before a linear output layer makes the final prediction. In this section, we prove that no linear model can solve the four-point XOR problem, solve XOR explicitly with two ReLU hidden units, examine what functions shallow and deep networks can represent, and compare the activation functions used across modern deep learning.

4.1.1 Limits of Linear Models: The XOR Problem

The starkest possible demonstration of feature interaction uses just four data points: the corners of the unit square $\{0,1\}^2$ labeled by their *exclusive-OR* (XOR), where $y = 1$ if and only if *exactly one* coordinate equals 1:

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

Notice the interaction: flipping x_1 from 0 to 1 increases y when $x_2 = 0$, but decreases y when $x_2 = 1$.

Proposition. For all $w_1, w_2, b \in \mathbb{R}$, the linear decision rule "predict class 1 if $w_1 x_1 + w_2 x_2 + b > 0$" misclassifies at least one of the four XOR points.

Proof. Suppose for contradiction that some w_1, w_2, b classify all four corners correctly. The two class-1 points $(0,1)$ and $(1,0)$ require $w_2 + b > 0$ and $w_1 + b > 0$, whose sum gives $w_1 + w_2 + 2b > 0$. Meanwhile, the two class-0 points $(0,0)$ and $(1,1)$ require $b \le 0$ and $w_1 + w_2 + b \le 0$, whose sum gives $w_1 + w_2 + 2b \le 0$, an immediate contradiction. ■

Geometrically, both diagonals of the unit square share the midpoint $(1/2, 1/2)$ (Figure 4.2), where any affine function equals the average of its values at either diagonal's endpoints, so no straight line can ever separate the two diagonals.

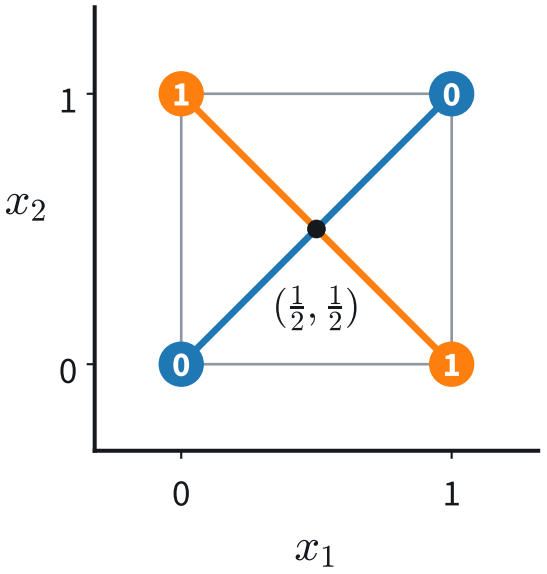


Figure 4.2: The four XOR points, labeled by class. Each diagonal joins the two points of one class, and the two diagonals cross at $(1/2, 1/2)$, so no straight line separates the classes.

4.1.2 Hidden Layers

4.1.2.1 The Model

To escape linearity, an MLP inserts one or more *hidden layers* between the input features and the output layer (Figure 4.3). For a minibatch design matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$ (n examples, d input features), a single hidden layer of width h holds weights $\mathbf{W}^{(1)} \in \mathbb{R}^{d \times h}$ and biases $\mathbf{b}^{(1)} \in \mathbb{R}^{h}$, while the output layer holds weights $\mathbf{W}^{(2)} \in \mathbb{R}^{h \times q}$ and biases $\mathbf{b}^{(2)} \in \mathbb{R}^{q}$. The two-layer (one-hidden-layer) MLP computes

$$\begin{aligned} \mathbf{H} &= \phi\left(\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}\right), \\ \mathbf{O} &= \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}, \end{aligned} \tag{4.1}$$

where biases broadcast across rows and the nonlinear *activation function* ϕ acts elementwise on the *pre-activations* $\mathbf{Z} = \mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}$ to produce the hidden *activations* $\mathbf{H} \in \mathbb{R}^{n \times h}$. Notice the parameter count: $(d+1)h + (h+1)q$. On Fashion-MNIST ($d = 784, q = 10$), a hidden layer of $h = 256$ units has $785 \times 256 + 257 \times 10 = 203{,}530$ parameters, 26 times softmax regression's $7{,}850$.

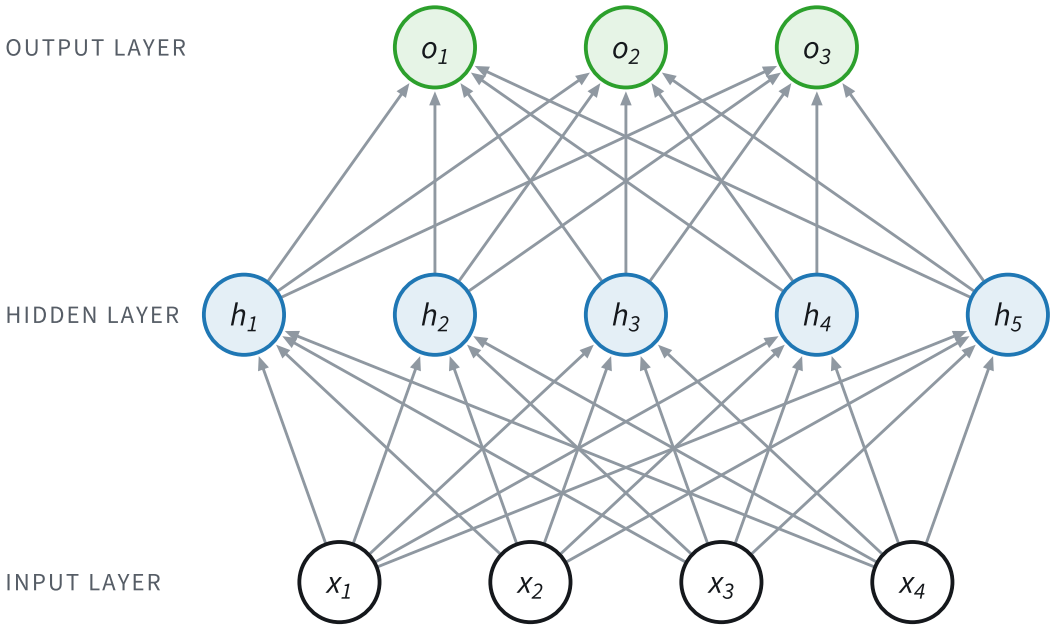


Figure 4.3: An MLP with 4 inputs, 1 hidden layer of 5 hidden units, and 3 outputs. Both parameterized layers are fully connected.

4.1.2.2 The Role of the Nonlinearity

Why is the nonlinear activation ϕ essential? Suppose we omitted ϕ (setting $\phi(z) = z$), so $\mathbf{H} = \mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}$. Substituting $\mathbf{H}$ into the output equation collapses the two layers back into a single affine transformation:

$$\mathbf{O} = \left(\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}\right)\mathbf{W}^{(2)} + \mathbf{b}^{(2)} = \mathbf{X}\underbrace{\mathbf{W}^{(1)}\mathbf{W}^{(2)}}_{\mathbf{W}} + \underbrace{\mathbf{b}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}}_{\mathbf{b}}.$$

Without a nonlinearity, stacking any number of fully connected layers never expands the family of functions beyond a single linear layer (and if a hidden layer is narrower than $\min(d, q)$, it actually restricts $\mathbf{W}$ to low rank; see Exercise 1). Inserting an elementwise nonlinearity such as the *rectified linear unit* (ReLU),

$$\mathrm{ReLU}(z) = \max(z, 0), \tag{4.2}$$

prevents this algebraic collapse.

4.1.2.3 Solving XOR with Two ReLU Units

With ReLU activations, a hidden layer of just $h = 2$ units solves XOR cleanly (Goodfellow et al. 2016). Set

$$\mathbf{W}^{(1)} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}, \quad \mathbf{b}^{(1)} = (0, -1), \quad \mathbf{w}^{(2)} = (1, -2), \quad b^{(2)} = 0. \tag{4.3}$$

Both hidden units depend on the coordinate sum $s = x_1 + x_2$: unit 1 computes $h_1 = \mathrm{ReLU}(s) = s$, while unit 2 computes $h_2 = \mathrm{ReLU}(s - 1)$, turning on *only* when $s = 2$ (at $(1, 1)$) so that $o = h_1 - 2h_2$ subtracts 2 and folds the output back down to 0:

```
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
W1, b1 = np.array([[1, 1], [1, 1]]), np.array([0, -1])
w2, b2 = np.array([1, -2]), 0
H = np.maximum(X @ W1 + b1, 0)  # hidden layer with ReLU
o = H @ w2 + b2  # output layer
for x, h, out in zip(X, H, o):
    print(f'x = {x}, h = {h}, o = {out}')
```

```
x = [0 0], h = [0 0], o = 0
x = [0 1], h = [1 0], o = 1
x = [1 0], h = [1 0], o = 1
x = [1 1], h = [2 1], o = 0
```

Look at the geometry in Figure 4.4: the hidden layer maps *both* class-1 corners $(0, 1)$ and $(1, 0)$ onto the exact same hidden point $(1, 0)$, allowing the straight line $h_1 - 2h_2 = 1/2$ in hidden space (right) to carve out the non-halfspace diagonal band $1/2 < x_1 + x_2 < 3/2$ in input space (left).

4.1.2.4 Deeper Networks

More generally, a K-layer MLP stacks $K - 1$ hidden layers above $\mathbf{H}^{(0)} = \mathbf{X}$,

$$\begin{aligned} \mathbf{H}^{(k)} &= \phi\left(\mathbf{H}^{(k-1)}\mathbf{W}^{(k)} + \mathbf{b}^{(k)}\right) \quad (k = 1, \ldots, K-1), \\ \mathbf{O} &= \mathbf{H}^{(K-1)}\mathbf{W}^{(K)} + \mathbf{b}^{(K)}, \end{aligned}$$

building a compositional hierarchy of representations from raw inputs to class logits.

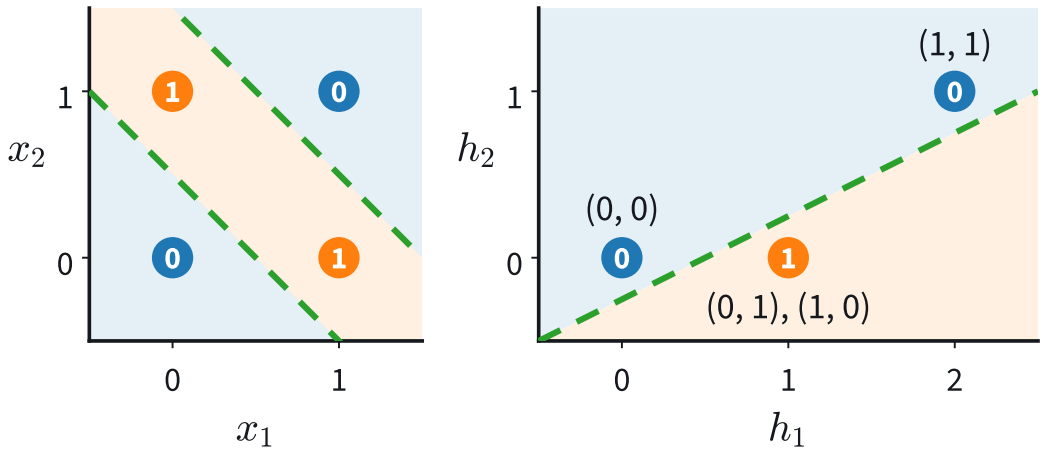


Figure 4.4: Solving XOR with two ReLU hidden units. Left: in input space (x_1, x_2), the network predicts class 1 inside the diagonal band $1/2 < x_1 + x_2 < 3/2$. Right: in hidden space (h_1, h_2), the two class-1 points collapse onto $(1, 0)$, where a single line $h_1 - 2h_2 = 1/2$ separates the classes.

4.1.3 Expressive Power

4.1.3.1 Universal Approximation

How rich is the class of functions computable by a single hidden layer ($K = 2$)? On the real line ($d = 1$), each ReLU hidden unit $a_j \,\mathrm{ReLU}(x - t_j)$ contributes a *hinge* that is flat to the left of its knot t_j and changes slope by a_j to the right (Figure 4.5). Summing h hinges plus a constant produces any continuous piecewise-linear polyline with h interior knots; since any continuous function on a closed bounded interval $[a, b]$ can be approximated uniformly to within any $\epsilon > 0$ by a fine enough polyline, a single wide enough ReLU hidden layer approximates any continuous 1D function to arbitrary precision.

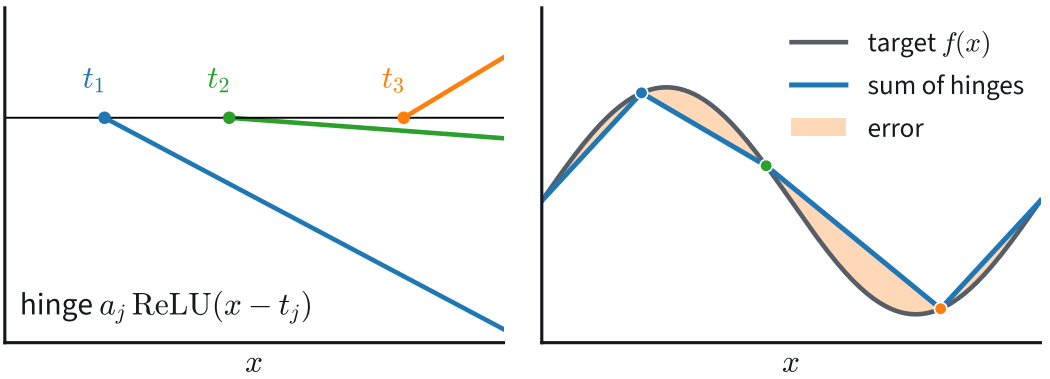


Figure 4.5: Approximating a smooth curve on an interval by a sum of ReLU hinges $a_j \,\mathrm{ReLU}(x - t_j)$, one hidden unit per slope change.

The *universal approximation theorem* generalizes this fact to arbitrary input dimension d: a

single-hidden-layer MLP can approximate *any* continuous function on any compact subset of $\mathbb{R}^d$ to arbitrary uniform accuracy using *any* continuous non-polynomial activation function ϕ (Cybenko 1989; Leshno et al. 1993) (see Exercise 5 for a 2D proof via ridge functions and the Stone–Weierstrass theorem).

4.1.3.2 Depth and Space Folding

If one hidden layer can already approximate anything, why build *deep* networks? Because universal approximation says nothing about *how many units* a single layer needs (which can grow exponentially in d), nor whether gradient descent can learn or generalize from finite data.

Depth achieves exponential parameter efficiency by *composing* space-folding maps (Telgarsky 2016). Consider the 1D tent function on $[0, 1]$ computed by two ReLU units:

$$g(z) = 2\,\mathrm{ReLU}(z) - 4\,\mathrm{ReLU}(z - 1/2), \tag{4.4}$$

which rises from 0 to 1 on $[0, 1/2]$ and falls back to 0 on $[1/2, 1]$ (Figure 4.6). Composing g with itself folds $[0, 1]$ in half at every layer: $g \circ g$ has 4 linear pieces, $g \circ g \circ g$ has 8, and the m-fold composition $g^{\circ m}$, computed by m hidden layers of 2 ReLU units ($2m$ total units), produces a sawtooth wave with 2^m linear pieces. Representing that same 2^m-piece sawtooth with a *single* hidden layer requires at least $2^m - 1$ hidden units ($1{,}023$ units in one layer vs. 20 units across 10 layers when $m = 10$).

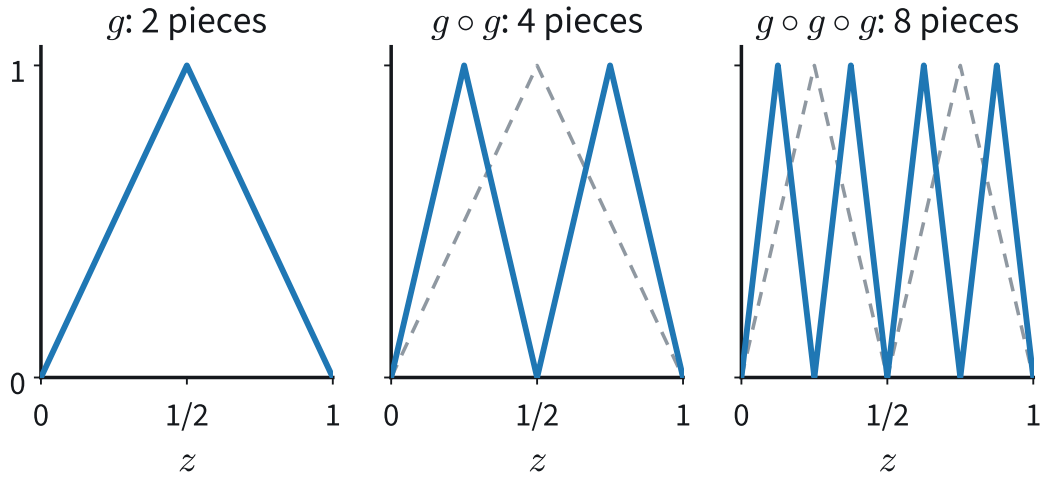


Figure 4.6: Composing the two-ReLU tent function g with itself m times doubles the number of linear segments at every layer, producing 2^m segments with only $2m$ hidden units.

4.1.4 Activation Functions

While almost any non-polynomial ϕ suffices for universal approximation, optimization via gradient descent depends critically on the derivative $\phi'(z)$. For hidden unit j with pre-activation $z_j = \sum_i x_i w_{ij}^{(1)} + b_j^{(1)}$ and activation $h_j = \phi(z_j)$, the chain rule scales every incoming weight gradient by the local slope $\phi'(z_j)$:

$$\frac{\partial l}{\partial w_{ij}^{(1)}} = \frac{\partial l}{\partial h_j}\,\phi'(z_j)\,x_i. \tag{4.5}$$

Wherever $\phi'(z_j) \approx 0$, gradient flow stalls. Figure 4.7 compares the four principal activations and their derivatives.

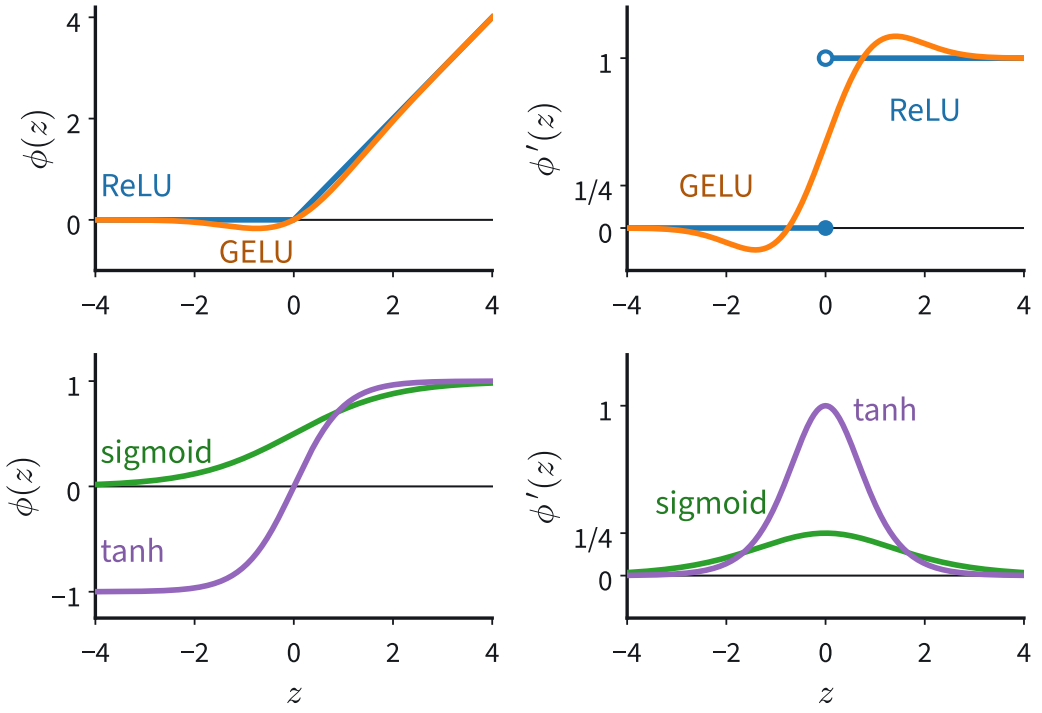


Figure 4.7: Four activation functions $\phi(z)$ (left) and their derivatives $\phi'(z)$ (right): ReLU and GELU (top), sigmoid and tanh (bottom).

- **ReLU and Leaky ReLU:** $\mathrm{ReLU}(z) = \max(z, 0)$ has derivative 1 for $z > 0$ and 0 for $z < 0$ (conventionally 0 at $z = 0$). Because its slope is identically 1 across the entire positive half-line, active ReLU units never attenuate gradients as z grows (Nair and Hinton 2010; Glorot et al. 2011). If a unit's weights shift so that $z < 0$ on every training input, however, its gradient vanishes permanently (a *dead ReLU*); *leaky ReLU*, $\max(z, \alpha z)$ with $\alpha \approx 0.01$, keeps a small positive slope for $z < 0$ so dead units can recover.
- **GELU and SiLU (Swish):** The *Gaussian error linear unit* $\mathrm{GELU}(z) = z\,\Phi(z)$ (where Φ is the standard normal CDF) and its cousin $\mathrm{SiLU}(z) = z\,\sigma(z)$ smooth out ReLU's kink at the origin (passing through $z = 0$ with slope $1/2$ and a gentle negative dip near $z \approx -0.75$) and serve as the default activations in modern Transformers (Hendrycks and Gimpel 2016).
- **Sigmoid and Tanh:** The logistic sigmoid $\sigma(z) = 1/(1 + e^{-z}) \in (0, 1)$ has derivative $\sigma'(z) = \sigma(z)(1 - \sigma(z)) \le 1/4$, while the zero-centered hyperbolic tangent $\tanh(z) = (1 - e^{-2z})/(1 + e^{-2z}) = 2\sigma(2z) - 1 \in (-1, 1)$ (Exercise 3) has derivative $1 - \tanh^2(z) \le 1$. Both saturate to flat horizontal asymptotes ($\phi'(z) \to 0$) as $|z|$

grows beyond ~ 3, and multiplying $\sigma'(z) \leq 1/4$ across K layers shrinks gradients by at least 4^{-K} (*vanishing gradients*, analyzed in Section 4.4). Today, sigmoid and tanh are used primarily as bounded *gates* and state squashers in recurrent architectures (Chapter 12).

4.1.5 Summary

Linear models are monotonic in every coordinate and cannot capture feature interactions such as XOR. Multilayer perceptrons overcome both limitations by inserting hidden layers $\mathbf{H}^{(k)} = \phi(\mathbf{H}^{(k-1)}\mathbf{W}^{(k)} + \mathbf{b}^{(k)})$ before the linear output layer; without a nonlinear activation ϕ, stacked affine transformations collapse into a single linear map. A single sufficiently wide hidden layer with any continuous non-polynomial activation can approximate any continuous function on a compact set, whereas composing m hidden layers folds space repeatedly to create 2^m linear regions with $O(m)$ units. Among activation functions, rectifiers such as ReLU ($\max(z, 0)$) and GELU ($z\Phi(z)$) maintain unit or near-unit slope on positive inputs, avoiding the tail saturation that attenuates gradients through deep stacks of sigmoid or tanh units.

4.1.6 Exercises

1. **Linear networks.** Without activation functions, stacking layers composes affine maps into another affine map. Consider such a network with input dimension d, output dimension q, and hidden widths $h_1, \dots, h_{K-1}$. Show that it represents exactly the affine maps $\mathbf{XW} + \mathbf{b}$ whose weight matrix $\mathbf{W}$ has rank at most $\min(h_1, \dots, h_{K-1})$. Use this result to give an example in which adding a hidden layer to a linear model strictly shrinks its function class.
2. **XOR by hand.** Figure 4.4 shows a two-unit ReLU network that computes XOR by mapping the two inputs of class 1 to the same hidden point.
   1. Find weights for a two-unit ReLU network that also computes XOR but maps the two inputs of class 1 to *distinct* hidden points, and verify it on the four inputs.
   2. Show that no network with a single ReLU hidden unit and a linear output classifies XOR correctly, for any threshold applied to its output.
3. **Sigmoid and tanh.** The two squashing functions are closely related.
   1. Show that $\tanh(z) = 2\sigma(2z) - 1$.
   2. Show that for every MLP with tanh hidden activations there is an MLP of the same architecture with sigmoid hidden activations that computes the same function, and conversely. State how the weights and biases change.
4. **Depth and linear pieces.** ● A one-hidden-layer ReLU network with h hidden units on the real line has at most $h + 1$ linear pieces, while each composition with the two-unit tent of Figure 4.6 doubles the number of pieces. This problem makes the gain from depth exact for a network with three units per layer.
   1. Take three hidden units $\mathrm{ReLU}(w_j x + b_j)$ with input weights $w_1 = w_2 = 1$ and $w_3 = -1$ and with joints $-b_j/w_j$ at $1/6$, $2/6$, and $4/6$. Choose output weights and an output bias so that the network's output f takes the values $0, 1, 0, 1, 0$ at $x = 0, 1/6, 2/6, 4/6, 1$.
   2. Count the linear pieces of $f \circ f$ on $[0, 1]$, and of the m-fold composition $f \circ \cdots \circ f$. Explain, in terms of how composing with f folds the graph, why each composition multiplies the count rather than adding to it.
   3. The m-fold composition is computed by a network with m hidden layers of three units each. How many hidden units would a network with a single hidden layer

need to represent the same function? Compare the parameter counts of the two networks.

Adapted from Simon Prince, Understanding Deep Learning, Problem 4.8.

5. **One hidden layer suffices.** ● On the real line, a single hidden layer approximates a continuous function by a polyline, one hinge per joint. In the plane there are no joints to place, and the proof needs another idea: functions that vary along one direction only.
   1. Let ψ be continuous on $\mathbb{R}$ and let $\mathbf{w} \in \mathbb{R}^2$. Show that a single hidden layer approximates the *ridge function* $\psi(\mathbf{w}^\top \mathbf{x})$, which is constant along every line orthogonal to $\mathbf{w}$, uniformly on any compact subset of the plane.
   2. The Stone–Weierstrass theorem states the following. Let a set of continuous functions on a compact set be closed under sums, products, and scalar multiples, contain the constants, and contain, for any two distinct points, a function that takes different values at them. Then every continuous function on the compact set is a uniform limit of functions in the set. Show that the finite linear combinations of $\cos(\omega^\top \mathbf{x})$ and $\sin(\omega^\top \mathbf{x})$, with $\omega \in \mathbb{R}^2$, form such a set.
   3. Conclude that a single hidden layer approximates any continuous function on a compact subset of the plane.

Adapted from Michael Nielsen, Neural Networks and Deep Learning, Chapter 4.

4.2 Implementation

In Section 4.1, we hand-crafted the weights of a two-unit ReLU network to solve XOR. Now let us let minibatch SGD learn hidden features directly from Fashion-MNIST images. Inserting a single hidden layer of $h = 256$ ReLU units between the $d = 784$ input pixels and the $q = 10$ class logits multiplies our parameter count and per-image FLOPs 26-fold ($203{,}530$ parameters vs. $7{,}850$ for softmax regression).

In this section, we implement this two-layer MLP both from scratch (`MLPScratch`) and concisely (`MLP`) using framework `Sequential` containers, verify our untrained network with three quick pre-training sanity checks, compare its learning curve and GPU epoch time head-to-head against softmax regression, and look at the weight patterns that the hidden units learn in $\mathbf{W}^{(1)}$.

```
import time
```

4.2.1 Implementation from Scratch

4.2.1.1 The Parameters and Forward Pass

Flattening each 28×28 image to $d = 784$ features, a one-hidden-layer MLP with $h = 256$ hidden units and $q = 10$ outputs (Figure 4.8) allocates four parameter tensors:

$$\mathbf{W}^{(1)} \in \mathbb{R}^{784\times256}, \quad \mathbf{b}^{(1)} \in \mathbb{R}^{256}, \quad \mathbf{W}^{(2)} \in \mathbb{R}^{256\times10}, \quad \mathbf{b}^{(2)} \in \mathbb{R}^{10}. \tag{4.6}$$

To keep our head-to-head comparison with softmax regression controlled, we initialize both weight matrices from $\mathcal{N}(0, 0.01^2)$ and both biases to 0 (in Section 4.4, we will see why random initialization is mandatory to break symmetry between hidden units, and how He initialization improves upon `std=0.01`):

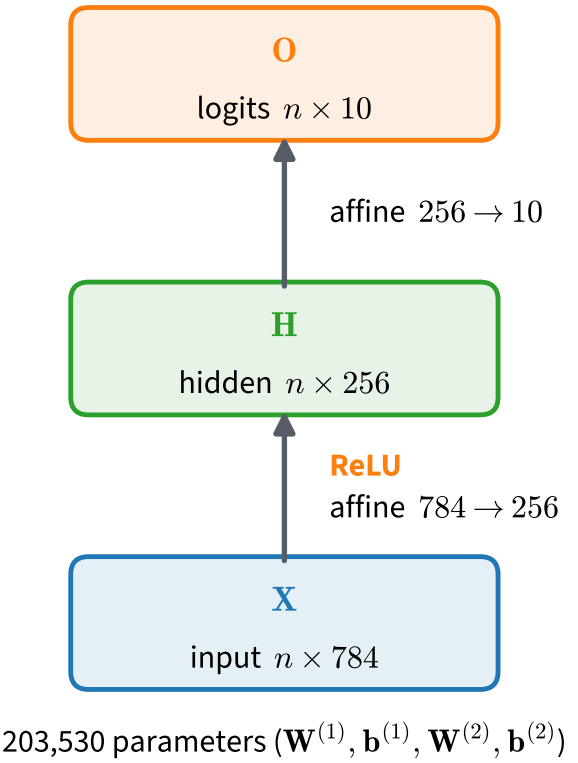


Figure 4.8: Architecture of our Fashion-MNIST MLP: a minibatch of n images is flattened to $n \times 784$, mapped through an affine layer and ReLU to an $n \times 256$ hidden representation $\mathbf{H}$, and projected to $n \times 10$ class logits $\mathbf{O}$.

```
class MLPScratch(d2l.Classifier):
    """An MLP with one hidden layer, implemented from scratch."""
    def __init__(self, num_inputs, num_outputs, num_hiddens, lr, std=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.W1 = nn.Parameter(d2l.normal(0, std, (num_inputs, num_hiddens)))
        self.b1 = nn.Parameter(d2l.zeros(num_hiddens))
        self.W2 = nn.Parameter(d2l.normal(0, std, (num_hiddens, num_outputs)))
        self.b2 = nn.Parameter(d2l.zeros(num_outputs))
```

Next, we implement the elementwise activation `relu(X)` from scratch by multiplying `X` with the Boolean mask `X > 0` so positive entries pass through unchanged and negative entries become zero:

```
def relu(X):
    return X * (X > 0)
```

We then factor the hidden layer into `hidden(self, X)`, which flattens each image minibatch to `(-1, self.num_inputs)` and computes $\mathbf{H} = \text{relu}(\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)})$, so `forward(self, X)` simply returns the raw class logits $\mathbf{O} = \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}$ (consumed directly by `Classifier.loss`):

```
@d2l.add_to_class(MLPScratch)
def hidden(self, X):
    X = d2l.reshape(X, (-1, self.num_inputs))  # One flattened image per row
    return relu(d2l.matmul(X, self.W1) + self.b1)

@d2l.add_to_class(MLPScratch)
def forward(self, X):
    return d2l.matmul(self.hidden(X), self.W2) + self.b2  # The logits
```

4.2.1.2 Checks before Training

Before launching a long training run, passing a tiny 8-image slice through the *untrained* network verifies three invariants in milliseconds:

1. **Shapes:** `H` has shape `(8, 256)` and `O` has shape `(8, 10)`.
2. **Initial loss:** Near-zero weights produce near-zero logits ($\hat{y}_j \approx 1/10$), so the initial cross-entropy loss sits right near $\log 10 \approx 2.303$.
3. **Active ReLU fraction:** With zero biases and symmetric weights, roughly 50% of hidden pre-activations are positive.

```
data = d2l.FashionMNIST(batch_size=256)
model = MLPScratch(num_inputs=784, num_outputs=10, num_hiddens=256, lr=0.1)
X, y = next(iter(data.train_dataloader()))
X, y = X[:8], y[:8]  # Eight images suffice for the checks
H, O = model.hidden(X), model(X)
active = d2l.reduce_mean(d2l.astype(H > 0, d2l.float32))
print('H:', H.shape, 'O:', O.shape)
print(f'loss {d2l.numpy(model.loss(O, y)):.3f}, '
```

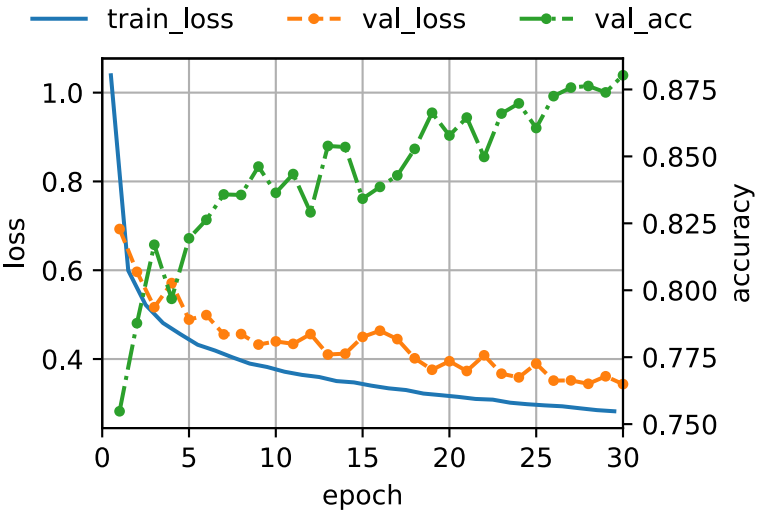
```
        f'fraction of active units {d2l.numpy(active):.2f}')
```

```
H: torch.Size([8, 256]) O: torch.Size([8, 10])
loss 2.300, fraction of active units 0.52
```

4.2.1.3 Training

Now we train `MLPScratch` for 30 epochs on Fashion-MNIST with minibatch size 256 and learning rate $\eta = 0.1$ (passing `num_gpus=1` so `Trainer` uses a GPU when available):

```
trainer = d2l.Trainer(max_epochs=30, num_gpus=1)
trainer.fit(model, data)
```



4.2.2 Concise Implementation

Using each framework's `Sequential` container and built-in linear and ReLU layers, we can express the exact same architecture in a few lines without manual parameter arrays (initializing weights from $\mathcal{N}(0, 0.01^2)$ and biases to 0 to match `MLPScratch`):

```
class MLP(d2l.Classifier):
    """An MLP with one hidden layer, built from framework layers."""
    def __init__(self, num_outputs, num_hiddens, lr):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(nn.Flatten(), nn.LazyLinear(num_hiddens),
                                 nn.ReLU(), nn.LazyLinear(num_outputs))

    def weight_init(self, module):
        # The Trainer calls this once the lazy layers know their shapes
        if isinstance(module, nn.Linear):
```

```
            nn.init.normal_(module.weight, std=0.01)
            nn.init.zeros_(module.bias)
```

4.2.3 Comparison with Softmax Regression

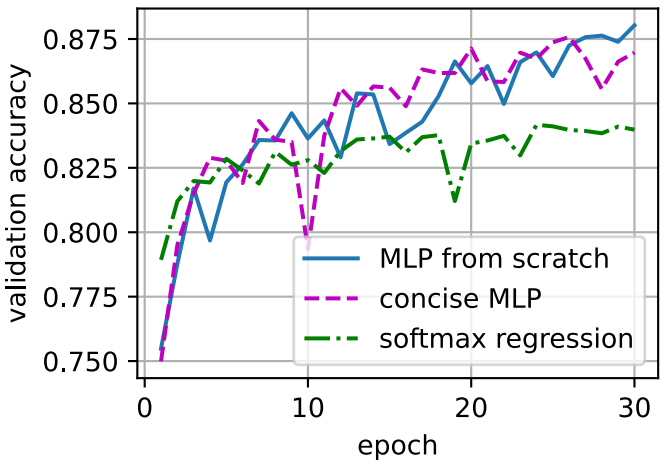
4.2.3.1 Accuracy

Let us train both the concise `MLP` and `d2l.SoftmaxRegression` for 30 epochs under identical settings, recording each model's average wall-clock time per epoch in `seconds` and its validation accuracy trajectory in `curves`:

```
curves, seconds = {'MLP from scratch': trainer.history['val_acc']}, {}
trainer.board.display = False
models = {'concise MLP': MLP(num_outputs=10, num_hiddens=256, lr=0.1),
          'softmax regression': d2l.SoftmaxRegression(num_outputs=10, lr=0.1)}
for name, net in models.items():
    start = time.perf_counter()
    trainer.fit(net, data)
    seconds[name] = (time.perf_counter() - start) / trainer.max_epochs
    curves[name] = trainer.history['val_acc']
```

Plotting all three validation accuracy histories in `curves` across epochs 1 to 30 compares our scratch and concise MLPs directly against the linear softmax baseline:

```
d2l.plot(list(range(1, trainer.max_epochs + 1)), list(curves.values()),
         'epoch', 'validation accuracy', legend=list(curves))
```



To smooth out single-epoch validation fluctuations, we print each model's mean validation accuracy over the final five epochs (`acc[-5:]`) alongside the training-set accuracy of `MLPScratch`:

```
print('validation accuracy averaged over the last 5 epochs')
for name, acc in curves.items():  # A single epoch can land a few points low
    print(f'  {name}: {sum(acc[-5:]) / 5:.3f}')
train_acc = trainer.evaluate(model, data, train=True)['acc']
print(f'training accuracy of the MLP from scratch: {train_acc:.3f}')
```

```
validation accuracy averaged over the last 5 epochs
  MLP from scratch: 0.876
  concise MLP: 0.867
  softmax regression: 0.840
training accuracy of the MLP from scratch: 0.903
```

Notice two instructive patterns in the learning curves:

1. **Slow warmup from `std=0.01`:** On epoch 1, both MLPs actually trail softmax regression. Because both $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(2)}$ start with entries around 0.01, the hidden activations $\mathbf{H}$ and the backpropagated gradients into $\mathbf{W}^{(1)}$ are suppressed by a factor of 0.01 until $\mathbf{W}^{(2)}$ grows, a bottleneck we eliminate with He initialization in Section 4.4.
2. **Cutting a fifth of the classification errors:** By epochs 26–30, softmax regression plateaus around 84% validation accuracy (16% error rate), whereas both MLPs reach $\sim 87\%$ validation accuracy ($\sim 13\%$ error rate, and 87–90% training accuracy), eliminating roughly 20% of softmax regression's mistakes.

Which mistakes are left? In Section 3.2 we saw that the four upper-body tops (T-shirt, pullover, coat, and shirt) look alike, so they are the natural suspects. For each of the two models we just trained, we count the validation errors and how many of them mistake one top for another:

```
tops = [0, 2, 4, 6]  # T-shirt/top, pullover, coat, shirt
for name, net in models.items():
    errors = among_tops = 0
    for batch in data.val_dataloader():
        X, y = trainer.prepare_batch(batch)
        pred, y = d2l.numpy(net(X)).argmax(axis=1), d2l.numpy(y)
        errors += (pred != y).sum()
        among_tops += ((pred != y) & np.isin(y, tops) & np.isin(pred, tops)).sum()
    print(f'{name}: {errors} errors, {among_tops} of them between two tops')
```

```
concise MLP: 1303 errors, 772 of them between two tops
softmax regression: 1602 errors, 959 of them between two tops
```

For both models, more than half of all errors confuse two tops. The MLP makes fewer such errors than softmax regression does, but the tops remain the hard part of Fashion-MNIST.

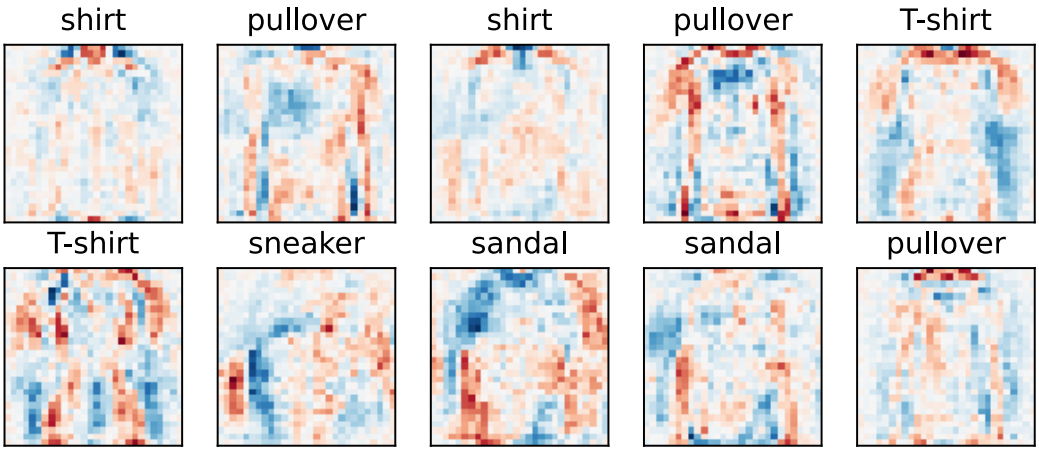
4.2.3.2 The Learned Features

In Section 3.3, the 10 columns of softmax regression's weight matrix $\mathbf{W}$ had to act as whole-garment templates. To inspect what our MLP's 256 hidden units learned instead, we extract its trained weight matrices $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(2)}$ as NumPy arrays:

```
W1, W2 = d2l.numpy(model.W1), d2l.numpy(model.W2)
```

We display the 10 hidden units with the largest Euclidean weight norms as 28×28 images, labeling each unit by the garment class to which it assigns its largest positive outgoing weight:

```
names = data.text_labels(range(10))
units = np.argsort(np.linalg.norm(W1, axis=0))[::-1][:10]  # Largest weights
images = W1[:, units].T.reshape(10, 28, 28)
images /= np.abs(images).max(axis=(1, 2), keepdims=True)  # Own color scale
d2l.show_images(images, 2, 5, titles=[names[k] for k in W2[units].argmax(1)],
                cmap='RdBu_r', vmin=-1, vmax=1);
```



These images do not look like whole-garment templates. Each high-norm hidden unit weighs some regions of the image positively (red) and neighboring regions negatively (blue), so it responds to a **contrast**: bright pixels in one region together with dark pixels next to it, for instance along the shoulders or down the sides of the torso. Most of the patterns still extend over much of the garment, so we should not read them as small edge detectors. The output layer forms its ten scores by combining 256 such contrasts.

4.2.3.3 Parameters, Computation, and Time

Model	Parameters	Multiply-adds / image	Validation accuracy (epochs 26–30)
Softmax regression	7,850	7,840	$\sim 84\%$
MLP ($h = 256$ ReLU units)	203,530	203,264	$\sim 87\%$

Even though our MLP performs 26 times as many multiply-adds per image as softmax regression, let us compare their wall-clock times per epoch:

```
for name, s in seconds.items():
    print(f'{name}: {s:.1f} s per epoch')
```

```
concise MLP: 1.3 s per epoch
softmax regression: 1.3 s per epoch
```

On modern hardware (especially a GPU), both models take almost the same wall-clock time per epoch. At width $h = 256$, data loading and Python or kernel-launch overhead dominate the runtime, so the 26-fold increase in dense matrix arithmetic adds almost no wall-clock cost (Exercise 3 explores the width at which matrix arithmetic finally overtakes data-loading overhead).

4.2.4 Summary

Before launching training, a single forward pass on a small batch verifies intermediate and output tensor shapes, confirms that the initial cross-entropy loss sits near $\log q \approx 2.30$, and checks that roughly half of the ReLU hidden units activate. On Fashion-MNIST, inserting one 256-unit ReLU hidden layer lifts validation accuracy from $\sim 84\%$ to $\sim 87\%$, cutting the error rate by nearly a fifth at almost identical per-epoch GPU wall-clock time. Rather than forming blurred whole-garment templates, the hidden units learn contrast patterns that weigh neighboring regions with opposite signs and combine across the output layer.

4.2.5 Exercises

1. code **A hidden layer without activation.** Replace `relu` by the identity in `MLPScratch` and train the resulting model as in this section.
   1. Compare its validation accuracy after 30 epochs with that of the MLP and of softmax regression.
   2. After training, compute $\mathbf{W} = \mathbf{W}^{(1)}\mathbf{W}^{(2)}$ and $\mathbf{b} = \mathbf{W}^{(2)\top}\mathbf{b}^{(1)} + \mathbf{b}^{(2)}$, and verify on a minibatch that $\mathbf{XW} + \mathbf{b}$ reproduces the logits of the model.
   3. Exercise 1 showed that such a network computes exactly the affine maps whose weight matrix has rank at most the hidden width. Explain why this restriction costs nothing here, and state below which hidden width it would reduce what the model can represent on Fashion-MNIST.
2. code **Single hidden unit.** Set `num_hiddens=1` in `MLPScratch`, so that its hidden layer has a single ReLU unit with weights $\mathbf{w}^{(1)} \in \mathbb{R}^{784\times 1}$ and bias $b^{(1)} \in \mathbb{R}$. Each input $\mathbf{x} \in \mathbb{R}^{1\times 784}$ is then reduced to the scalar $t = \mathrm{ReLU}(\mathbf{x}\mathbf{w}^{(1)} + b^{(1)}) \geq 0$, and the logits are $\mathbf{o} = t\,\mathbf{w}^{(2)} + \mathbf{b}^{(2)}$ with $\mathbf{w}^{(2)} \in \mathbb{R}^{1\times 10}$ and $\mathbf{b}^{(2)} \in \mathbb{R}^{10}$.
   1. Show that each class is predicted on at most one interval of t, and that the order of these intervals along the t axis is fixed by the entries of $\mathbf{w}^{(2)}$. Describe the regions of input space that the model assigns to the classes, and state what they imply for separating the ten Fashion-MNIST classes.
   2. Suppose that after some update $\mathbf{x}\mathbf{w}^{(1)} + b^{(1)} < 0$ for every training example. Which parameters receive nonzero gradients from then on, and what does the network predict? Why does the same event do little harm when it hits one of the 256 hidden units of this section?
   3. Train the model five times, and report for each run the validation accuracy and the fraction of validation examples on which $t > 0$, which `model.hidden` gives you. Compare any run in which this fraction is zero with your answer to (b).
3. code **Width.** Train `MLP` as in this section, for 30 epochs at learning rate 0.1, with hidden widths $h \in \{16, 64, 256, 1024, 4096\}$, and record the final validation accuracy and the time per epoch.
   1. Plot both against h on a logarithmic axis. Beyond which width does the accuracy stop improving by more than its variation from epoch to epoch?
   2. The multiply-adds grow in proportion to h. Time one pass over the training data

that only reads the minibatches, without any model, and use it to explain the part of the time curve that does not grow with h. From which width on does the arithmetic dominate on your hardware?

3. Repeat the training at widths 64 and 1024 with learning rates $\{0.03, 0.1, 0.3\}$. Does the best learning rate depend on the width?

4.3 Backpropagation

In Section 4.2, we trained all $203{,}530$ parameters of our Fashion-MNIST MLP without writing a single derivative by hand: automatic differentiation computed every gradient for us. In Section 1.5, we saw the general principle—reverse-mode automatic differentiation records the forward pass as a computational graph and walks the chain rule backward from the scalar loss.

In this section, we open that black box for multilayer perceptrons. An entire MLP requires just **three local matrix-calculus rules**: matrix multiplication, broadcasted bias addition, and elementwise activation. Walking through them answers three essential engineering questions:

1. How does a single backward pass compute hundreds of thousands of partial derivatives for no more than about $2\times$ the arithmetic cost of one forward pass?
2. Why does training consume so much more GPU memory than inference, and why does that memory scale linearly with minibatch size n and depth K?
3. How do dead ReLU units and vanishing/exploding gradients arise mechanically from the backward recurrence?

4.3.1 Computational Graphs

4.3.1.1 A Small Graph

To review reverse-mode differentiation in its barest form, consider the three-scalar expression $e = (a + b)\,c$ at $a = 2, b = 1, c = -3$ (Figure 4.9). The *forward pass* (left to right) computes $u = a + b = 3$ and $e = u\,c = -9$. The *backward pass* (right to left) seeds $\partial e/\partial e = 1$ and applies one local derivative per node:

- At the multiplication node $e = u\,c$: $\partial e/\partial u = c = -3$ and $\partial e/\partial c = u = 3$.
- At the addition node $u = a + b$: $\partial u/\partial a = \partial u/\partial b = 1$, so $\partial e/\partial a = \partial e/\partial b = -3$.

Notice two universal rules already visible here: (i) an addition node copies its incoming gradient unchanged to each summand, whereas a multiplication node multiplies the incoming gradient by the *other* factor (requiring the forward pass to cache u and c); and (ii) the backward pass traverses the graph in *reverse topological order*.

4.3.1.2 The Graph of an MLP

Now consider our one-hidden-layer MLP (Figure 4.10) on a minibatch $\mathbf{X} \in \mathbb{R}^{n\times d}$ with parameters $\mathbf{W}^{(1)} \in \mathbb{R}^{d\times h}$, $\mathbf{b}^{(1)} \in \mathbb{R}^{h}$, $\mathbf{W}^{(2)} \in \mathbb{R}^{h\times q}$, $\mathbf{b}^{(2)} \in \mathbb{R}^{q}$, and ℓ_2 weight-decay penalty $s = \frac{\lambda}{2}(\|\mathbf{W}^{(1)}\|_{\mathrm{F}}^2 + \|\mathbf{W}^{(2)}\|_{\mathrm{F}}^2)$:

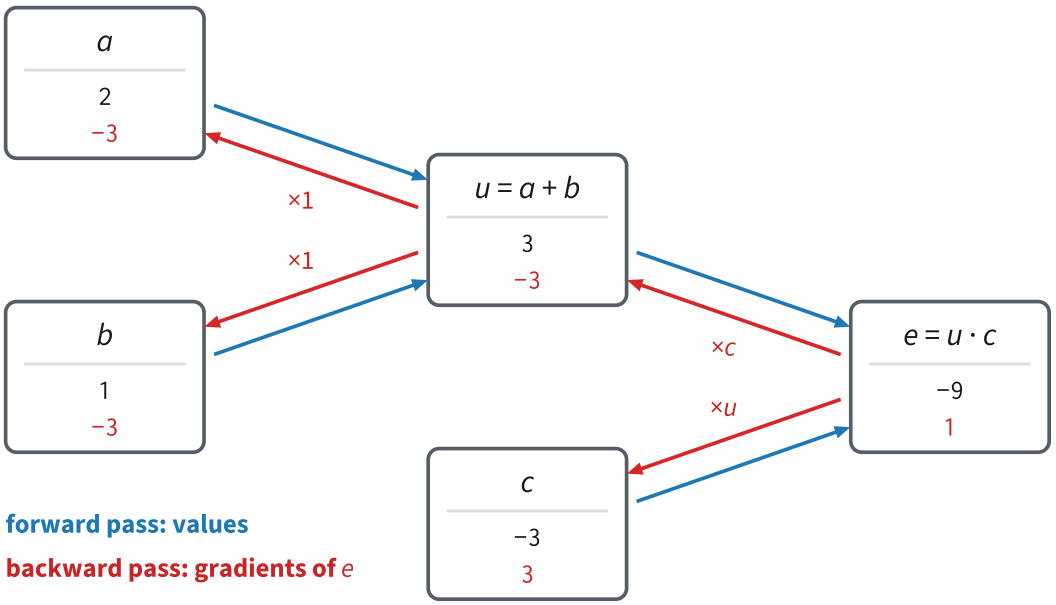


Figure 4.9: Computational graph of $e = (a+b)\,c$ at $a = 2, b = 1, c = -3$, showing forward values (black) and reverse-mode derivatives $\partial e/\partial(\cdot)$ (red).

$$\begin{aligned} \mathbf{Z} &= \mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}, \\ \mathbf{H} &= \phi(\mathbf{Z}), \\ \mathbf{O} &= \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}, \\ J &= L(\mathbf{O}, \mathbf{Y}) + s, \end{aligned} \tag{4.7}$$

where L is the mean cross-entropy loss over the minibatch and $\|\mathbf{W}\|_{\mathrm{F}}^2 = \sum_{r,j} w_{rj}^2$ is the squared Frobenius norm.

4.3.2 Backpropagation through an MLP

4.3.2.1 Local Rules for Matrices

For any matrix $\mathbf{A}$, we write $\partial J/\partial \mathbf{A}$ for the matrix of partial derivatives $(\partial J/\partial a_{ij})$ of the *exact same shape* as $\mathbf{A}$. Three local vector–Jacobian rules (Section 34.4.3.2) handle every layer of an MLP:

1. **Matrix multiplication ($\mathbf{A} = \mathbf{X}\mathbf{W}$ with $\mathbf{X} \in \mathbb{R}^{n\times d}, \mathbf{W} \in \mathbb{R}^{d\times h}$):** Given upstream gradient $\mathbf{G} = \partial J/\partial \mathbf{A} \in \mathbb{R}^{n\times h}$, since $a_{ij} = \sum_r x_{ir} w_{rj}$, the chain rule yields $\partial J/\partial w_{rj} = \sum_i x_{ir} g_{ij}$ and $\partial J/\partial x_{ir} = \sum_j g_{ij} w_{rj}$, or in matrix form:

$$\frac{\partial J}{\partial \mathbf{W}} = \mathbf{X}^\top \mathbf{G}, \qquad \frac{\partial J}{\partial \mathbf{X}} = \mathbf{G}\,\mathbf{W}^\top. \tag{4.8}$$

   Notice how the shapes lock into place: $\mathbf{X}^\top\mathbf{G}$ is $(d \times n)(n \times h) = d \times h$ (matching $\mathbf{W}$), and $\mathbf{G}\mathbf{W}^\top$ is $(n \times h)(h \times d) = n \times d$ (matching $\mathbf{X}$). Just like scalar multiplication,

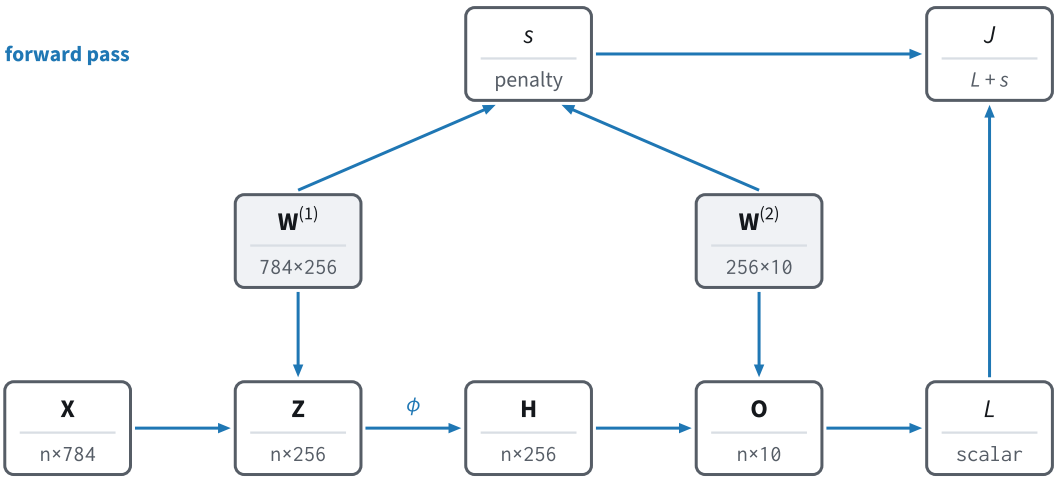


Figure 4.10: Computational graph of the forward pass of a one-hidden-layer MLP with weight decay ($d = 784, h = 256, q = 10$). Parameter nodes are shaded; each weight matrix feeds both its affine layer and the regularization penalty s.

computing $\partial J/\partial \mathbf{W}$ requires the cached input $\mathbf{X}$, while computing $\partial J/\partial \mathbf{X}$ propagates the error signal backward through $\mathbf{W}^\top$.

2. **Broadcasted bias addition ($\mathbf{Z} = \mathbf{A} + \mathbf{b}$ with $\mathbf{b} \in \mathbb{R}^h$):** Since b_j is added to every row i of column j, $\partial J/\partial \mathbf{A} = \mathbf{G}$ passes straight through unchanged, while $\partial J/\partial \mathbf{b} = \mathbf{G}^\top \mathbf{1}_n$ sums $\mathbf{G}$ across the n examples (rows).
3. **Elementwise activation ($\mathbf{H} = \phi(\mathbf{Z})$):** Since $h_{ij} = \phi(z_{ij})$ depends only on entry (i, j), the upstream gradient is gated entrywise (Hadamard product $\odot$) by the local activation slope:

$$\frac{\partial J}{\partial \mathbf{Z}} = \frac{\partial J}{\partial \mathbf{H}} \odot \phi'(\mathbf{Z}).$$

4.3.2.2 The Backward Pass

Starting from $\partial J/\partial \mathbf{O} = \frac{1}{n}(\hat{\mathbf{Y}} - \mathbf{Y})$ (Section 3.1.2.2, where $\hat{\mathbf{Y}} = \text{softmax}(\mathbf{O})$ and $\mathbf{Y} \in \{0, 1\}^{n \times q}$ is the one-hot label matrix) and $\partial s/\partial \mathbf{W}^{(k)} = \lambda \mathbf{W}^{(k)}$, applying our three rules in reverse topological order from output to input gives the complete MLP backward pass:

$$\begin{aligned}
\frac{\partial J}{\partial \mathbf{O}} &= \frac{1}{n}(\hat{\mathbf{Y}} - \mathbf{Y}), \\
\frac{\partial J}{\partial \mathbf{W}^{(2)}} &= \mathbf{H}^\top \frac{\partial J}{\partial \mathbf{O}} + \lambda \mathbf{W}^{(2)}, \qquad \frac{\partial J}{\partial \mathbf{b}^{(2)}} = \left(\frac{\partial J}{\partial \mathbf{O}}\right)^\top \mathbf{1}, \\
\frac{\partial J}{\partial \mathbf{H}} &= \frac{\partial J}{\partial \mathbf{O}} \mathbf{W}^{(2)\top}, \\
\frac{\partial J}{\partial \mathbf{Z}} &= \frac{\partial J}{\partial \mathbf{H}} \odot \phi'(\mathbf{Z}), \\
\frac{\partial J}{\partial \mathbf{W}^{(1)}} &= \mathbf{X}^\top \frac{\partial J}{\partial \mathbf{Z}} + \lambda \mathbf{W}^{(1)}, \qquad \frac{\partial J}{\partial \mathbf{b}^{(1)}} = \left(\frac{\partial J}{\partial \mathbf{Z}}\right)^\top \mathbf{1}.
\end{aligned} \tag{4.9}$$

For a general K-layer MLP ($\mathbf{Z}^{(k)} = \mathbf{H}^{(k-1)}\mathbf{W}^{(k)} + \mathbf{b}^{(k)}, \mathbf{H}^{(k)} = \phi(\mathbf{Z}^{(k)})$), each layer $k = K-1, \ldots, 1$ executes the exact same two-step module:

$$\frac{\partial J}{\partial \mathbf{Z}^{(k)}} = \left(\frac{\partial J}{\partial \mathbf{Z}^{(k+1)}}\mathbf{W}^{(k+1)\top}\right) \odot \phi'(\mathbf{Z}^{(k)}),$$
$$\frac{\partial J}{\partial \mathbf{W}^{(k)}} = \mathbf{H}^{(k-1)\top}\frac{\partial J}{\partial \mathbf{Z}^{(k)}} + \lambda \mathbf{W}^{(k)}.$$

4.3.2.3 A Worked Example

Let us trace Equation 4.9 by hand on a tiny network ($n = 1, d = h = 2, q = 1$, ReLU activation, zero biases, $\lambda = 0$, and squared loss $L = \frac{1}{2}(o-y)^2$):

$$\mathbf{X} = \begin{bmatrix}1 & 2\end{bmatrix}, \quad \mathbf{W}^{(1)} = \begin{bmatrix}1 & 0\\ -1 & 1\end{bmatrix}, \quad \mathbf{W}^{(2)} = \begin{bmatrix}2\\ -1\end{bmatrix}, \quad y = 0. \tag{4.10}$$

In the forward pass (Figure 4.11), $\mathbf{Z} = \mathbf{X}\mathbf{W}^{(1)} = \begin{bmatrix}-1 & 2\end{bmatrix}$, $\mathbf{H} = \text{ReLU}(\mathbf{Z}) = \begin{bmatrix}0 & 2\end{bmatrix}$ (unit 1 is inactive), $o = \mathbf{H}\mathbf{W}^{(2)} = -2$, and $L = \frac{1}{2}(-2)^2 = 2$. In the backward pass, $\partial L/\partial o = o - y = -2$, and

$$\frac{\partial L}{\partial \mathbf{W}^{(2)}} = \mathbf{H}^\top \frac{\partial L}{\partial o} = \begin{bmatrix}0\\ 2\end{bmatrix}(-2) = \begin{bmatrix}0\\ -4\end{bmatrix},$$
$$\frac{\partial L}{\partial \mathbf{H}} = \frac{\partial L}{\partial o}\mathbf{W}^{(2)\top} = \begin{bmatrix}-4 & 2\end{bmatrix},$$
$$\frac{\partial L}{\partial \mathbf{Z}} = \begin{bmatrix}-4 & 2\end{bmatrix} \odot \begin{bmatrix}0 & 1\end{bmatrix} = \begin{bmatrix}0 & 2\end{bmatrix},$$
$$\frac{\partial L}{\partial \mathbf{W}^{(1)}} = \mathbf{X}^\top \frac{\partial L}{\partial \mathbf{Z}} = \begin{bmatrix}1\\ 2\end{bmatrix}\begin{bmatrix}0 & 2\end{bmatrix} = \begin{bmatrix}0 & 2\\ 0 & 4\end{bmatrix}.$$

Notice that the entire first column of $\partial L/\partial \mathbf{W}^{(1)}$ is zero: because $z_1 = -1 < 0$, $\phi'(z_1) = 0$ blocks the upstream gradient $\partial L/\partial h_1 = -4$ from reaching any weight feeding unit 1.

4.3.2.4 Checking the Backward Pass on a Minibatch

Let us implement Equation 4.9 in `mlp_backward` and verify it in 64-bit floating point against each framework's automatic differentiation on a random 256×784 minibatch ($h = 256, q = 10, \lambda = 10^{-3}$):

```
def mlp_backward(X, Y, W1, b1, W2, b2, lambd):
    """Gradients of J = cross-entropy + weight decay, by backprop."""
    Z = X @ W1 + b1  # forward pass, keeping X, Z, H, and Y_hat
    H = torch.relu(Z)
    Y_hat = torch.softmax(H @ W2 + b2, dim=1)
    G_O = (Y_hat - Y) / X.shape[0]  # dJ/dO
    dW2, db2 = H.T @ G_O + lambd * W2, G_O.sum(dim=0)
    G_Z = (G_O @ W2.T) * (Z > 0)  # dJ/dH, then dJ/dZ
    dW1, db1 = X.T @ G_Z + lambd * W1, G_Z.sum(dim=0)
    return dW1, db1, dW2, db2
```

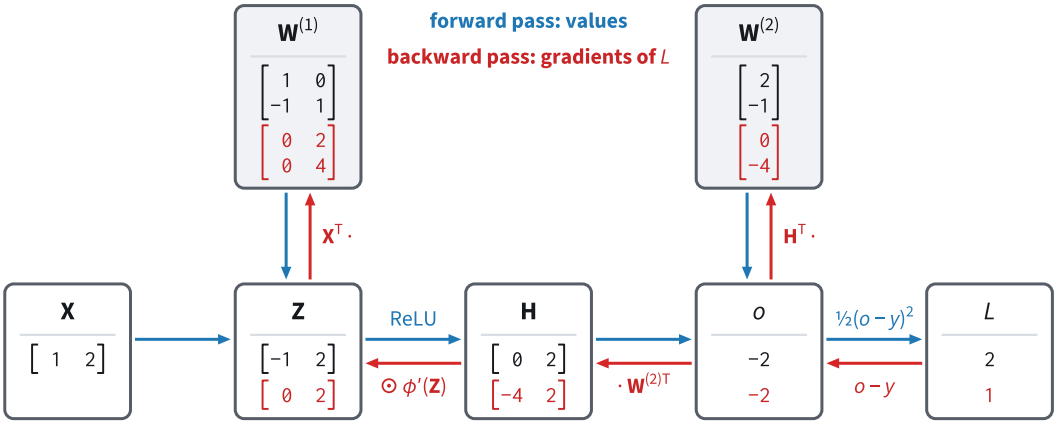


Figure 4.11: Forward values (black) and backward gradients (red) for the worked example Equation 4.10. Because $z_1 = -1 < 0$, the ReLU gate zeroes out $\partial L/\partial z_1$ and the first column of $\partial L/\partial \mathbf{W}^{(1)}$.

Let us generate a random 256-example minibatch with the exact dimensions and initialization of Section 4.2 plus $\lambda = 10^{-3}$, and compare `mlp_backward` against each framework's built-in automatic differentiation. Following best practice for gradient checks, we run in 64-bit floating point (whose ~10^{-16} rounding floor catches even tiny algebraic bugs cleanly) and report the maximum relative discrepancy per parameter tensor.

```
n, d, h, q, lambd = 256, 784, 256, 10, 1e-3
f64 = torch.float64  # 64-bit floating point throughout the check
X, y = torch.randn(n, d, dtype=f64), torch.randint(0, q, (n,))
params = [torch.randn(d, h, dtype=f64) * 0.01, torch.zeros(h, dtype=f64),
          torch.randn(h, q, dtype=f64) * 0.01, torch.zeros(q, dtype=f64)]
for p in params:
    p.requires_grad_(True)
W1, b1, W2, b2 = params
J = (F.cross_entropy(torch.relu(X @ W1 + b1) @ W2 + b2, y)
     + lambd / 2 * ((W1 ** 2).sum() + (W2 ** 2).sum()))
J.backward()  # automatic differentiation
with torch.no_grad():
    grads = mlp_backward(X, F.one_hot(y, q).to(f64), *params, lambd)
for name, p, g in zip(['W1', 'b1', 'W2', 'b2'], params, grads):
    rel = (p.grad - g).abs().max() / p.grad.abs().max()
    print(f'{name}: relative difference {float(rel):.1e}')
```

```
W1: relative difference 4.4e-16
b1: relative difference 2.9e-16
W2: relative difference 3.8e-16
b2: relative difference 2.1e-16
```

Every parameter gradient matches automatic differentiation to within 10^{-15} relative error, a few units of the double-precision machine epsilon 2.2×10^{-16}.

4.3.3 Memory and Computation

4.3.3.1 Stored Activations vs. Parameter Memory

Looking at Equation 4.9, computing $\partial J/\partial \mathbf{W}^{(2)}$ and $\partial J/\partial \mathbf{W}^{(1)}$ during the backward pass requires the forward pass to retain $\mathbf{X}$, $\mathbf{H}$ (which for ReLU also encodes the mask $\mathbf{Z} > 0$), and $\widehat{\mathbf{Y}}$. On our Fashion-MNIST MLP ($d = 784, h = 256, q = 10$) with minibatch size $n = 256$, these cached tensors hold

$$256 \times 784 + 256 \times 256 + 256 \times 10 = 268{,}800 \text{ floats} \approx 1.1 \text{ MB}, \tag{4.11}$$

already exceeding the $203{,}530$ parameters (≈ 0.8 MB). In a K-layer network, activation memory grows linearly with *both* batch size n and depth K, whereas inference can discard each layer's activations immediately after computing the next layer.

4.3.3.2 Arithmetic Cost

In the forward pass, each example performs $dh + hq = 784 \times 256 + 256 \times 10 = 203{,}264$ multiply-adds. In the backward pass, each interior matrix multiply $\mathbf{H}^{(k-1)}\mathbf{W}^{(k)}$ spawns *two* backward matrix multiplies of the same size ($\mathbf{H}^{(k-1)\top}\mathbf{G}$ for the weights and $\mathbf{G}\mathbf{W}^{(k)\top}$ for the activations below), while the first layer needs only $\mathbf{X}^\top \mathbf{G}_Z$ (since $\partial J/\partial \mathbf{X}$ is not needed). For our two-layer MLP the backward pass therefore takes $dh + 2hq = 205{,}824$ multiply-adds, about the same as the forward pass: the first layer holds almost all of the work, and it needs a single backward product. In a deep network nearly every layer is interior, so the backward pass costs $\sim 2\times$ the forward pass and a full training step roughly $3\times$ forward inference. Either way, all $203{,}530$ derivatives cost one or two extra forward passes, whereas a one-sided finite difference spends one extra forward pass on a *single* scalar parameter.

4.3.4 Summary

Backpropagation through a multilayer perceptron reduces to three local matrix rules applied in reverse topological order: $\partial J/\partial \mathbf{W} = \mathbf{X}^\top \mathbf{G}$ and $\partial J/\partial \mathbf{X} = \mathbf{G}\mathbf{W}^\top$ for matrix products $\mathbf{A} = \mathbf{X}\mathbf{W}$, row summation $\mathbf{G}^\top \mathbf{1}$ for broadcasted biases, and elementwise gating $\mathbf{G} \odot \phi'(\mathbf{Z})$ for activations, summing incoming gradients wherever a parameter feeds both the data loss and weight decay. Because $\mathbf{X}^\top \mathbf{G}$ depends on the forward inputs to each layer, training must cache intermediate activations across the forward sweep, requiring $O(nKh)$ memory that grows linearly with batch size and depth. In return, a single backward sweep computes all parameter gradients in at most about twice the FLOPs of one forward pass.

4.3.5 Exercises

1. **Jacobian of a layer.** The gradient of the scalar J with respect to $\mathbf{W}^{(2)} \in \mathbb{R}^{h \times q}$ has the shape of $\mathbf{W}^{(2)}$. The derivative of the matrix $\mathbf{O} = \mathbf{H}\mathbf{W}^{(2)} \in \mathbb{R}^{n \times q}$ with respect to $\mathbf{W}^{(2)}$ is a larger object, with one entry $\partial o_{ij}/\partial w_{rt}$ for every combination of the four indices.
    1. How many entries does $\partial \mathbf{O}/\partial \mathbf{W}^{(2)}$ have, and how many of them can be nonzero?
    2. Evaluate the chain rule $\partial J/\partial w_{rt} = \sum_{i,j} (\partial J/\partial o_{ij})(\partial o_{ij}/\partial w_{rt})$ to recover $\mathbf{H}^\top \partial J/\partial \mathbf{O}$.
    3. How much memory would $\partial \mathbf{O}/\partial \mathbf{W}^{(2)}$ take in 32-bit floating point for the MLP of Section 4.2 with $n = 256$? Explain why backpropagation never forms it.

2. **Memory of a deeper network.** Consider an MLP with d inputs, q outputs, and K layers, of which the $K-1$ hidden ones are ReLU layers of width h, trained on minibatches of n examples.

    1. Using the step for K layers in the text, list the values that the backward pass reads, and count them. Count the parameters.
    2. Prediction can discard each value as soon as the next layer has used it. Count the largest number of values that prediction holds at any moment.
    3. For $d = 784$, $h = 256$, and $q = 10$, find the smallest minibatch size at which the stored values outnumber the parameters, for $K = 2$ and for $K = 11$.

3. code **Gradient checking.** Before automatic differentiation, a hand-written backward pass was checked against finite differences.

    1. Pick one entry w of $\mathbf{W}^{(1)}$ and compare its gradient from `mlp_backward` with the central difference $(J(w+\epsilon) - J(w-\epsilon))/(2\epsilon)$ for $\epsilon = 10^{-1}, 10^{-2}, \dots, 10^{-7}$, once in 32-bit and once in 64-bit floating point. Plot the relative error against ϵ on logarithmic axes.
    2. Explain the shape of the curves. Which error decreases with ϵ, which increases, and where do they balance in each precision?
    3. The check can fail at a correct gradient when a pre-activation lies within ϵ of zero. Explain why, and how to avoid it.

4. **Forward mode.** Backpropagation seeds $\partial J/\partial J = 1$ at the output and sweeps the graph backward. Forward-mode differentiation instead seeds $\partial w/\partial w = 1$ at one chosen scalar parameter w and sweeps forward: each node computes the derivative of its value with respect to w from the derivatives of its inputs.

    1. Use forward mode to recompute $\partial L/\partial \mathbf{W}^{(1)}$ for the network in Equation 4.10, one sweep per entry of $\mathbf{W}^{(1)}$.
    2. How many sweeps does each mode need to obtain all six weight gradients of this network? Which products does forward mode compute again in every sweep that reverse mode computes only once?

    Adapted from Simon Prince, Understanding Deep Learning, Problems 7.12 and 7.13.

5. [extended] **Autograd engine.** ● Build a miniature automatic differentiation engine and use it to recompute the gradients of the network in Equation 4.10.

    1. Write a scalar `Value` class that records, for each result, its inputs and the operation that produced it (the computational graph), supporting `+`, `*`, and `relu`. (*Hint:* implement `__add__` and `__mul__` so that each returns a new `Value` holding references to its parents and a small function that propagates the gradient one step.)
    2. Implement `backward()`: sort the graph topologically, seed the output's gradient with 1, and visit the nodes in reverse order, letting each node pass its gradient to its parents. Make sure that gradients *accumulate* (`+=`, not `=`) when a value is used more than once.
    3. Recompute the graph of Figure 4.9 and the worked example Equation 4.10 with your engine (unroll the matrix products into scalars), and compare the gradients with those derived in this section.
    4. Consider the chain in which a scalar input x produces $v_i = f_i(x)$ for $i = 1, 2, 3$, the intermediate value $m = v_1 + v_2 + v_3$, the values $p_j = g_j(m)$ for $j = 1, 2, 3$, and the loss $L = p_1 + p_2 + p_3$. Write $\alpha_i = f_i'(x)$ and $\beta_j = g_j'(m)$. The chain rule as a sum over paths enumerates $3 \times 3 = 9$ paths from x to L, each contributing

the product $\alpha_i\beta_j$. Show that your engine instead computes $(\alpha_1+\alpha_2+\alpha_3)(\beta_1+\beta_2+\beta_3)$ by accumulating the gradient at m once, and explain why this sharing keeps backpropagation affordable in deep networks.

Adapted from Andrej Karpathy, micrograd, `micrograd/engine.py`.

4.4 Numerical Stability and Initialization

In Section 4.2, initializing our one-hidden-layer MLP with $\mathcal{N}(0, 0.01^2)$ weights caused a sluggish first epoch, yet the network recovered and reached 87% accuracy. What happens if we stack *ten* hidden layers of width 256 with that same $\mathcal{N}(0, 0.01^2)$ initialization? Ten epochs later, the loss has not budged from $\log 10 \approx 2.30$ and accuracy sits at 10% random guessing. Conversely, if we initialize the weights with $\mathcal{N}(0, 1)$, activations and gradients come out astronomically large, and a single SGD update throws the weights so far that the next forward pass overflows to `inf` and `NaN`.

As we saw in Section 4.3, forward activations and backward gradients pass through a product of K weight and slope matrices across a K-layer network. Unless each layer preserves the signal variance, keeping its multiplicative factor near 1, signals either *vanish* exponentially to zero or *explode* exponentially to infinity. In this section, we quantify per-layer variance propagation, derive *Xavier* and *He* initialization, verify them on a 10-layer MLP on Fashion-MNIST, and prove why initial weights must be *random* to break permutation symmetry among hidden units.

4.4.1 Vanishing and Exploding Gradients

4.4.1.1 A Product over Layers

In a K-layer MLP with hidden layers $\mathbf{Z}^{(k)} = \mathbf{H}^{(k-1)}\mathbf{W}^{(k)} + \mathbf{b}^{(k)}$ and $\mathbf{H}^{(k)} = \phi(\mathbf{Z}^{(k)})$ of width h, the backward recurrence from Section 4.3 propagates activation gradients one layer down via

$$\frac{\partial L}{\partial \mathbf{H}^{(k-1)}} = \left(\frac{\partial L}{\partial \mathbf{H}^{(k)}} \odot \phi'\big(\mathbf{Z}^{(k)}\big)\right) \mathbf{W}^{(k)\top}. \tag{4.12}$$

Unrolling Equation 4.12 from the top hidden layer $K-1$ down to layer k expresses the gradient at layer k as a product of $K-1-k$ matrix–slope factors:

$$\begin{aligned}\frac{\partial L}{\partial \mathbf{H}^{(k)}} = \frac{\partial L}{\partial \mathbf{H}^{(K-1)}} \underbrace{\odot\ \phi'\big(\mathbf{Z}^{(K-1)}\big)\, \mathbf{W}^{(K-1)\top}}_{\text{layer } K-1} \\ \cdots \underbrace{\odot\ \phi'\big(\mathbf{Z}^{(k+1)}\big)\, \mathbf{W}^{(k+1)\top}}_{\text{layer } k+1}.\end{aligned} \tag{4.13}$$

For a single example with upstream gradient row $\mathbf{g}'$ and downstream gradient row $\mathbf{g}$ across a layer of width h whose weights w_{ij} are drawn i.i.d. with mean 0 and variance τ^2, each coordinate $g_i = \sum_{j=1}^{h} w_{ij}\, \phi'(z_j)\, g'_j$ has expected square (treating weights, slopes, and upstream gradients as approximately independent at initialization)

$$\begin{aligned}\mathbb{E}[g_i^2] &= \sum_{j=1}^{h} \mathbb{E}[w_{ij}^2]\,\mathbb{E}[\phi'(z_j)^2]\,\mathbb{E}[(g_j')^2] \\ &= \underbrace{h\,\tau^2\,\mathbb{E}\big[\phi'(z)^2\big]}_{\text{per-layer factor}}\,\mathbb{E}[(g_j')^2].\end{aligned} \tag{4.14}$$

Across t layers, the gradient's mean square is multiplied by $\big(h\,\tau^2\,\mathbb{E}[\phi'(z)^2]\big)^t$. For symmetric pre-activations and ReLU, $\phi'(z) \in \{0, 1\}$ is 1 half the time, so $\mathbb{E}[\phi'(z)^2] = 1/2$ and the per-layer multiplier is $\frac{1}{2}h\tau^2$. At width $h = 256$:

- With $\tau = 0.01$: the multiplier is $256 \times 10^{-4}/2 = 0.0128$, which after 9 hops shrinks the gradient mean square by $0.0128^9 \approx 10^{-17}$ (**vanishing gradients**).
- With $\tau = 1$: the multiplier is $256 \times 1/2 = 128$, which after 9 hops inflates the gradient mean square by $128^9 \approx 10^{19}$ (**exploding gradients**).
- With sigmoid activations ($\sigma'(z) \le 1/4$, so $\mathbb{E}[\sigma'(z)^2] \le 1/16$): even if $h\tau^2 = 1$, every layer shrinks the gradient mean square by at least a factor of 16 ((Bengio et al. 1994)).

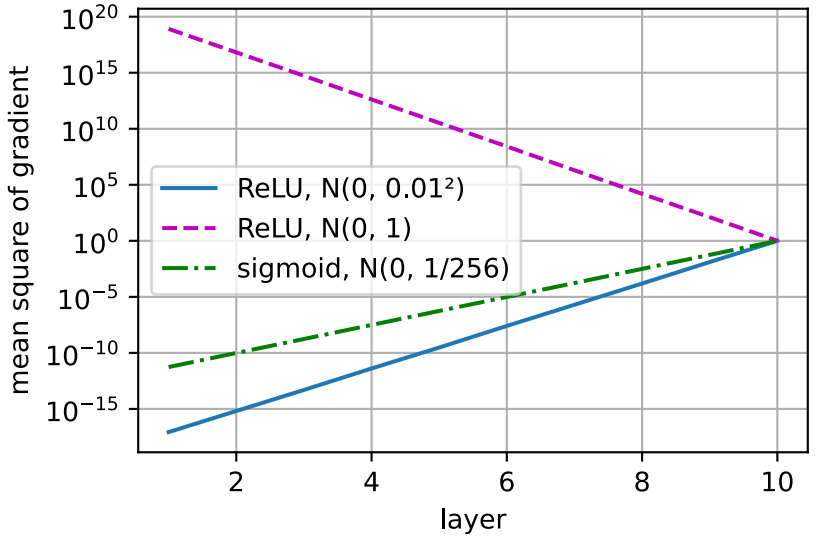
Let us verify these three predictions directly on a 10-layer, width-256 network at initialization using `moments`:

```
def moments(phi, dphi, scale, depth=10, width=256, n=256):
    """Mean squares of the activations and of their gradients, per layer."""
    Ws = [scale * torch.randn(width, width) for _ in range(depth)]
    H, Zs, acts = torch.randn(n, width), [], []
    for W in Ws:  # forward pass, keeping the pre-activations
        Zs.append(H @ W)
        H = phi(Zs[-1])
        acts.append(float((H ** 2).mean()))
    G, grads = torch.randn(n, width), []  # dL/dH at the top layer
    for W, Z in zip(reversed(Ws), reversed(Zs)):  # backward pass
        grads.insert(0, float((G ** 2).mean()))
        G = (G * dphi(Z)) @ W.T  # dL/dH one layer down
    return acts, grads

relu = (torch.relu, lambda Z: (Z > 0).float())
sigmoid = (torch.sigmoid, lambda Z: torch.sigmoid(Z) * (1 - torch.sigmoid(Z)))
```

Let us compare three setups: ReLU with $\tau = 0.01$ (as in Section 4.2), ReLU with $\tau = 1$, and sigmoid with $\tau^2 = 1/256$ (so $h\tau^2 = 1$, isolating pure activation attenuation).

```
runs = {'ReLU, N(0, 0.01²)': moments(*relu, 0.01),
        'ReLU, N(0, 1)': moments(*relu, 1.0),
        'sigmoid, N(0, 1/256)': moments(*sigmoid, 1 / 16)}
for name, (acts, grads) in runs.items():
    print(f'{name}: gradient mean square at layer 1: {grads[0]:.1e}')
d2l.plot(list(range(1, 11)), [grads for acts, grads in runs.values()],
         'layer', 'mean square of gradient', legend=list(runs),
         yscale='log', figsize=(4.5, 3))
```

```
ReLU, N(0, 0.01²): gradient mean square at layer 1: 8.5e-18
ReLU, N(0, 1): gradient mean square at layer 1: 8.2e+18
sigmoid, N(0, 1/256): gradient mean square at layer 1: 5.5e-12
```



On a logarithmic vertical axis, all three curves are straight lines whose slopes match our analytical multiplier Equation 4.14 (10^{-17}, 10^{19}, and 5×10^{-12} at layer 1).

4.4.2 Parameter Initialization

4.4.2.1 Xavier Initialization

How do we choose the initial weight variance τ^2 so that every layer's multiplier equals 1? Consider a linear layer $o_j = \sum_{i=1}^{n_{\text{in}}} x_i w_{ij}$ with n_{in} inputs (the layer's *fan-in*) and n_{out} outputs (its *fan-out*), where weights w_{ij} are i.i.d. with mean 0 and variance τ^2 and inputs x_i are uncorrelated with mean 0 and variance γ^2. In the forward pass, $\mathbb{E}[o_j] = 0$ and

$$\text{Var}[o_j] = \mathbb{E}[o_j^2] = \sum_{i=1}^{n_{\text{in}}} \mathbb{E}[x_i^2]\,\mathbb{E}[w_{ij}^2] = n_{\text{in}}\,\tau^2\gamma^2.$$

Preserving forward variance across the layer requires $n_{\text{in}}\tau^2 = 1$. By Equation 4.14 with unit slope, preserving backward gradient variance requires $n_{\text{out}}\tau^2 = 1$. When $n_{\text{in}} \neq n_{\text{out}}$, *Xavier initialization* (or *Glorot initialization*, (Glorot and Bengio 2010)) averages the two fan counts:

$$\frac{1}{2}(n_{\text{in}} + n_{\text{out}})\,\tau^2 = 1 \quad \Longleftrightarrow \quad \tau^2 = \frac{2}{n_{\text{in}} + n_{\text{out}}}, \tag{4.15}$$

drawing weights either from $\mathcal{N}\left(0, \frac{2}{n_{\text{in}}+n_{\text{out}}}\right)$ or from the variance-matched uniform interval $U\left(-\sqrt{\frac{6}{n_{\text{in}}+n_{\text{out}}}}, \sqrt{\frac{6}{n_{\text{in}}+n_{\text{out}}}}\right)$. Xavier initialization is the standard choice for linear and zero-centered unit-slope activations such as tanh.

4.4.2.2 He Initialization

For ReLU layers, zeroing out the negative half of a symmetric zero-mean pre-activation z cuts its mean square in half: $\mathbb{E}[\mathrm{ReLU}(z)^2] = \frac{1}{2}\mathbb{E}[z^2]$. Because $\mathbb{E}[o_j^2] = n_{\text{in}}\tau^2\,\mathbb{E}[x_i^2]$ holds for zero-mean independent weights even when the inputs $x_i = \mathrm{ReLU}(z_i)$ have positive mean, the activation mean square $v^{(k)} = \mathbb{E}[(h_i^{(k)})^2]$ evolves across a ReLU layer as

$$v^{(k)} = \frac{1}{2}\, n_{\text{in}}\, \tau^2\, v^{(k-1)}. \tag{4.16}$$

Keeping $v^{(k)} = v^{(k-1)}$ constant across depth therefore requires doubling the variance:

$$\tau^2 = \frac{2}{n_{\text{in}}}. \tag{4.17}$$

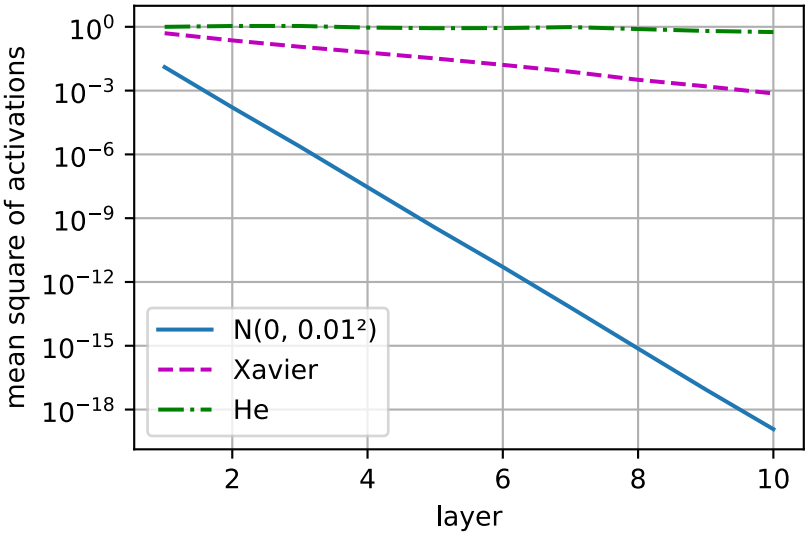
This is *He initialization* (or *Kaiming initialization*, (He et al. 2015)), the default for ReLU networks (note that standard framework layers often default to $1/(3n_{\text{in}})$, $1/n_{\text{in}}$, or Xavier unless He is requested explicitly).

4.4.2.3 Activations, Gradients, and Training across Depth

Let us first compare $\mathcal{N}(0, 0.01^2)$, Xavier ($\tau^2 = 1/256$), and He ($\tau^2 = 2/256$) across our 10-layer synthetic ReLU probe `moments(*relu, ...)`:

```
runs = {'N(0, 0.01²)': moments(*relu, 0.01),
        'Xavier': moments(*relu, (1 / 256) ** 0.5),
        'He': moments(*relu, (2 / 256) ** 0.5)}
for name, (acts, grads) in runs.items():
    print(f'{name}: activations at layer 10: {acts[-1]:.1e}, '
          f'gradient at layer 1: {grads[0]:.1e}')
d2l.plot(list(range(1, 11)), [acts for acts, grads in runs.values()],
         'layer', 'mean square of activations', legend=list(runs),
         yscale='log', figsize=(4.5, 3))
```

```
N(0, 0.01²): activations at layer 10: 1.2e-19, gradient at layer 1: 1.2e-17
Xavier: activations at layer 10: 7.4e-04, gradient at layer 1: 1.9e-03
He: activations at layer 10: 5.7e-01, gradient at layer 1: 8.3e-01
```



Notice how $\mathcal{N}(0, 0.01^2)$ crushes layer-10 activations to $\sim 10^{-19}$ and Xavier halves the activation mean square at every ReLU layer ($\sim 2^{-10} \approx 10^{-3}$ by layer 10), whereas He initialization keeps both activations and gradients near $O(1)$. To see how this variance propagation governs real training, we now define a 10-hidden-layer ReLU classifier `DeepMLP` parameterized by a per-layer standard-deviation rule `std(n_in, n_out)`:

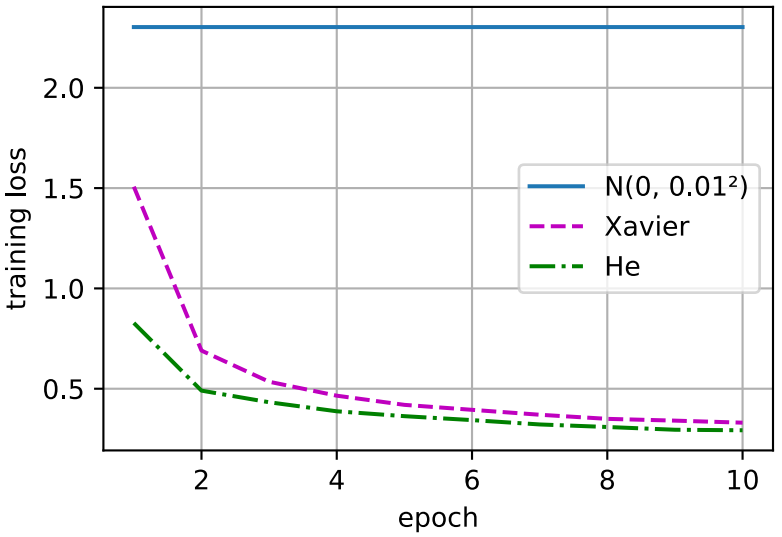
```
class DeepMLP(d2l.Classifier):
    """A deep MLP whose weights have the standard deviation std(n_in, n_out)."""
    def __init__(self, std, lr, depth=10, width=256):
        super().__init__()
        self.save_hyperparameters(ignore=['std'])
        sizes = [784] + [width] * depth + [10]
        layers = [nn.Flatten()]
        for n_in, n_out in zip(sizes[:-1], sizes[1:]):
            layer = nn.Linear(n_in, n_out)
            nn.init.normal_(layer.weight, std=std(n_in, n_out))
            nn.init.zeros_(layer.bias)
            layers += [layer, nn.ReLU()]
        self.net = nn.Sequential(*layers[:-1])  # No ReLU after the logits
```

All three runs share the exact same architecture and hyperparameters, varying only `std(n_in, n_out)`. Let us compare their training curves and final validation accuracies.

```
inits = {'N(0, 0.01²)': lambda n_in, n_out: 0.01,
         'Xavier': lambda n_in, n_out: (2 / (n_in + n_out)) ** 0.5,
         'He': lambda n_in, n_out: (2 / n_in) ** 0.5}
data = d2l.FashionMNIST(batch_size=256)
losses = {}
for name, std in inits.items():
    trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
    trainer.board.display = False
    trainer.fit(DeepMLP(std, lr=0.05), data)
```

```
    losses[name] = trainer.history['train_loss']
    print(f'{name}: validation accuracy {trainer.history["val_acc"][-1]:.2f}')
d2l.plot(list(range(1, 11)), list(losses.values()), 'epoch', 'training loss',
         legend=list(losses), figsize=(4.5, 3))
```

```
N(0, 0.01²): validation accuracy 0.10
Xavier: validation accuracy 0.84
He: validation accuracy 0.87
```



Look at how closely the 10-epoch training curves follow our variance analysis:

- Under $\mathcal{N}(0, 0.01^2)$, layer-10 activations ($\sim 10^{-19}$) and layer-1 gradients ($\sim 10^{-17}$) vanish completely, leaving the training loss frozen at $\log 10 \approx 2.30$ and validation accuracy at 10%.
- Under Xavier, losing a factor of 2 per ReLU layer shrinks signals by $2^{-10} \approx 10^{-3}$, causing a slower first epoch before training recovers to above 80% accuracy.
- Under He initialization, both activations and gradients stay near order 1 across all 10 layers and training drops smoothly from the very first epoch.

Initialization is not the whole answer to depth, though. The variance argument describes the network at step 0 only: once SGD has moved the weights, nothing keeps the per-layer factor of Equation 4.14 near 1. Networks with dozens or hundreds of layers therefore rely on mechanisms that act throughout training. Normalization layers (Section 7.3) rescale the activations at every step, residual connections (Section 7.4) give signals and gradients a path around each layer, and gradient clipping (Section 8.4) caps the occasional exploding gradient. A good initialization gets training started. These mechanisms keep the signals scaled afterwards.

4.4.3 Symmetry among Hidden Units

Why must initial weights be *random* rather than setting every entry of $\mathbf{W}^{(1)}$ to the deterministic constant $c = \sqrt{2/n_{\text{in}}}$? Because hidden units within a layer are permutation-symmetric:

if all columns of $\mathbf{W}^{(1)}$ start identical (and all rows of $\mathbf{W}^{(2)}$ start identical), then every hidden unit computes the same forward activation, receives the same backward gradient, and takes the same SGD update step forever, collapsing a 256-unit layer into a single effective neuron. Let us verify on a 16-unit layer that constant He-scaled weights give identical gradient columns (`0.00` difference between units), whereas random He-scaled weights give every unit a distinct gradient direction:

```
def hidden_grad(W1, W2, X, y):
    """Gradient of the squared loss with respect to the hidden weights."""
    W1 = W1.clone().requires_grad_(True)
    L = ((torch.relu(X @ W1) @ W2 - y) ** 2).mean() / 2
    L.backward()
    return W1.grad

d, h = 20, 16
X, y = torch.randn(64, d), torch.randn(64, 1)
c, c_out = (2 / d) ** 0.5, (1 / h) ** 0.5
inits = {'constant': (torch.full((d, h), c), torch.full((h, 1), c_out)),
         'random': (c * torch.randn(d, h), c_out * torch.randn(h, 1))}
for name, (W1, W2) in inits.items():
    G = hidden_grad(W1, W2, X, y)
    print(f'{name}: largest difference between two hidden units '
          f'{float((G - G[:, :1]).abs().max()):.2f}')
```

```
constant: largest difference between two hidden units 0.00
random: largest difference between two hidden units 0.16
```

4.4.4 Summary

Across a K-layer MLP of width h and initial weight variance τ^2, each layer multiplies activation and gradient mean squares by roughly $h\tau^2\,\mathbb{E}[\phi'(z)^2]$, causing signals to vanish or explode exponentially with depth unless that factor stays close to 1. Xavier initialization sets $\tau^2 = 2/(n_{\text{in}} + n_{\text{out}})$ to balance forward and backward variance in linear and tanh layers, while He initialization doubles the variance to $\tau^2 = 2/n_{\text{in}}$ to offset ReLU zeroing out its negative half-line, allowing deep ReLU stacks to train cleanly from the first epoch. Because hidden units within a layer are permutation-symmetric, weights must also be drawn independently at random rather than set to a constant; otherwise every unit receives identical gradients and updates throughout training. Beyond step 0, deeper architectures pair variance-preserving initialization with normalization layers and residual connections to keep signals well scaled as weights evolve.

4.4.5 Exercises

1. **Spectral norm bounds.** For a single example, written as a column vector, the step Equation 4.12 multiplies the gradient $\mathbf{g}'$ above layer k by the matrix $\mathbf{M}^{(k)} = \mathbf{W}^{(k)}\,\text{diag}(\phi'(\mathbf{z}^{(k)}))$, where $\text{diag}(\cdot)$ places the entries of a vector on the diagonal of an otherwise zero matrix, so the gradient in Equation 4.13 is a product of $K-1-k$ such matrices times the gradient at the top. The spectral norm (the largest singular value) is submultiplicative: $\|\mathbf{AB}\|_2 \le \|\mathbf{A}\|_2\|\mathbf{B}\|_2$.
   1. Prove submultiplicativity from the definition $\|\mathbf{A}\|_2 = \max_{\mathbf{x}\neq 0} \|\mathbf{Ax}\|/\|\mathbf{x}\|$.

   2. Use it to bound the norm of the gradient in terms of the norms $\|\mathbf{M}^{(k)}\|_2$ of the factors. Which condition on these norms keeps the bound from growing geometrically with depth?
   3. This bound is only an upper bound. Explain why the condition of the previous part rules out exploding gradients but not vanishing ones, and give two 2×2 matrices that satisfy the condition with equality but whose product has spectral norm strictly smaller than one.
2. **Other symmetries.** Permuting the hidden units is not the only change of parameters that leaves the function of an MLP unchanged. Consider a one-hidden-layer MLP in which hidden unit j has incoming weights $\mathbf{w}_j$, bias b_j, and outgoing weights $\mathbf{v}_j$.
   1. Prove that $\mathrm{ReLU}(\alpha z) = \alpha\,\mathrm{ReLU}(z)$ for $\alpha > 0$. Use this identity to show that replacing $(\mathbf{w}_j, b_j, \mathbf{v}_j)$ by $(\alpha\mathbf{w}_j, \alpha b_j, \mathbf{v}_j/\alpha)$ leaves the function of a ReLU network unchanged.
   2. Find a transformation of the parameters of a single hidden unit that leaves the function of a tanh network unchanged.
   3. Identical hidden units are a fixed point of the permutation, and gradient descent started at that fixed point stays there. Find the parameters of a unit that the transformation of the previous part leaves unchanged. Then show that if all weights and biases of a tanh network are initialized to zero, gradient descent changes only the output bias, at every step.

   Adapted from Simon Prince, Understanding Deep Learning, Problems 3.5, 4.3, and 7.15.
3. **Fan-in versus averaged fans.** Following He et al. (2015), consider a linear network with K layers in which layer k maps n_{k-1} inputs to n_k outputs and draws its weights with variance τ_k^2. By the computation of Section 4.4.2.1, layer k multiplies the variance of the forward signal by $n_{k-1}\tau_k^2$ and that of the backward signal by $n_k\tau_k^2$.
   1. Set $\tau_k^2 = 1/n_{k-1}$ in every layer. Show that the forward variance is preserved and that the backward variance changes by the total factor n_K/n_0 at any depth.
   2. Let the widths alternate as $100, 400, 100, 400, \ldots$, and use Xavier initialization in every layer. Compute the factor by which one $100 \to 400 \to 100$ pair of layers multiplies the forward variance, and the factor after 50 such pairs.
   3. Which of the two rules keeps both variances within a depth-independent factor of their initial values for every sequence of widths?
4. **Variance under ReLU.** ● The derivation in Section 4.4.2.2 tracks the mean square $\mathbb{E}[\mathrm{ReLU}(z)^2]$ rather than the variance, because the next layer's pre-activation $o = \sum_i x_i w_i$, with inputs $x_i = \mathrm{ReLU}(z_i)$ and independent zero-mean weights has $\mathrm{Var}[o] = n_{\text{in}}\tau^2\mathbb{E}[x_i^2]$ for any mean of x_i.
   1. For $z \sim \mathcal{N}(0, \nu^2)$, compute $\mathbb{E}[\mathrm{ReLU}(z)]$ and $\mathrm{Var}[\mathrm{ReLU}(z)]$, and show that the variance equals $\left(\frac{1}{2} - \frac{1}{2\pi}\right)\nu^2$ rather than $\frac{1}{2}\nu^2$.
   2. A tempting shortcut substitutes $\mathrm{Var}[x_i]$ for $\mathbb{E}[x_i^2]$ in the Xavier computation. Show that this shortcut prescribes $\tau^2 = 2/(n_{\text{in}}(1-1/\pi))$ instead of $2/n_{\text{in}}$. Compute the factor by which each ReLU layer initialized this way actually multiplies the mean square, and the total factor over fifty layers.
5. code **Tanh networks.** ● Repeat the measurement of Section 4.4.2.3 with tanh in place of ReLU, with Xavier initialization and fifty layers. A layer then maps the mean square m of its input to $m' = \mathbb{E}[\tanh(z)^2]$, where z is approximately $\mathcal{N}(0, m)$.
   1. Use $\tanh(z)^2 \approx z^2 - \frac{2}{3}z^4$ for small z to show that $m' \approx m - 2m^2$. Over many layers, does the mean square stay constant, decay geometrically, or decay more slowly? Show that the mean square at layer k is about $1/(2k)$ for large k.
   2. Check the prediction by plotting the measured mean square against the layer on

logarithmic axes. What happens to the gradient?

4.5 Generalization in Deep Learning

In Section 2.6, a degree-19 polynomial with 20 parameters interpolated 20 noisy data points and blew up on test data. By that classical intuition, our Fashion-MNIST MLP in Section 4.2, with $203{,}530$ parameters for $60{,}000$ training images (3.4 parameters per image), ought to overfit severely, especially if trained until it interpolates every single training image. Yet it reached $\sim 87\%$ accuracy on unseen images. And as we will see below, the same architecture given only $1{,}000$ training images and driven to 100% training accuracy still classifies about 80% of unseen images correctly.

Understanding why heavily overparameterized networks generalize is one of the central open problems of modern machine learning. In this section, we combine controlled experiments with exact results on tractable models to uncover four key pieces of the puzzle:

1. **Random-label memorization:** Standard MLPs easily reach 100% training accuracy on completely random labels (Zhang et al. 2021), proving that classical distribution-free capacity bounds (like VC dimension) cannot explain their generalization on real labels.
2. **Double descent:** Sweeping model width past the interpolation threshold $h = n$ reveals a spike in test error at $h = n$ (where the feature matrix is nearly singular) followed by a *second descent* as extra parameters allow smaller-norm interpolators (Belkin et al. 2019).
3. **Implicit bias of gradient descent:** Among infinitely many zero-loss solutions, gradient descent automatically selects the minimum-norm interpolator under squared loss and the maximum-margin separator under logistic loss (Soudry et al. 2018).
4. **Clean-first learning and early stopping:** On noisy datasets, gradient descent fits clean structured patterns long before memorizing corrupted labels (Arpit et al. 2017), making *early stopping* a potent regularizer.

4.5.1 Memorization and Capacity

4.5.1.1 Random Labels

To test raw memorization capacity following Zhang et al. (2021), we draw $2{,}000$ Fashion-MNIST training images ($1{,}000$ for training, $1{,}000$ for validation) and $2{,}000$ test images, define `ArrayData` and `Tracker` helpers along with a He-initialized 256-unit `MLP`, and train for 200 epochs once on the true labels (`fraction=0`) and once on completely random labels (`fraction=1`):

```
data = d2l.FashionMNIST(batch_size=2000)
X, y = next(iter(data.train_dataloader()))          # 2,000 random images
X_test, y_test = next(iter(data.val_dataloader()))  # the first 2,000
X, y, X_test, y_test = (d2l.numpy(a) for a in (X, y, X_test, y_test))
X, X_test = X.reshape(2000, -1), X_test.reshape(2000, -1)
X_train, y_train, X_val, y_val = X[:1000], y[:1000], X[1000:], y[1000:]
```

Two lightweight helpers streamline the experiments ahead: `ArrayData` wraps NumPy arrays into shuffled minibatches, and `Tracker` subclasses `Trainer` to log epoch-by-epoch accuracy across an arbitrary dictionary of named datasets.

```
class ArrayData(d2l.DataModule):
    """Minibatches of the arrays X and y; training batches are shuffled."""
    def __init__(self, X, y, batch_size=64):
        super().__init__()
        self.save_hyperparameters()

    def get_dataloader(self, train):
        return self.get_tensorloader(
            (d2l.tensor(self.X), d2l.tensor(self.y)), train)

class Tracker(d2l.Trainer):
    """A Trainer that records the accuracy on named datasets every epoch."""
    def __init__(self, max_epochs, datasets):
        super().__init__(max_epochs, num_gpus=1)
        self.datasets = datasets
        self.accuracy = {name: [] for name in datasets}
        self.board.display = False  # We plot the accuracies ourselves

    def validate(self):
        for name, dataset in self.datasets.items():
            self.val_dataloader = dataset.val_dataloader()
            self.accuracy[name].append(super().validate()['acc'])
        return {}
```

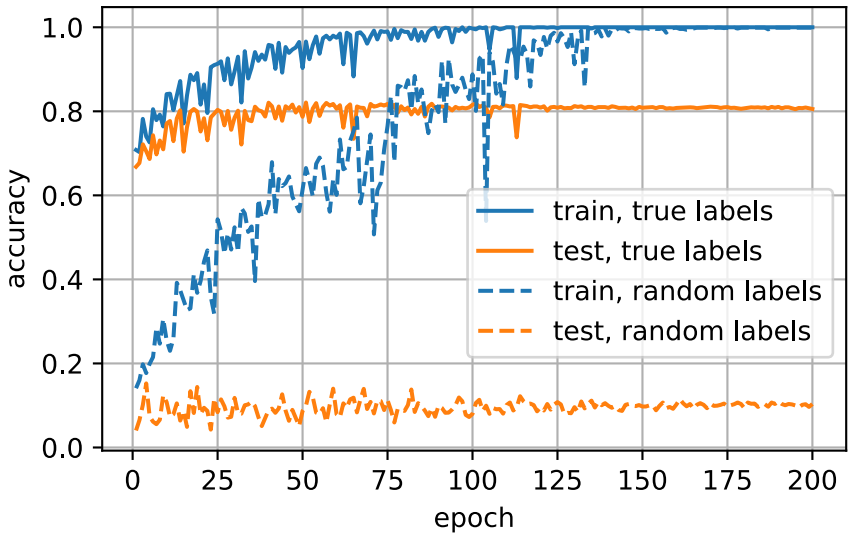
Our model is the same 256-unit ReLU MLP from Section 4.2, upgraded with He initialization (Section 4.4.2.2) in place of `std=0.01` so training takes off immediately from epoch 1 (with zero biases and pre-flattened inputs).

```
class MLP(d2l.Classifier):
    def __init__(self, num_hiddens, lr):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(nn.LazyLinear(num_hiddens), nn.ReLU(),
                                 nn.LazyLinear(10))

    def weight_init(self, module):
        if isinstance(module, nn.Linear):
            nn.init.kaiming_normal_(module.weight, nonlinearity='relu')
            nn.init.zeros_(module.bias)
```

Now we define `corrupt(y, fraction, rng)` to replace each training label with a uniform random class in $\{0, \ldots, 9\}$ with probability `fraction`, train `MLP(num_hiddens=256, lr=0.1)` for 200 epochs at `fraction = 0` (true labels) and `fraction = 1` (pure random labels), and plot both training and test accuracy trajectories:

```
def corrupt(y, fraction, rng):
    """Replace each label, with probability fraction, by a random class."""
    redraw = rng.random(len(y)) < fraction
    y = y.copy()
    y[redraw] = rng.integers(0, 10, redraw.sum())
    return y
```

```
rng = np.random.default_rng()
curves = []
for fraction in (0, 1):
    labels = corrupt(y_train, fraction, rng)
    trainer = Tracker(max_epochs=200, datasets={
        'train': ArrayData(X_train, labels, 1000),
        'test': ArrayData(X_test, y_test, 1000)})
    trainer.fit(MLP(num_hiddens=256, lr=0.1), ArrayData(X_train, labels))
    curves += [trainer.accuracy['train'], trainer.accuracy['test']]
d2l.plot(list(range(1, 201)), curves, 'epoch', 'accuracy',
         fmts=('C0-', 'C1-', 'C0--', 'C1--'), figsize=(5, 3))
d2l.plt.legend(['train, true labels', 'test, true labels',
                'train, random labels', 'test, random labels'],
               loc='center right', bbox_to_anchor=(1, 0.4));
```



Look at what the two runs prove:

- On **true labels**, training accuracy hits 100% within 60–90 epochs while test accuracy plateaus above 80% (even on just $1{,}000$ training images).
- On **completely random labels**, the exact same architecture and optimizer still reach 100% training accuracy (taking slightly longer, $\sim$ 120–150 epochs) while test accuracy remains at 10% random chance.

4.5.1.2 Limits of Capacity Bounds and Inductive Bias

Because our MLP achieves zero training error ($\epsilon_{\mathcal{S}}(f) = 0$) on random labels where the true population error is $\epsilon(f) = 0.90$, *any* distribution-free bound of the form $\epsilon(f) \leq \epsilon_{\mathcal{S}}(f) + B(\mathcal{F}, n, \delta)$ that depends only on the hypothesis class $\mathcal{F}$ must have $B(\mathcal{F}, n, \delta) \geq 0.90$, making it vacuous on real labels as well (Exercise 1).

By the *no free lunch theorem* (Wolpert 1996), no learner can generalize across all distributions; generalization depends on whether the learner's *inductive bias*—which interpolating

function it prefers among infinitely many—matches the structure of real data, just as a 1-nearest-neighbor classifier interpolates 100% of its training data yet generalizes well whenever nearby inputs share labels (Cover and Hart 1967).

4.5.2 Double Descent

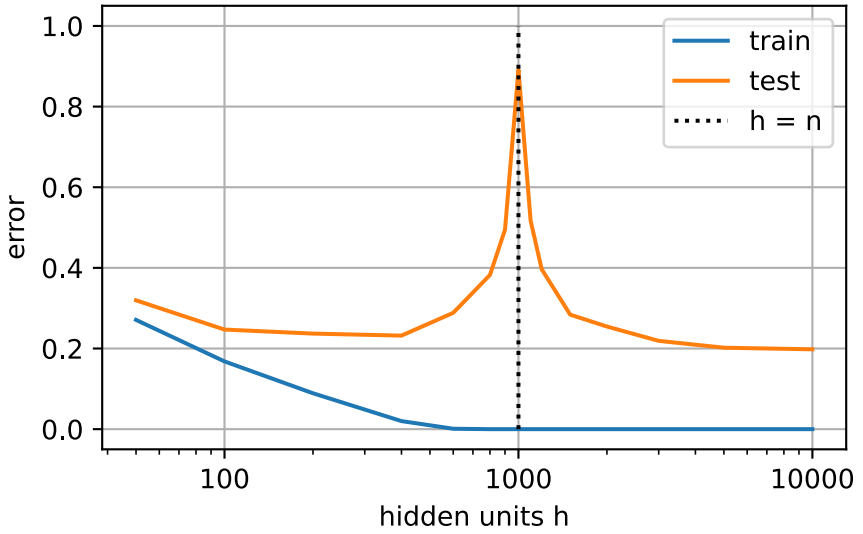
4.5.2.1 The Interpolation Threshold and Conditioning

What happens when we increase model width h right past the sample size n? Let us test a tractable *random-features* model (Rahimi and Recht 2007): we freeze a He-initialized first layer $\mathbf{W}^{(1)} \in \mathbb{R}^{784\times 10000}$ on our $n = 1{,}000$ centered training images to precompute $10{,}000$ random ReLU features `H` and `H_test`, encode the labels as a one-hot matrix `Y`, and define `error(H, W2, y)`:

```
W1 = rng.normal(0, np.sqrt(2 / 784), (784, 10000))  # He-initialized, frozen
mean = X_train.mean(axis=0)                         # The mean training image
H = np.maximum((X_train - mean) @ W1, 0)
H_test = np.maximum((X_test - mean) @ W1, 0)
Y = np.eye(10)[y_train]                             # One-hot labels

def error(H, W2, y):
    """Fraction of examples whose largest output is not their label."""
    return (np.argmax(H @ W2, axis=1) != y).mean()
```

We now sweep the hidden width `h` across 15 values from 50 to $10{,}000$, fitting the output weights $\mathbf{W}^{(2)} \in \mathbb{R}^{h\times 10}$ on the first `h` feature columns `H[:, :h]` by minimum-norm least squares (`np.linalg.lstsq`) while recording the smallest singular value `s_min[h]` and Frobenius norm `norm[h]`:

```
widths = [50, 100, 200, 400, 600, 800, 900, 1000, 1100, 1200, 1500, 2000,
          3000, 5000, 10000]
train_err, test_err, s_min, norm = [], [], {}, {}
for h in widths:
    W2, _, _, s = np.linalg.lstsq(H[:, :h], Y, rcond=None)
    train_err.append(error(H[:, :h], W2, y_train))
    test_err.append(error(H_test[:, :h], W2, y_test))
    s_min[h], norm[h] = s.min(), np.linalg.norm(W2)
d2l.plot([widths, widths, [1000, 1000]], [train_err, test_err, [0, 1]],
         'hidden units h', 'error', legend=['train', 'test', 'h = n'],
         xscale='log', fmts=('C0-', 'C1-', 'k:'), figsize=(5, 3))
```



This is the celebrated *double-descent* curve (Belkin et al. 2019):

1. **Classical U-curve ($h < n$):** Test error drops from $\sim 30\%$ at $h = 50$ to $\sim 23\%$ around $h = 200$, then shoots up to $\sim 90\%$ right at the *interpolation threshold* $h = n = 1{,}000$.
2. **Second descent ($h > n$):** As h grows past $n = 1{,}000$ up to $10{,}000$, every model interpolates the training set, yet test error plunges back down to $\sim 20\%$, beating the best underparameterized model.

Why does test error spike so violently at $h = n$ and heal for $h \gg n$? Let us inspect the smallest singular value $s_{\min}$ of $\mathbf{H}_{:,:h}$ and the Frobenius norm $\|\mathbf{W}^{(2)}\|_{\mathrm{F}}$:

```
for h in (200, 900, 1000, 1100, 10000):
    print(f'h = {h:5d}: smallest singular value {s_min[h]:.1e}, '
          f'norm of W2 {norm[h]:6.1f}')
```

```
h =   200: smallest singular value 1.7e+00, norm of W2    3.7
h =   900: smallest singular value 1.2e-01, norm of W2   20.4
h =  1000: smallest singular value 2.0e-04, norm of W2 1577.9
h =  1100: smallest singular value 2.3e-02, norm of W2   20.4
h = 10000: smallest singular value 2.1e-01, norm of W2    2.2
```

At $h = n = 1{,}000$, $\mathbf{H}$ is a square 1000×1000 matrix with a tiny smallest singular value $s_{\min}$, so the unique exact inverse $\mathbf{H}^{-1}\mathbf{Y}$ divides label noise by $s_{\min}$ and explodes $\|\mathbf{W}^{(2)}\|_{\mathrm{F}}$ into the hundreds or more. For $h \gg n$, the extra columns open up an $(h-n)$-dimensional subspace of exact interpolators, allowing the *minimum-norm* solution to avoid poorly conditioned directions and shrink $\|\mathbf{W}^{(2)}\|_{\mathrm{F}}$ all the way down to ~ 2. In fact, holding $h = 1{,}000$ fixed, doubling the training set from $n = 500$ to $n = 1{,}000$ lands the unregularized fit squarely on the $h = n$ cliff and drives test error *up* from $\sim 30\%$ to 85–90%:

```
for n in (500, 1000):
    W2 = np.linalg.lstsq(H[:n, :1000], Y[:n], rcond=None)[0]
    test_error = error(H_test[:, :1000], W2, y_test)
```

```
    print(f'{n} training images: test error {test_error:.2f}')
```

```
500 training images: test error 0.28
1000 training images: test error 0.89
```

4.5.3 Implicit Bias of Gradient Descent

The second descent above relied on `lstsq` picking the *minimum-norm* interpolator among infinitely many zero-loss solutions. Remarkably, gradient descent has that exact preference built in—an *implicit bias* (or *implicit regularization*) that operates even without any explicit penalty term.

4.5.3.1 Minimum-Norm Interpolation under Squared Loss

For an overparameterized linear system $\mathbf{H}\mathbf{w} = \mathbf{y}$ ($h > n$, full row rank), the unique minimum-norm interpolator is $\mathbf{w}^\star = \mathbf{H}^\top(\mathbf{H}\mathbf{H}^\top)^{-1}\mathbf{y}$.

Proposition. Gradient descent on $L(\mathbf{w}) = \frac{1}{2}\|\mathbf{H}\mathbf{w} - \mathbf{y}\|_2^2$ initialized at $\mathbf{w}_0 = \mathbf{0}$ (with a converging step size) converges to the minimum-norm interpolator $\mathbf{w}^\star$.

Proof. Every gradient $\nabla L(\mathbf{w}) = \mathbf{H}^\top(\mathbf{H}\mathbf{w} - \mathbf{y})$ lies in the row space of $\mathbf{H}$. Starting at $\mathbf{w}_0 = \mathbf{0}$, every iterate $\mathbf{w}_t$, and hence the limit $\mathbf{w}_\infty = \mathbf{H}^\top\mathbf{a}$, remains in the row space of $\mathbf{H}$ (Figure 4.12). Since $\mathbf{H}\mathbf{w}_\infty = \mathbf{H}\mathbf{H}^\top\mathbf{a} = \mathbf{y}$, we have $\mathbf{a} = (\mathbf{H}\mathbf{H}^\top)^{-1}\mathbf{y}$ and $\mathbf{w}_\infty = \mathbf{w}^\star$. Any other interpolator $\mathbf{w}^\star + \mathbf{v}$ has $\mathbf{H}\mathbf{v} = \mathbf{0}$ (so $\mathbf{v} \perp \mathbf{w}^\star$) and strictly larger norm $\|\mathbf{w}^\star\|_2^2 + \|\mathbf{v}\|_2^2$. ■

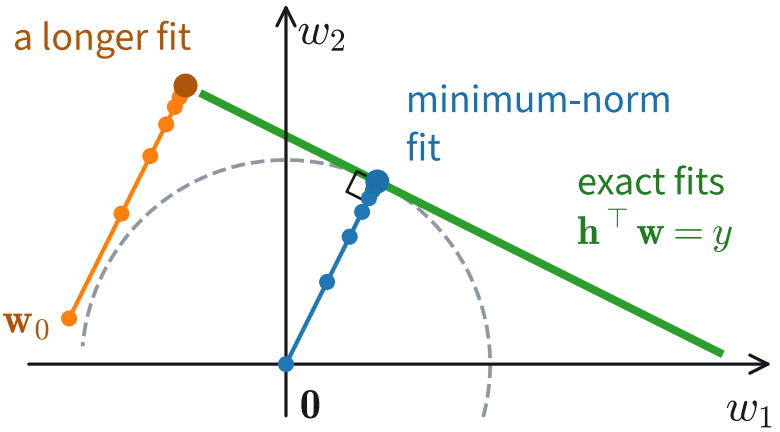


Figure 4.12: Gradient descent on an underdetermined linear equation $\mathbf{h}^\top\mathbf{w} = y$ moves strictly perpendicular to the solution hyperplane, so starting from $\mathbf{0}$ (blue) lands at the minimum-norm solution $\mathbf{w}^\star$, whereas starting from $\mathbf{w}_0 \neq \mathbf{0}$ (orange) preserves the null-space component of $\mathbf{w}_0$.

Let us verify this proposition numerically on 20 equations in 100 unknowns, comparing $\mathbf{w}_0 = \mathbf{0}$ against a random initialization:

```
H_small = rng.normal(size=(20, 100))
y_small = rng.normal(size=20)
w_star = np.linalg.lstsq(H_small, y_small, rcond=None)[0]
lr = 1 / np.linalg.norm(H_small, 2) ** 2  # Below 2 / largest curvature
for name, w in (('zero', np.zeros(100)), ('random', rng.normal(size=100))):
    for _ in range(2000):
        w = w - lr * H_small.T @ (H_small @ w - y_small)
    residual = np.linalg.norm(H_small @ w - y_small)
    print(f'start {name:6s}: residual {residual:.0e}, norm '
          f'{np.linalg.norm(w):5.2f}, distance to w* '
          f'{np.linalg.norm(w - w_star):.1f}')
```

```
start zero  : residual 8e-17, norm  0.79, distance to w* 0.0
start random: residual 3e-15, norm  9.39, distance to w* 9.4
```

4.5.3.2 Maximum-Margin Classification under Logistic Loss

On linearly separable binary classification data ($y_i \in \{-1, +1\}$) under the logistic (cross-entropy) loss $\sum_i \log(1 + \exp(-y_i \mathbf{w}^\top \mathbf{x}_i))$, $\|\mathbf{w}_t\|_2 \to \infty$ as $t \to \infty$, and Soudry et al. (2018) proved that the *direction* $\mathbf{w}_t / \|\mathbf{w}_t\|_2$ converges (at a $1/\log t$ rate) to the hard-margin SVM *maximum-margin separator* (Figure 4.13), because support vectors with the smallest margins dominate the exponentially decaying tail of $\log(1 + e^{-m})$ (see Exercise 5 for how layer rescaling complicates parameter-norm margins in multi-layer ReLU networks).

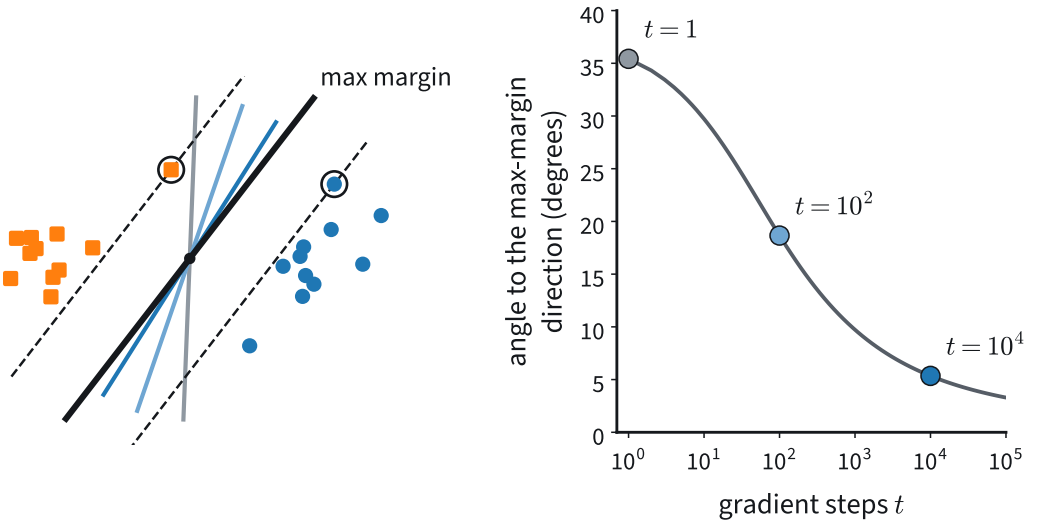


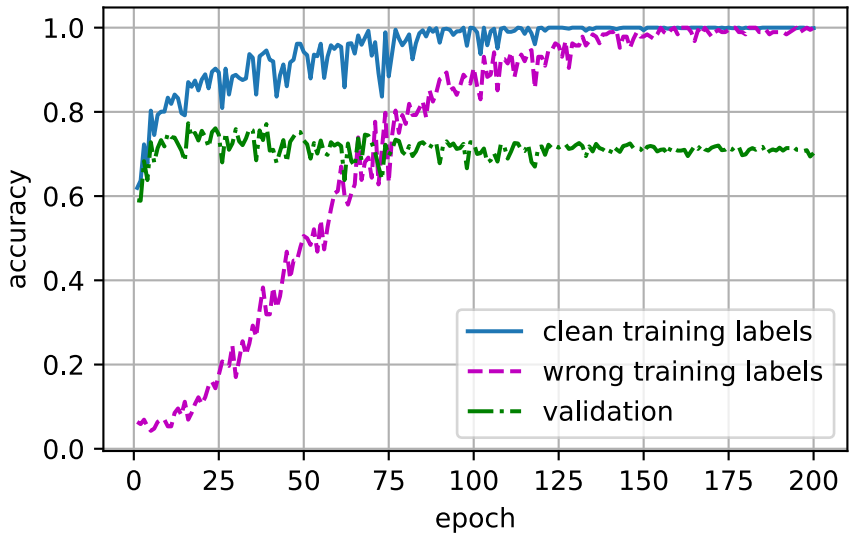
Figure 4.13: Gradient descent on linearly separable logistic regression rotates $\mathbf{w}_t / \|\mathbf{w}_t\|_2$ toward the maximum-margin hyperplane (left) at a logarithmic rate that continues long after training accuracy reaches 100% (right).

4.5.4 Early Stopping

Even when an MLP has enough capacity to memorize corrupted labels, gradient descent learns *clean structured patterns first* and memorizes *label noise much later* (Arpit et al. 2017).

Let us corrupt 20% of the labels in our $1{,}000$-image training set and track accuracy separately on the clean training examples, the mislabeled training examples (scored against their corrupted labels), and the $1{,}000$ clean validation images over 200 epochs:

```
labels = corrupt(y_train, 0.2, rng)
wrong = labels != y_train
trainer = Tracker(max_epochs=200, datasets={
    'clean': ArrayData(X_train[~wrong], labels[~wrong], 1000),
    'mislabeled': ArrayData(X_train[wrong], labels[wrong], 1000),
    'validation': ArrayData(X_val, y_val, 1000),
    'test': ArrayData(X_test, y_test, 1000)})
trainer.fit(MLP(num_hiddens=256, lr=0.1), ArrayData(X_train, labels))
acc = trainer.accuracy
d2l.plot(list(range(1, 201)), [acc['clean'], acc['mislabeled'],
                               acc['validation']], 'epoch', 'accuracy',
         legend=['clean training labels', 'wrong training labels',
                 'validation'], figsize=(5, 3))
```



By epoch 20, the MLP gets over 80% of the clean training images right while memorizing fewer than 20% of the corrupted labels, and validation accuracy reaches its peak in this early window before late-epoch noise memorization drags it down. *Early stopping*, halting training once validation performance fails to improve for a specified *patience* window (Exercise 2) and restoring the best validation checkpoint, recovers several percentage points of test accuracy while saving compute (and in linear least squares acts analogously to ℓ_2 weight decay; see Exercise 4):

```
best = int(np.argmax(acc['validation']))
print(f'best validation accuracy after epoch {best + 1}: '
      f'test accuracy {acc["test"][best]:.3f}')
print(f'after all 200 epochs: test accuracy {acc["test"][-1]:.3f}')
```

```
best validation accuracy after epoch 16: test accuracy 0.775
```

```
after all 200 epochs: test accuracy 0.716
```

One famous exception to early stopping is *grokking* (Power et al. 2022) (Figure 4.14, Exercise 6), where networks trained with weight decay on small algorithmic tasks (such as modular addition) memorize the training table rapidly while validation accuracy remains near chance for $100\times$ longer before suddenly jumping to 100%.

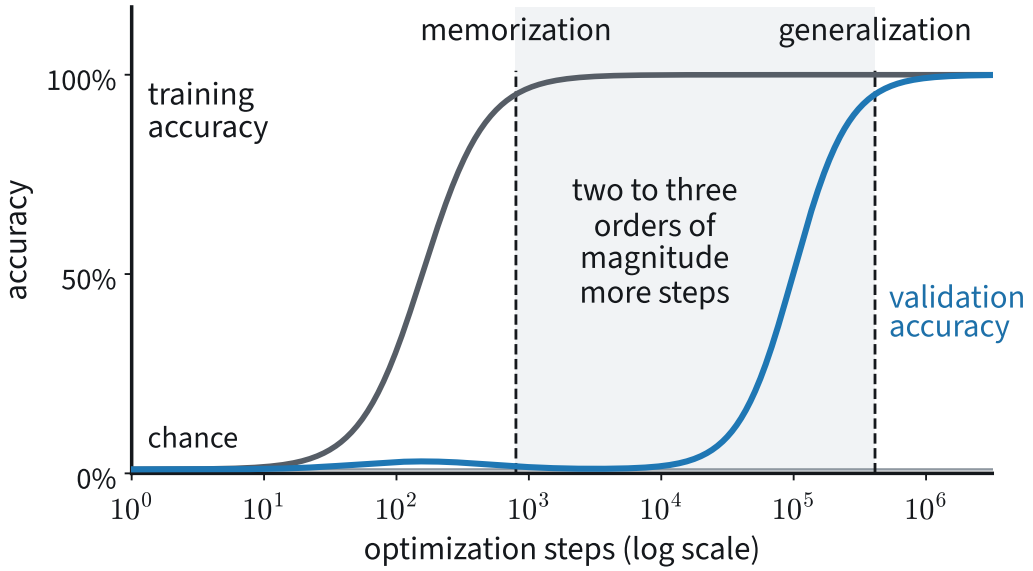


Figure 4.14: Grokking (Power et al. (2022)): on modular arithmetic, training accuracy saturates within $\sim 10^3$ steps while validation accuracy jumps to generalization only after $\sim 10^5$ steps.

4.5.5 Summary

Overparameterized neural networks can memorize completely random labels while generalizing well on structured data, rendering distribution-free capacity bounds vacuous and placing the burden of explanation on inductive bias. In random-features least squares, test error spikes at the interpolation threshold $h = n$, where a tiny smallest singular value $s_{\min}(\mathbf{H})$ inflates $\|\mathbf{W}^{(2)}\|$, and descends a second time for $h \gg n$ as minimum-norm interpolation spreads weights across redundant features. Remarkably, gradient descent from zero finds this minimum-norm interpolator automatically under squared loss and converges in direction to the maximum-margin separator under logistic loss. Because gradient descent also fits coherent patterns before memorizing corrupted labels, early stopping on a validation set serves as a simple and effective regularizer.

4.5.6 Exercises

1. **Limits of complexity measures.** Consider a generalization bound that holds for every data distribution: with probability at least $1 - \delta$ over the draw of n training examples, every classifier f in a class $\mathcal{F}$ satisfies

$$\epsilon(f) \le \epsilon_{\mathcal{S}}(f) + B(\mathcal{F}, n, \delta). \tag{4.18}$$

Here $\epsilon(f)$ is the population error, $\epsilon_{\mathcal{S}}(f)$ is the training error, and the term B depends only on $\mathcal{F}$, n, and δ, as the term $c\sqrt{(v + \log(1/\delta))/n}$ of the VC bound does. The text ran the random-label argument against such a bound for 1,000 images.

1. Suppose that training also finds a member of $\mathcal{F}$ with zero training error on all 60,000 Fashion-MNIST training images when their labels are drawn uniformly at random, as Zhang et al. (2021) observed for image classifiers. What lower bound on $B(\mathcal{F}, 60{,}000, \delta)$ follows, and what does the bound then guarantee for the network trained on the true labels?
2. Many later bounds let the second term depend on the classifier that training returns, for example through the norm of its weights or through its margins on the training examples. Which step of the argument no longer goes through? What must such a quantity do on the random-label run if the bound is to say something on the true labels?

2. code **Patience-based stopping.** `FashionMNIST` uses the Fashion-MNIST test set as its validation data, so hold out 10,000 training images as a validation set instead. Train an MLP with one hidden layer of 256 units, as in Section 4.2, on the remaining images for a fixed number of epochs large enough to overfit, and record the validation loss and the test accuracy after every epoch. A patience rule with tolerance θ and patience k stops once the validation loss has failed to improve on its best value by more than θ for k consecutive epochs. It then retains the parameters of the epoch with the lowest validation loss so far.

   1. Fix θ and apply the rule to the recorded curves for $k \in \{1, 5, 20\}$. For each k, report the stopping epoch, the retained epoch, and the test accuracy at both.
   2. How does the test accuracy at the retained epoch depend on k? Relate the dependence to the epoch-to-epoch fluctuation of the validation loss.

   Adapted from Michael Nielsen, Neural Networks and Deep Learning, Chapter 3.

3. code **Weight decay at the threshold.** Replace the minimum-norm fit of the double-descent experiment by the ridge solution $\mathbf{W}^{(2)} = (\frac{1}{n}\mathbf{H}^\top\mathbf{H} + \lambda\mathbf{I})^{-1}\frac{1}{n}\mathbf{H}^\top\mathbf{Y}$ of Section 2.7, which, as there, averages the squared error over the n examples.

   1. Plot the test error against the width for $\lambda \in \{10^{-5}, 10^{-3}, 10^{-1}\}$. What happens to the peak at $h = n$?
   2. Show that the ridge solution approaches the minimum-norm solution as $\lambda \to 0$ for $h > n$. Hint: write $(\frac{1}{n}\mathbf{H}^\top\mathbf{H} + \lambda\mathbf{I})^{-1}\frac{1}{n}\mathbf{H}^\top = \mathbf{H}^\top(\mathbf{H}\mathbf{H}^\top + n\lambda\mathbf{I})^{-1}$.
   3. For each width, choose λ on the 1,000 validation images. Does the curve of the tuned model still descend twice?

4. **Early stopping and weight decay.** ● Early stopping limits the number of training epochs where weight decay limits the weights. Linear least squares makes this precise. Let $\mathbf{X}^\top\mathbf{X}/n$ have eigenvalues $\mu_i > 0$, let $\mathbf{w}^*$ minimize $L(\mathbf{w}) = \frac{1}{2n}\|\mathbf{X}\mathbf{w} - \mathbf{y}\|^2$, and run gradient descent on L from $\mathbf{w}_0 = \mathbf{0}$ with learning rate $\eta < 1/\max_i \mu_i$.

   1. Show that after t steps the coordinate of $\mathbf{w}_t$ along the i-th eigenvector equals $1 - (1 - \eta\mu_i)^t$ times that of $\mathbf{w}^*$.

2. The minimizer of $L(\mathbf{w}) + \frac{\lambda}{2}\|\mathbf{w}\|^2$ scales the same coordinate by $\mu_i/(\mu_i + \lambda)$ (Section 2.7.3). For which λ, as a function of η and t, do the two factors approximately agree both in the directions with $\eta\mu_i t \ll 1$ and in those with $\eta\mu_i t \gg 1$?

Adapted from Ian Goodfellow, Yoshua Bengio, and Aaron Courville, Deep Learning, Section 7.8.

5. **Implicit bias of the logistic loss.** ● For linearly separable data, gradient descent on the logistic loss converges in direction to the maximum-margin separator (Soudry et al. 2018). Take labels $y_i \in \{-1, +1\}$.
   1. For a linear predictor $f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x}$, show that on separable data the loss $\sum_i \log\left(1 + \exp(-y_i \mathbf{w}^\top \mathbf{x}_i)\right)$ is positive for every $\mathbf{w}$ but has infimum zero. Conclude that $\|\mathbf{w}_t\|$ grows without bound when gradient descent drives the loss to zero, so that only the direction $\mathbf{w}_t/\|\mathbf{w}_t\|$ can converge.
   2. For a two-layer ReLU network $f(\mathbf{x}) = \sum_j v_j \max(0, \mathbf{w}_j^\top \mathbf{x})$, show that replacing $(\mathbf{w}_j, v_j)$ by $(c_j \mathbf{w}_j, v_j/c_j)$ with $c_j > 0$ leaves f unchanged. Find the smallest value of $\sum_j (\|\mathbf{w}_j\|^2 + v_j^2)$ over these rescalings. Explain why a margin measured with the parameter norm of the network, unlike one measured with $\|\mathbf{w}\|$ for the linear predictor, is not a property of the function f alone.

6. [extended] **Grokking.** ●● Reproduce the setup of Power et al. (2022): train a small network (they use a two-layer transformer, but a wide MLP on one-hot pairs also works) to predict $c = (a + b) \bmod 97$ from the pair (a, b), using a random 50% of all pairs for training, with weight decay, for 10^5 or more steps. Plot training and validation accuracy against the logarithm of the step count, and compare with Figure 4.14. How does the delay before generalization change with the training fraction and with the weight-decay strength?

4.6 Dropout

When an overparameterized network trains for many epochs on scarce or noisy data, it eventually memorizes training quirks and corrupted labels (Section 4.5). Where weight decay (Section 2.7) penalizes large weights and early stopping (Section 4.5.4) halts training before late-stage memorization, *dropout* (Hinton et al. 2012; Srivastava et al. 2014) regularizes a network by injecting multiplicative noise directly into its hidden layers during training: on every forward pass, each hidden unit is independently zeroed out with probability p, forcing every unit to learn features that work reliably without counting on specific partner units (*co-adaptation*). At test time, dropout is disabled and the full network runs deterministically.

In this section, we formulate inverted dropout masking and rescaling, implement it from scratch and concisely in a two-hidden-layer MLP on Fashion-MNIST, and demonstrate on a heavily corrupted label-noise experiment how dropout slows memorization and how much validation accuracy that saves.

4.6.1 Noise in Hidden Layers

4.6.1.1 Masking and Rescaling (Inverted Dropout)

Injecting random noise into inputs during training encourages smooth, perturbation-resilient functions (Bishop 1995); in fact, for a linear model $\mathbf{w}^\top \mathbf{x} + b$ under squared error,

adding zero-mean independent noise of variance s^2 to the inputs adds $s^2\|\mathbf{w}\|_2^2$ to the expected loss, which is ℓ_2 weight decay. Dropout extends noise injection to hidden layers via random binary masking.

For a hidden activation h_j and *dropout probability* $p \in [0, 1)$, on each training step we draw an independent Bernoulli mask $\xi_j \in \{0, 1\}$ with $P(\xi_j = 0) = p$ and $P(\xi_j = 1) = 1 - p$, and replace h_j with

$$h_j' = \frac{\xi_j h_j}{1-p} = \begin{cases} 0 & \text{with probability } p, \\ \dfrac{h_j}{1-p} & \text{with probability } 1-p. \end{cases} \tag{4.19}$$

Dividing the surviving activations by $1 - p$ during training (*inverted dropout*) keeps the expectation unbiased:

$$\mathbb{E}[h_j'] = p \cdot 0 + (1-p)\,\frac{h_j}{1-p} = h_j, \qquad \mathrm{Var}[h_j'] = \frac{p}{1-p}\,h_j^2.$$

Because $\mathbb{E}[h_j'] = h_j$ is preserved during training, **at inference time we simply turn dropout off ($h_j' = h_j$) without any weight rescaling.** (Note that for a linear regression model with input dropout, the variance formula $\mathrm{Var}[x_j'] = \frac{p}{1-p}x_j^2$ adds an adaptive ridge penalty $\mathbb{E}[(\mathbf{w}^\top\mathbf{x}' + b - y)^2] = (\mathbf{w}^\top\mathbf{x} + b - y)^2 + \frac{p}{1-p}\sum_j x_j^2 w_j^2$ to the squared loss; in a deep nonlinear network, because $\mathbb{E}[\mathrm{ReLU}(z)] \neq \mathrm{ReLU}(\mathbb{E}[z])$, the deterministic test-time network is an effective approximation to the ensemble of 2^h thinned subnetworks—see Exercises 3 and 5.)

4.6.2 Implementation

4.6.2.1 The Dropout Layer

Our from-scratch `dropout_layer(X, dropout)` draws a uniform tensor in $[0, 1)$, keeps entries $\geq$ `dropout`, and divides by `1.0 - dropout` (returning zeros when `dropout == 1`):

```
def dropout_layer(X, dropout):
    assert 0 <= dropout <= 1
    if dropout == 1: return torch.zeros_like(X)
    mask = torch.rand_like(X) >= dropout
    return mask.to(X.dtype) * X / (1.0 - dropout)
```

Let us test `dropout_layer` on a 2×8 tensor with dropout probabilities 0, 0.5, and 1. We also apply $p = 0.2$ to a vector of one million ones and verify that its empirical mean and variance match the theoretical values 1 and $p/(1-p) = 0.25$.

```
X = d2l.reshape(d2l.arange(16, dtype=d2l.float32), (2, 8))
for p in (0, 0.5, 1):
    print(f'p = {p}:', dropout_layer(X, p))
Y = dropout_layer(d2l.ones(10**6), 0.2)
print(f'mean {float(d2l.reduce_mean(Y)):.3f}, '
```

```
        f'variance {float(d2l.reduce_mean((Y - 1) ** 2)):.3f}')
```

```
p = 0: tensor([[ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11., 12., 13., 14., 15.]])
p = 0.5: tensor([[ 0.,  2.,  4.,  6.,  8., 10., 12., 14.],
        [ 0., 18.,  0., 22., 24., 26.,  0., 30.]])
p = 1: tensor([[0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0.]])
mean 1.000, variance 0.250
```

4.6.2.2 Defining and Training the Model from Scratch

We apply dropout to the outputs of each hidden layer right after its ReLU activation (Figure 4.15)—active *only* during training. Below we define our He-initialized two-hidden-layer classifier `DropoutMLPScratch` ($784 \to 256 \to 256 \to 10$) with separate dropout probabilities `dropout_1` and `dropout_2`:

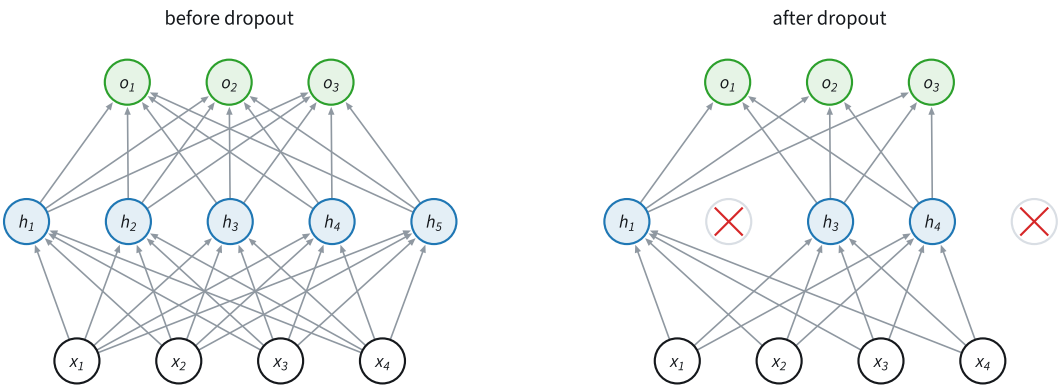


Figure 4.15: Dropout silences a random subset of hidden units (h_2 and h_5 shown) and their incident connections on each training step.

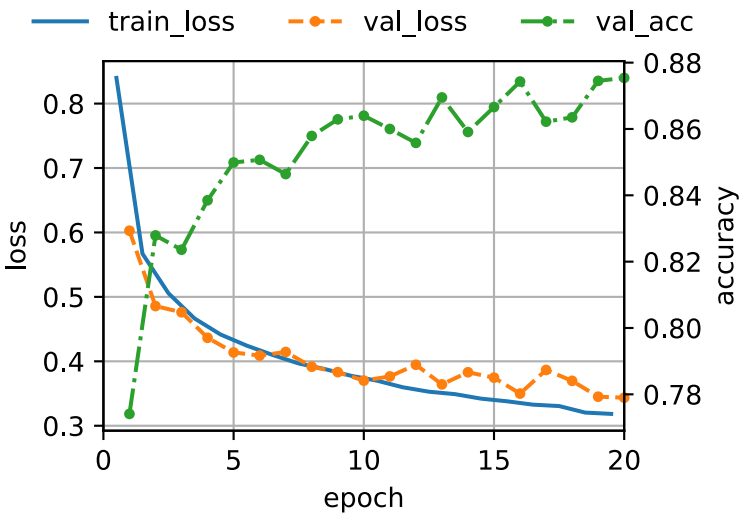
The layers test `self.training`, which the trainer's `model.eval()` sets to `False`.

```
class DropoutMLPScratch(d2l.Classifier):
    def __init__(self, num_outputs, num_hiddens_1, num_hiddens_2,
                 dropout_1, dropout_2, lr, num_inputs=784):
        super().__init__()
        self.save_hyperparameters()
        he = lambda n_in, n_out: nn.Parameter(  # Variance 2 / n_in
            d2l.normal(0, (2 / n_in) ** 0.5, (n_in, n_out)))
        self.W1 = he(num_inputs, num_hiddens_1)
        self.W2 = he(num_hiddens_1, num_hiddens_2)
        self.W3 = he(num_hiddens_2, num_outputs)
        self.b1 = nn.Parameter(d2l.zeros(num_hiddens_1))
        self.b2 = nn.Parameter(d2l.zeros(num_hiddens_2))
        self.b3 = nn.Parameter(d2l.zeros(num_outputs))

    def forward(self, X):
```

```
        X = X.reshape(-1, self.num_inputs)
        H1 = torch.relu(X @ self.W1 + self.b1)
        if self.training:
            H1 = dropout_layer(H1, self.dropout_1)
        H2 = torch.relu(H1 @ self.W2 + self.b2)
        if self.training:
            H2 = dropout_layer(H2, self.dropout_2)
        return H2 @ self.W3 + self.b3
```

We now instantiate `DropoutMLPScratch` with `dropout_1 = 0.2` after the first hidden layer, `dropout_2 = 0.5` after the second, and learning rate `lr = 0.1` (packing the hyperparameters into `hparams` for reuse below), and train it for 20 epochs on Fashion-MNIST:

```
hparams = {'num_outputs': 10, 'num_hiddens_1': 256, 'num_hiddens_2': 256,
           'dropout_1': 0.2, 'dropout_2': 0.5, 'lr': 0.1}
model = DropoutMLPScratch(**hparams)
data = d2l.FashionMNIST(batch_size=256)
trainer = d2l.Trainer(max_epochs=20, num_gpus=1)
trainer.fit(model, data)
```



Notice that during the opening epochs, training loss sits *above* validation loss because training loss is evaluated with noisy dropout masks active while validation loss runs the full deterministic network.

4.6.2.3 Concise Implementation

Using framework `Dropout` layers inside `Sequential`, we can express the exact same two-hidden-layer architecture in a few lines:

```
class DropoutMLP(d2l.Classifier):
    def __init__(self, num_outputs, num_hiddens_1, num_hiddens_2,
```

```
                dropout_1, dropout_2, lr):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(
            nn.Flatten(), nn.LazyLinear(num_hiddens_1), nn.ReLU(),
            nn.Dropout(dropout_1), nn.LazyLinear(num_hiddens_2), nn.ReLU(),
            nn.Dropout(dropout_2), nn.LazyLinear(num_outputs))

    def weight_init(self, module):
        if isinstance(module, nn.Linear):
            nn.init.kaiming_normal_(module.weight, nonlinearity='relu')
            nn.init.zeros_(module.bias)
```

4.6.3 Effects of Dropout under Label Noise

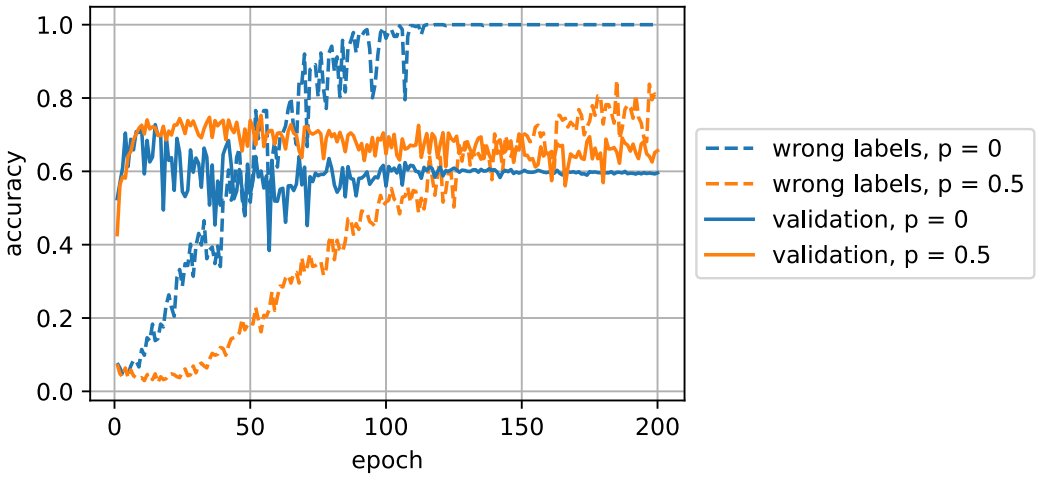
On the full $60{,}000$-image Fashion-MNIST dataset over 20 epochs, an unregularized MLP barely overfits, leaving little room for dropout to shine. To watch dropout combat severe memorization, we return to the $1{,}000$-image noisy-label benchmark from Section 4.5.4, corrupting 40% of the training labels at random via `d2l.corrupt` ($\sim 36\%$ wrong labels, indexed by the Boolean mask `wrong = labels != y`):

```
images = d2l.FashionMNIST(batch_size=2000)
X, y = (d2l.numpy(a) for a in next(iter(images.train_dataloader())))
X_val, y_val = (d2l.numpy(a) for a in next(iter(images.val_dataloader())))
X, y, X_val = X[:1000].reshape(1000, -1), y[:1000], X_val.reshape(2000, -1)
labels = d2l.corrupt(y, 0.4, np.random.default_rng())
wrong = labels != y  # The training labels that are now wrong
```

We now train `DropoutMLP` for 200 epochs on `(X, labels)` with dropout probability `p = 0` (no dropout) versus `p = 0.5` in both hidden layers, using `d2l.Tracker` to record accuracy on the corrupted training subset `X[wrong]` alongside held-out validation accuracy on `X_val`:

```
acc = {}
for p in (0, 0.5):
    trainer = d2l.Tracker(max_epochs=200, datasets={
        'wrong labels': d2l.ArrayData(X[wrong], labels[wrong], 1000),
        'validation': d2l.ArrayData(X_val, y_val, 1000)})
    trainer.fit(DropoutMLP(**{**hparams, 'dropout_1': p, 'dropout_2': p}),
                d2l.ArrayData(X, labels))
    acc[p] = trainer.accuracy
    print(f'dropout {p}: wrong labels reproduced '
          f'{acc[p]["wrong labels"][-1]:.2f}, validation accuracy over the '
          f'last 10 epochs {np.mean(acc[p]["validation"][-10:]):.3f}')
d2l.plot(list(range(1, 201)), [acc[0]['wrong labels'], acc[0.5]['wrong labels'],
                               acc[0]['validation'], acc[0.5]['validation']],
         'epoch', 'accuracy', fmts=('C0--', 'C1--', 'C0-', 'C1-'),
         legend=['wrong labels, p = 0', 'wrong labels, p = 0.5',
                 'validation, p = 0', 'validation, p = 0.5'],
         legend_loc='outside', figsize=(4.5, 3))
```

```
dropout 0: wrong labels reproduced 1.00, validation accuracy over the last 10 epochs 0.595
dropout 0.5: wrong labels reproduced 0.80, validation accuracy over the last 10 epochs 0.654
```



Look at the two trajectories:

- **Without dropout ($p = 0$):** By epoch 100, the MLP memorizes 100% of the corrupted training labels, and validation accuracy degrades by ~ 10 percentage points from its early peak.
- **With dropout ($p = 0.5$):** Memorization of wrong labels slows down. At epoch 200 the MLP reproduces roughly 70–85% of them instead of all, and validation accuracy over the final 10 epochs is **3 to 6 percentage points higher** than for the unregularized network. The wrong-label curve is still rising, though: dropout delays the damage rather than removing it, which is why it is combined with early stopping.

In practice, dropout is most valuable precisely when parameter count dwarfs dataset size and training runs for many epochs (as in early-2010s vision classifiers like AlexNet in Section 7.1); in single-pass foundation-model pretraining or directly before batch normalization layers (Section 7.3, where dropout's $1/(1-p)$ training variance inflation creates a train–test variance shift (Li et al. 2019)), dropout is typically reduced or omitted.

4.6.4 Summary

Inverted dropout zeroes out each hidden activation independently with probability p and rescales surviving activations by $1/(1-p)$ during training. Because this rescaling preserves $\mathbb{E}[h'_j] = h_j$ while injecting variance $\frac{p}{1-p}h_j^2$, the full unmasked network runs deterministically at test time without any weight rescaling. On small or noisy datasets where parameter count far exceeds sample size, dropout discourages fragile co-adaptation across hidden units, slows the memorization of corrupted labels, and lifts held-out accuracy by several percentage points. Because it delays rather than prevents eventual memorization, dropout works best alongside early stopping and weight decay.

4.6.5 Exercises

1. code **Dropout probabilities.** The model of this section drops units with probability 0.2 after the first hidden layer and 0.5 after the second.

   1. Train it with the two probabilities swapped, with both set to 0.5, and with both set to 0.8. For each run, report the best validation accuracy. Which layer tolerates the higher probability?
   2. The `Trainer` computes the training loss that it plots with the masks switched on. After training, evaluate the loss on the training images with the masks switched off, and compare the gap between training and validation loss for the networks trained with and without dropout.

2. **Dropout and He initialization.** The He initialization of Section 4.4.2.2 chooses the variance of the weights so that the mean square of the pre-activations stays the same from layer to layer.

   1. During training, by what factor does dropout with probability p change the mean square of the pre-activations of the next layer? Give the factor for each of the two dropout layers of this section's model.
   2. In a network with $K-1$ hidden layers, each followed by dropout with probability p, how does the factor grow with K? Should the initialization correct for it, given that the network runs without dropout at test time? Relate your answer to the mismatch between training and test that dropout causes before a batch normalization layer.

3. code **Monte Carlo dropout.** Instead of switching dropout off at test time, keep it on in the trained `DropoutMLPScratch` and run $T = 20$ forward passes per validation image, then average the softmax outputs (Gal and Ghahramani 2016). Compare the accuracy and the calibration (the mean of the largest softmax probability against the accuracy) with those of the single pass of the full network. How does the accuracy change as T grows from 1 to 20?

4. code **Dropout and weight decay.** Weight decay and dropout both regularize the model of this section, by different mechanisms. Measure each run by its best validation accuracy and by the gap between training and validation loss, with both losses evaluated with the masks switched off.

   1. With both dropout probabilities set to zero, add weight decay as in Section 2.7 at strengths $\lambda \in \{10^{-4}, 10^{-3}, 10^{-2}\}$. Compare these runs with the runs with and without dropout of this section.
   2. Keep the dropout probabilities at 0.2 and 0.5, and add weight decay at the same strengths. Are the effects of the two regularizers additive, do they show diminishing returns, or do they cancel each other out?

5. **Dropout at test time.** ● Set `dropout_1` to zero, so that only the second hidden layer is masked, and let ξ range over its masks. Let $\mathbf{o}(\xi)$ and $\hat{\mathbf{y}}(\xi)$ be the logits and the softmax output under mask ξ, which occurs with probability $P(\xi)$. Show that the logits of the full network equal $\sum_\xi P(\xi)\,\mathbf{o}(\xi)$. Show further that its softmax output equals the normalized weighted geometric mean

$$\tilde{y}_j = \frac{\prod_\xi \hat{y}_j(\xi)^{P(\xi)}}{\sum_k \prod_\xi \hat{y}_k(\xi)^{P(\xi)}}. \tag{4.20}$$

   Explain why neither equality holds in general once `dropout_1` is nonzero.

4.7 Predicting House Prices on Kaggle

Clean image benchmarks like Fashion-MNIST arrive with uniform pixel grids in $[0, 1]$ and zero missing values. Real-world tabular data rarely looks so tidy. In this capstone section of the chapter, we tackle the Kaggle House Prices: Advanced Regression Techniques competition compiled by De Cock (2011) from Ames, Iowa property sales (2006–2010): $1{,}460$ labeled training homes and $1{,}459$ unlabeled test homes, each described by 79 heterogeneous numerical and categorical attributes with scales spanning five orders of magnitude and missing entries across 19 columns.

Working through this end-to-end tabular pipeline brings together the tools of Chapter 1 through Chapter 4: `pandas` preprocessing without data leakage, logarithmic target transformation, K-fold cross-validation on shared folds, linear regression baselines, and He-initialized MLPs regularized with weight decay.

```
import pandas as pd
```

4.7.1 The Dataset and the Log-RMSE Metric

We download and cache the training and test CSV files via `d2l.download` and load them into `pandas` DataFrames (Section 1.2):

```
def read_house_prices(root='../data'):
    """Download and read the training and test tables of the competition."""
    train = pd.read_csv(d2l.download(
        d2l.DATA_URL + 'kaggle_house_pred_train.csv', root,
        sha1_hash='585e9cc93e70b39160e7921475f9bcd7d31219ce'))
    test = pd.read_csv(d2l.download(
        d2l.DATA_URL + 'kaggle_house_pred_test.csv', root,
        sha1_hash='fa19780a7b011d9b009e8bff8e99922a8ee2eb90'))
    return train, test
```

Calling `read_house_prices()` loads the two tables into `raw_train` and `raw_test`, whose shapes confirm $1{,}460$ training homes across 81 columns (including `SalePrice`) and $1{,}459$ test homes across 80 columns:

```
raw_train, raw_test = read_house_prices()
print(raw_train.shape, raw_test.shape)
```

```
(1460, 81) (1459, 80)
```

Inspecting the first four rows across a few leading and trailing columns of `raw_train` shows a representative mix of integer codes (`Id`, `MSSubClass`), string categories (`MSZoning`, `SaleType`, `SaleCondition`), continuous measurements (`LotFrontage`), and the dollar target `SalePrice`:

```
print(raw_train.iloc[:4, [0, 1, 2, 3, -3, -2, -1]])
```

```
   Id  MSSubClass MSZoning  LotFrontage SaleType SaleCondition  SalePrice
0   1          60       RL         65.0       WD        Normal     208500
1   2          20       RL         80.0       WD        Normal     181500
```

```
2   3          60       RL         68.0       WD         Normal     223500
3   4          70       RL         60.0       WD         Abnorml    140000
```

Of the 79 predictive attributes (after dropping the non-informative row index `Id`), 36 are stored as numbers and 43 as strings, with missing values often encoding the absence of a feature (such as `PoolQC = NaN` for homes without a swimming pool).

Because home prices range from $34{,}900 to $755{,}000, a $50{,}000 error is catastrophic on a starter home yet modest on a mansion. Kaggle therefore scores predictions by *relative* accuracy via the root-mean-squared error of the *logarithms* of the prices:

$$\text{log-RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\log y_i - \log \hat{y}_i)^2}. \tag{4.21}$$

For small $\delta = |\log \hat{y} - \log y|$, $\hat{y}/y = e^{\pm\delta} \approx 1 \pm \delta$, so a log-RMSE of 0.135 corresponds to roughly a 13.5% typical percentage error. By training our models to predict $\log y$ under squared error loss, we optimize the square of the competition metric directly.

4.7.2 Leakage-Free Preprocessing

Turning the raw DataFrames into float32 feature matrices requires four careful steps, implemented in `preprocess(train, *others)` below:

1. **Recast numeric category codes to strings:** `MSSubClass` uses numbers $(20, 60, 90, \ldots)$ to name dwelling styles (1-story, 2-story, duplex); converting it to `str` prevents the model from treating code 90 as a linear multiple of code 20.
2. **Standardize numerical features using training-set statistics only:** Computing means μ and standard deviations s exclusively on `train` and applying $(x - \mu)/s$ to both training and validation/test tables puts all numerical features on a unit scale for SGD and weight decay while preventing *data leakage* from validation or test rows.
3. **Impute missing numerical values with 0 (the training mean):** Filling standardized `NaN` entries with `0` replaces missing numerical measurements by their training-set mean (Exercise 1).
4. **One-hot encode categoricals (`pd.get_dummies(..., dummy_na=True)`):** Expands each categorical column into binary indicators (including an explicit indicator for `NaN`) and aligns validation/test columns to the exact training column vocabulary via `reindex(..., fill_value=0)`:

```
def preprocess(train, *others):
    """Fit the feature preprocessing on train; apply it to all tables."""
    frames = [df.drop(columns=['Id', 'SalePrice'], errors='ignore')
                .astype({'MSSubClass': str})  # A code, not a quantity
              for df in (train, *others)]
    numeric = frames[0].select_dtypes(include='number').columns
    mean, std = frames[0][numeric].mean(), frames[0][numeric].std()
    for df in frames:
        df[numeric] = ((df[numeric] - mean) / std).fillna(0)
    frames = [pd.get_dummies(df, dummy_na=True) for df in frames]
    return [df.reindex(columns=frames[0].columns, fill_value=0)
              .to_numpy(dtype=np.float32) for df in frames]
```

Running `preprocess(raw_train, raw_test)` on the full competition tables expands the 79 raw attributes into aligned `float32` design matrices `X_train` and `X_test` with 345 features per house:

```
X_train, X_test = preprocess(raw_train, raw_test)
X_train.shape, X_test.shape
```

```
((1460, 345), (1459, 345))
```

Finally, `KaggleHouse(d2l.DataModule)` pairs `preprocess` with logarithmic target centering, subtracting the training split's mean log price `self.offset` (≈ 12.02) from $\log(\text{SalePrice})$ so the network starts at the right baseline from step 1 (Exercise 2):

```
class KaggleHouse(d2l.DataModule):
    """Features and centered log prices, fitted on the training rows."""
    def __init__(self, raw_train, raw_val, batch_size=64):
        super().__init__()
        self.save_hyperparameters(ignore=['raw_train', 'raw_val'])
        self.X_train, self.X_val = preprocess(raw_train, raw_val)
        self.num_inputs = self.X_train.shape[1]
        log_price = lambda df: np.log(
            df['SalePrice'].to_numpy(np.float32)).reshape(-1, 1)
        self.offset = log_price(raw_train).mean()  # Mean log price
        self.y_train = log_price(raw_train) - self.offset
        self.y_val = (log_price(raw_val) - self.offset
                      if 'SalePrice' in raw_val else None)

    def get_dataloader(self, train):
        X, y = ((self.X_train, self.y_train) if train
                else (self.X_val, self.y_val))
        if y is None:  # No prices to validate against
            return None
        return self.get_tensorloader((d2l.tensor(X), d2l.tensor(y)), train)
```

4.7.3 Model Selection via K-Fold Cross-Validation

4.7.3.1 Shared 5-Fold Splits and the Linear Baseline

With only $1{,}460$ labeled houses, a single 20% validation split (292 houses) has a fold-to-fold standard error of several hundredths in log-RMSE. Our `k_fold_data(raw, k)` helper partitions `raw` into K random folds and builds a separate `KaggleHouse` data module for each round so validation folds never leak into the training feature statistics:

```
def k_fold_data(raw, k):
    """Split the table raw into k random folds; one data module per round."""
    folds = np.array_split(np.random.default_rng().permutation(len(raw)), k)
    return [KaggleHouse(raw.drop(index=raw.index[f]), raw.iloc[f])
            for f in folds]
```

Given a list of pre-built `folds` and a model factory `make_model`, `k_fold` trains a fresh model on each fold for `max_epochs` epochs, averages the last five epochs of squared loss before taking

the square root to damp single-epoch jitter, and prints the mean training and validation log-RMSEs across all folds:

```
def k_fold(folds, make_model, max_epochs=100):
    """Train a fresh model in each round; return the RMSEs of every fold."""
    rmse = []
    for fold in folds:
        trainer = d2l.Trainer(max_epochs=max_epochs)
        trainer.board.display = False
        trainer.fit(make_model(fold.num_inputs), fold)
        rmse.append([np.mean(trainer.history[f'{split}_loss'][-5:]) ** 0.5
                     for split in ('train', 'val')])  # Last five epochs
    train, val = np.array(rmse).T
    print(f'training RMSE {train.mean():.3f}, validation RMSE '
          f'{val.mean():.3f} (folds {val.min():.3f} to {val.max():.3f})')
    return train, val
```

We now generate our shared 5-fold partition `folds` once so every candidate model below is evaluated on the exact same five training and validation splits:

```
folds = k_fold_data(raw_train, 5)
```

First, let us define `make_linear` for our `d2l.LinearRegression` baseline (100 epochs, batch size 64, learning rate $\eta = 0.02$):

```
make_linear = lambda num_inputs: d2l.LinearRegression(lr=0.02)
```

Running `k_fold(folds, make_linear)` trains the unregularized linear baseline across all five folds and records its training and validation log-RMSE vectors in `linear`:

```
linear = k_fold(folds, make_linear)
```

```
training RMSE 0.114, validation RMSE 0.145 (folds 0.129 to 0.182)
```

The linear baseline achieves a 5-fold validation log-RMSE of ~ 0.15 ($\sim 15\%$ typical price error) vs. ~ 0.11 training log-RMSE.

4.7.3.2 An MLP with Weight Decay

Now let us build `HouseMLP`: a one-hidden-layer MLP with 64 ReLU units ($\sim 22{,}000$ parameters, or ~ 19 parameters per training house in a fold), He-initialized weights, and ℓ_2 weight decay `wd` applied exclusively to the weight matrices (excluding biases):

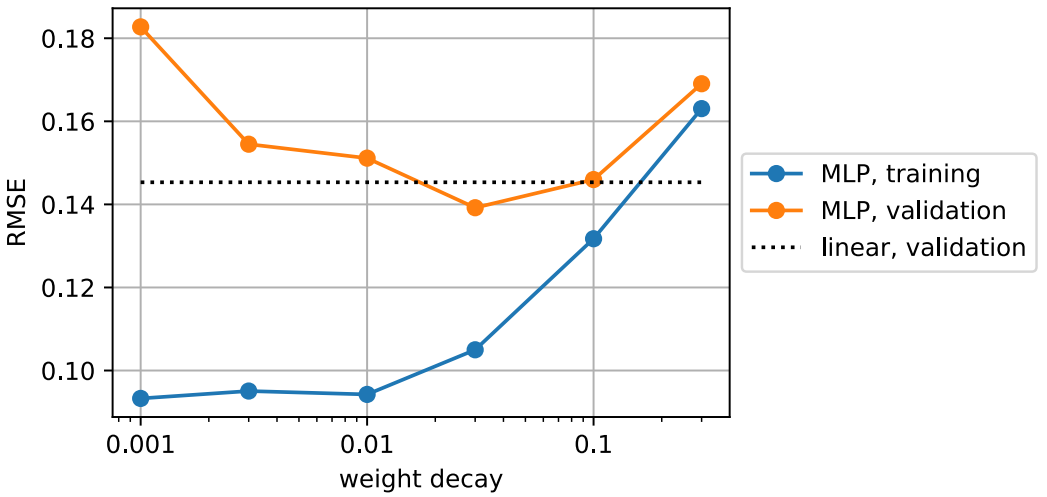
```
class HouseMLP(d2l.Module):
    """An MLP with one hidden layer, squared loss, and weight decay."""
    loss = d2l.LinearRegression.loss

    def __init__(self, num_hiddens, wd, lr):
        super().__init__()
        self.save_hyperparameters()
```

```
        self.net = nn.Sequential(nn.LazyLinear(num_hiddens), nn.ReLU(),
                                 nn.LazyLinear(1))

    def weight_init(self, module):
        if isinstance(module, nn.Linear):
            nn.init.kaiming_normal_(module.weight, nonlinearity='relu')
            nn.init.zeros_(module.bias)

    def configure_optimizers(self):
        weights = [p for p in self.parameters() if p.dim() > 1]
        biases = [p for p in self.parameters() if p.dim() == 1]
        return torch.optim.SGD([{'params': weights, 'weight_decay': self.wd},
                                {'params': biases}], lr=self.lr)

make_mlp = lambda num_inputs, wd: HouseMLP(num_hiddens=64, wd=wd, lr=0.02)
```

We now sweep `wd` across `wds = [0.001, 0.003, 0.01, 0.03, 0.1, 0.3]` on our shared 5-fold splits `folds` and plot the mean training and validation RMSEs against the linear baseline:

```
wds = [0.001, 0.003, 0.01, 0.03, 0.1, 0.3]
mlp = {}
for wd in wds:
    print(f'weight decay {wd}: ', end='')
    mlp[wd] = k_fold(folds, lambda num_inputs: make_mlp(num_inputs, wd))
d2l.plot(wds, [[mlp[wd][0].mean() for wd in wds],
               [mlp[wd][1].mean() for wd in wds], [linear[1].mean()] * 6],
         'weight decay', 'RMSE', xscale='log', fmts=('C0-o', 'C1-o', 'k:'),
         legend=['MLP, training', 'MLP, validation', 'linear, validation'],
         legend_loc='outside', figsize=(4.5, 3))
```

```
weight decay 0.001: training RMSE 0.093, validation RMSE 0.183 (folds 0.146 to 0.277)
weight decay 0.003: training RMSE 0.095, validation RMSE 0.154 (folds 0.137 to 0.174)
weight decay 0.01: training RMSE 0.094, validation RMSE 0.151 (folds 0.138 to 0.168)
weight decay 0.03: training RMSE 0.105, validation RMSE 0.139 (folds 0.130 to 0.153)
weight decay 0.1: training RMSE 0.132, validation RMSE 0.146 (folds 0.139 to 0.159)
weight decay 0.3: training RMSE 0.163, validation RMSE 0.169 (folds 0.153 to 0.180)
```



Selecting the weight decay `best` with the lowest 5-fold mean validation RMSE, we can also inspect the paired per-fold difference `mlp[best][1] - linear[1]` where fold-sampling noise cancels out:

```
best = min(wds, key=lambda wd: mlp[wd][1].mean())
print(f'weight decay {best}: MLP minus linear, per fold',
      np.round(mlp[best][1] - linear[1], 3))
```

```
weight decay 0.03: MLP minus linear, per fold [-0.009  0.    -0.029  0.012 -0.006]
```

The cross-validation curve traces a textbook U-shape: weak weight decay (`wd = 0.001`) overfits the training folds (training RMSE $\sim$ 0.09, validation RMSE $\sim$ 0.15–0.18, worse than linear regression), intermediate weight decay (`wd = 0.03` or `0.1`) hits the sweet spot at validation RMSE $\sim$ 0.13–0.14 (beating the linear baseline on most folds), and excessive decay (`wd = 0.3`) underfits.

4.7.4 Submitting Predictions

With the winning weight decay `best` selected by 5-fold cross-validation, we retrain `HouseMLP` on all $1{,}460$ labeled houses (or ensemble the 5 fold models; see Exercise 5), add back the training mean log-price `data.offset`, exponentiate to dollar predictions, and save `submission.csv`:

```
data = KaggleHouse(raw_train, raw_test)
model = make_mlp(data.num_inputs, wd=best)
trainer = d2l.Trainer(max_epochs=100)
trainer.board.display = False
trainer.fit(model, data)
log_price = d2l.numpy(model(d2l.tensor(data.X_val))).reshape(-1) + data.offset
submission = pd.DataFrame({'Id': raw_test['Id'],
                           'SalePrice': np.exp(log_price)})
submission.to_csv('submission.csv', index=False)
submission.head()
```

```
     Id      SalePrice
0  1461  119006.625000
1  1462  154548.484375
2  1463  177151.687500
3  1464  201103.609375
4  1465  186113.750000
```

4.7.5 Summary

Real-world tabular regression hinges on disciplined, leakage-free preprocessing: dropping arbitrary row identifiers, recasting numeric category codes to strings, standardizing numerical columns with training-split means and standard deviations before zero-imputing missing entries, one-hot encoding categoricals with an explicit missingness indicator, and centering log-transformed targets around the training mean. Evaluating all candidate models on the same $K = 5$ cross-validation folds—each refitting its own preprocessor on its four training folds—cancels fold-sampling noise when comparing models head to head. On Ames Housing, tuning ℓ_2 weight decay moves a 22,000-parameter ReLU MLP from overfitting (~ 0.15–0.18 log-RMSE) past the linear baseline (~ 0.15) to ~ 0.13–0.14 log-RMSE, though gradient-boosted decision trees remain a strong competitor on heterogeneous tabular features (Exercise 3).

4.7.6 Exercises

1. code **Mean imputation.** `preprocess` replaces every missing numerical value by the training-set mean of its column. Its call to `pd.get_dummies` with `dummy_na=True` also adds a missing-value indicator column for every categorical feature.

    1. Identify a numerical feature in this dataset whose missingness is not random with respect to the sale price.
    2. Explain what bias mean imputation alone would introduce for that feature, and in which direction it would shift the predictions for the affected houses. Then determine whether the indicator columns that `preprocess` already creates let a linear model remove this bias.
    3. Replace mean imputation by median imputation, and separately add a binary missingness indicator column for each imputed feature. Compare the cross-validated errors of the three variants.

    Adapted from Kaggle Learn, Intermediate Machine Learning, Exercise 2.

2. code **Scaling the features and the target.** Remove the standardization from `preprocess`. Because `fillna(0)` imputes the column mean only for standardized columns, fill missing numerical values with the training-set column means explicitly instead.

    1. Rerun the cross-validation of the linear model at the section's learning rate of 0.02, and describe the training loss.
    2. Find, to within a factor of ten, the largest learning rate at which the unstandardized linear model trains without diverging. Compare its cross-validated error after 100 epochs with that of the standardized pipeline, and explain both observations.
    3. Restore the standardization, but train on the log prices without subtracting their mean. Compare the cross-validated error after 100 epochs with that of the centered targets, and explain the difference.

4. With the section's pipeline, record the validation RMSE of the linear model and the mean of its residuals on the validation fold after every epoch. In the epochs where the RMSE jumps, what has moved? Every house has exactly 44 one-hot features equal to 1 (45 inputs equal to 1 if we count the bias). Explain why a step of SGD changes the mean prediction more than it changes any single weight, and check whether halving the learning rate removes the jumps.

3. code **Gradient-boosted trees.** On tabular data like this, ensembles of decision trees often predict better than neural networks (Grinsztajn et al. 2022; Shwartz-Ziv and Armon 2022). Fit a gradient-boosted tree model (for example scikit-learn's `GradientBoostingRegressor`, or XGBoost or LightGBM if installed) to the centered log prices of the training rows of each fold in `folds`, and compute its validation RMSE on each fold. Compare the average with those of this section's models. Which properties of the data favor trees?

4. code **Target leakage.** A feature that is recorded only after, or because of, the event being predicted inflates validation scores without helping a deployed model.

    1. Identify at least one raw feature in this dataset that could plausibly be filled in only after the sale, and explain why cross-validation does not catch the problem.
    2. Verify the mechanism empirically: add a synthetic feature equal to the log sale price plus small Gaussian noise, rerun the cross-validation, and report the change in the validation RMSE.

    Adapted from Kaggle Learn, Intermediate Machine Learning, Exercise 7.

5. code **Fold ensembling versus retraining.** Section 4.7.4 retrains the selected MLP on all houses. Alternatively, the five models trained during cross-validation could each predict the test houses, with the preprocessing of their own round, and their log-price predictions could be averaged.

    1. Implement this fold ensemble and submit both its predictions and those of the retrained model. Compare the two leaderboard scores with each other and with the cross-validated RMSE.
    2. Each model of the ensemble was trained on four fifths of the houses. Why might the ensemble nevertheless predict better than the model retrained on all of them?
    3. Explain why the training error of the retrained model is not a substitute for the cross-validated estimate.

6. code **Target encoding.** ● *Target encoding* replaces each category of a categorical column by the mean log sale price of the training rows in that category, so that the feature occupies a single column instead of the one column per category that `preprocess` creates.

    1. Target-encode every categorical column with more than ten categories instead of one-hot encoding it, computing the category means from the training rows of each fold. Give a category that does not occur in those rows the overall mean, and standardize the encoded columns like the numerical features. Compare the cross-validated error with that of one-hot encoding.
    2. The encoding above computes each training row's value from a mean that includes that row's own label. Encode the training rows out of fold instead: split the training rows of each fold into five parts, and encode each part with category means computed from the other four parts. Compare the cross-validated error

with that of the in-sample encoding, and explain how the size of a category affects the difference between the two encodings.

7. **Geometric averaging.** ● An ensemble of K models that predict the log prices $z_1, \dots, z_K$ of a house can average them before exponentiating, which amounts to a geometric mean in price space.

   1. Compare $\exp\big(\frac{1}{K}\sum_{k=1}^{K} z_k\big)$ with the price-space average $\frac{1}{K}\sum_{k=1}^{K} e^{z_k}$. Show which of the two never exceeds the other, and state when they coincide.

   2. Let $\log y$ be the true log price of the house. Show that the log-space average satisfies

$$\Big(\log y - \frac{1}{K}\sum_{k=1}^{K} z_k\Big)^2 \le \frac{1}{K}\sum_{k=1}^{K} \big(\log y - z_k\big)^2 . \tag{4.22}$$

   Determine whether the logarithm of the price-space average satisfies the same bound for all predictions $z_1, \dots, z_K$.

Chapter 5

Computation

Up to now we have relied on frameworks to manage model construction, parameters, initialization, and device execution behind the scenes. Moving from textbook prototypes to production-scale systems requires opening that black box. Training an 8-billion-parameter language model in 32-bit floating point with Adam demands 128 GB of parameter, gradient, and optimizer state before storing a single activation, and runs across thousands of GPUs where hardware failures interrupt training every few hours. At that scale, understanding how frameworks represent, initialize, compute with, and persist models is essential.

Everything in this chapter builds on one central data structure: a neural network is a *tree of modules* whose leaves hold its *state* (trainable parameters and fixed buffers alike), each addressed by its path from the root. The first half of the chapter develops that structural abstraction and shows how framework operations traverse it. We begin with module hierarchies, container registration, residual skip connections, lazy shape inference, and config-driven builders (Section 5.1), then open the state tree to separate trainable parameters from non-trainable buffers, derive the 16-byte-per-parameter memory floor of fp32 Adam training, count parameters without allocating storage, and share or freeze weights (Section 5.2). Next, we move from per-layer variance rules to whole-tree and residual initialization, showing why unscaled residual branches amplify activation variance as $(1 + g^2)^N$ and how $1/\sqrt{N}$ or zero output-projection scaling stabilizes deep stacks (Section 5.3), before extending the framework with stateless layers, parameterized layers (`RMSNorm`), fixed-buffer layers (`RandomFeatures`), and custom backward rules via the straight-through estimator (Section 5.4).

The second half turns from model structure to numerical representation, persistence, hardware execution, and diagnostics. We compare fp32, fp16, and bf16, explaining why 16-bit weights round away small updates and how mixed-precision training pairs fp32 master weights with 16-bit compute and fp16 loss scaling (Section 5.5). We then contrast unsafe `pickle` archives with code-free `safetensors` files, construct atomic five-part training checkpoints that resume bit for bit, and audit missing or unexpected keys when loading pretrained backbones (Section 5.6). Finally, we place tensors and models on GPUs, time asynchronous command queues accurately, dissect GPU memory alongside $O(\sqrt{K})$ activation checkpointing (Section 5.7), and combine pseudorandom seed control and GPU determinism with non-invasive forward and backward hooks (Section 5.8).

Resources and Further Reading

The resources below accompany the sections of this chapter; each names the section it goes with. All are freely accessible online except where noted.

Books

- Deep Learning with PyTorch (Stevens, Antiga & Viehmann): free PDF from the PyTorch team (archived copy); Part 1 covers tensors, storage, autograd, and `nn.Module` at the same level as Section 5.1 and Section 5.2.
- Machine Learning Systems (Vijay Janapa Reddi): free online; the systems view around this chapter, covering frameworks, data pipelines, training infrastructure, and efficient deployment (Section 5.7, Section 5.8).

Courses and video lectures

- CMU 10-414/714: Deep Learning Systems (Chen & Kolter): free lectures and assignments in which students build "needle", a miniature framework with autograd, modules, initialization, and GPU support (Section 5.1, Section 5.3, Section 5.7).
- Neural Networks: Zero to Hero (Andrej Karpathy): free video series building autograd, modules, and training loops from scratch in plain Python (Section 5.1, Section 5.4).
- fast.ai Part 2: Deep Learning Foundations (Howard et al.): free course rebuilding a training framework from tensor operations upward, including modules, initialization, and mixed precision (Section 5.1, Section 5.3, Section 5.5).

Tutorials, notes, and interactive

- PyTorch internals (Edward Yang): a guided tour of tensors, strides, dispatch, and autograd inside PyTorch, one level below Section 5.1.
- JAX: The Sharp Bits: pure functions, explicit PRNG keys, and JIT constraints, distilling the functional design behind JAX as a list of practical gotchas (Section 5.2, Section 5.8).
- Train With Mixed Precision (NVIDIA): the vendor guide to fp16/bf16 arithmetic and loss scaling that Section 5.5 distills.
- What Every Computer Scientist Should Know About Floating-Point Arithmetic (David Goldberg): the standard reference beneath every rounding and overflow issue in Section 5.5.
- Making Deep Learning Go Brrrr From First Principles (Horace He): a first-principles account of compute, memory bandwidth, and framework overhead (Section 5.7).
- safetensors (Hugging Face): the file format for sharing weights without executing code on load, and why it replaced pickle-based checkpoints (Section 5.6).
- Reproducibility (PyTorch notes): determinism flags, their performance costs, and their limits, complementing Section 5.8.

Foundational papers

- Automatic Differentiation in Machine Learning: a Survey (Baydin, Pearlmutter, Radul & Siskind, 2018, *JMLR*): the definitive survey of forward- and reverse-mode autodiff, the algorithm behind every custom gradient in Section 5.4.
- Mixed Precision Training (Micikevicius et al., 2018): the origin of the fp16 + master-weights + loss-scaling recipe in Section 5.5.
- PyTorch: An Imperative Style, High-Performance Deep Learning Library (Paszke et al., 2019): the design rationale (eager execution, autograd, caching allocator) for the imperative style of Section 5.1 and Section 5.7.

5.1 Modules and Model Construction

The multilayer perceptron in Section 4.2 had just four parameter arrays, small enough to manage by hand. Real-world architectures are far more intricate. A modern residual network for computer vision stacks dozens or hundreds of layers, and a language model such as Llama 3 8B (Grattafiori et al. 2024), holding 8 billion parameters across 32 identical transformer blocks, spreads its weights over 291 separate arrays. Every single one of those arrays must reach the optimizer, be written into checkpoints, and move onto the GPU. Worse, omitting an array raises no syntax error: it simply sits silently untrained or unsaved.

Deep learning frameworks solve this bookkeeping problem with a single recursive abstraction: the *module*. A module can represent an individual layer, a group of layers, or an entire model. It owns parameters, owns child modules, and defines a forward computation mapping inputs to outputs. Because modules nest inside other modules, any architecture forms a tree of modules (Figure 5.1). Every framework discovers all parameters by traversing this tree, provided each child is properly *registered* with its parent.

This section is hands-on, and readers who have already built custom modules can skim the basics. Do not skip how registration works and how it fails silently, however, because that bug is notoriously easy to hit. We also show how to write custom forward computations (such as residual connections), how lazy initialization infers shapes on the fly, and how to separate architectural hyperparameters into a compact configuration object. Later sections reuse the same tree traversal for counting parameters (Section 5.2), initializing weights (Section 5.3), saving checkpoints (Section 5.6), and placing models on GPUs (Section 5.7).

```
from dataclasses import dataclass
```

5.1.1 The Module Abstraction

5.1.1.1 Sequential Containers and the Registry

Every model we have written since Section 2.4 has been a module: `d2l.Module` subclasses each framework's base module class. The fastest way to assemble a pipeline of layers is a *sequential container*, which chains its children together one after another. Below we construct an MLP with one hidden layer of 256 ReLU units and 10 outputs, matching Section 4.2, and pass in a minibatch of two examples with 20 features each.

In PyTorch the base class is `nn.Module` and `nn.Sequential` chains layers passed as arguments.

```
net = nn.Sequential(nn.Linear(20, 256), nn.ReLU(), nn.Linear(256, 10))

X = torch.rand(2, 20)
net(X).shape
```

```
torch.Size([2, 10])
```

Under the hood, the container stores its children in an ordered *registry* mapping names to child modules, and its forward pass iterates through that table in insertion order.

`named_children` yields `(name, module)` pairs (`'0'`, `'1'`, `'2'`).

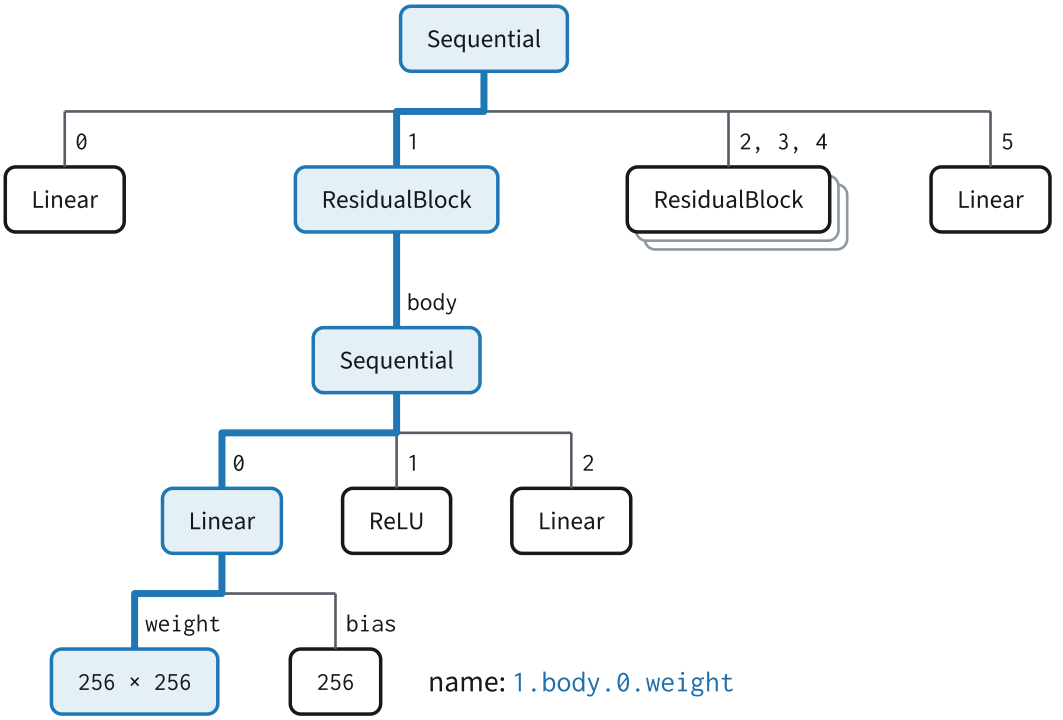


Figure 5.1: The module tree of the residual MLP that we build at the end of this section. Each edge carries the name under which the parent registered the child, and the path of names from the root to a parameter is the name of that parameter, here `1.body.0.weight`. Blocks 2 to 4 repeat block 1.

```
dict(net.named_children())
```

```
{'0': Linear(in_features=20, out_features=256, bias=True),
 '1': ReLU(),
 '2': Linear(in_features=256, out_features=10, bias=True)}
```

Every tree traversal over the model begins at this registry. Gathering parameters for the optimizer, serializing weights to disk, and moving tensors to a GPU all recurse through registered children down to every leaf parameter. As far as those operations are concerned, an unregistered layer might as well not exist.

5.1.1.2 Subclasses of the Module Class

A sequential container handles linear chains neatly, but many architectures branch, merge, or reuse components. For anything beyond a straight chain, we subclass the framework's base module and implement two methods: a constructor that creates child modules and a forward method that specifies how data flows through them.

In PyTorch the forward method is `forward`.

Here is the same MLP once more:

```
class MLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.hidden = nn.Linear(20, 256)
        self.out = nn.Linear(256, 10)

    def forward(self, X):
        return self.out(F.relu(self.hidden(X)))
```

Instantiating `MLP` and invoking `net(X)` dispatches to our custom forward method and maps the 2×20 input batch `X` to an output tensor of shape `(2, 10)`:

```
net = MLP()
net(X).shape
```

```
torch.Size([2, 10])
```

Two mechanisms do the heavy lifting behind the scenes. First, assigning `self.hidden = ...` is not plain Python attribute storage: the base module intercepts assignment via `__setattr__`, notices that the value is a module, and registers it under the attribute name (`hidden` and `out`). That automatic registration is all it takes for both layers' parameters to reach the optimizer. Second, we never have to write a backward method: automatic differentiation (Section 1.5) derives the gradients directly from the operations performed during the forward pass.

Always invoke a module as `net(X)`.

Calling `forward` directly skips the hooks that `__call__` runs (Section 5.8).

5.1.2 Module Containers

5.1.2.1 Writing a Container

A sequential container has just two responsibilities: register each child module as it is added, and pass the input through them in order. Implementing a minimal version ourselves clarifies exactly how registration works—and where it can go wrong.

```
class MySequential(nn.Module):
    def __init__(self, *args):
        super().__init__()
        for idx, module in enumerate(args):
            self.add_module(str(idx), module)

    def forward(self, X):
        for module in self.children():
            X = module(X)
        return X
```

`add_module` registers each child by string index, and `children()` iterates over the registry in insertion order.

Our custom `MySequential` behaves just like the built-in container:

```
net = MySequential(nn.Linear(20, 256), nn.ReLU(), nn.Linear(256, 10))
net(X).shape
```

```
torch.Size([2, 10])
```

5.1.2.2 Layers in a Plain List

What happens if we store layers in an ordinary Python `list`, as anyone unfamiliar with module registration naturally would? The forward pass can still loop over the list, but a plain `list` is not a module. Does the framework still discover the layers inside it?

```
class PlainListMLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.layers = [nn.Linear(20, 256), nn.ReLU(), nn.Linear(256, 10)]

    def forward(self, X):
        for layer in self.layers:
            X = layer(X)
        return X

net = PlainListMLP()
net(X).shape, sum(p.numel() for p in net.parameters())
```

```
(torch.Size([2, 10]), 0)
```

The forward pass succeeds, yet `net.parameters()` is empty because `__setattr__` treats a plain Python `list` as an ordinary attribute rather than a module container. Worse than a completely empty model (which `torch.optim.SGD` rejects) is a *partly* registered model whose outer

layers are attributes and inner layers sit in a plain list. We take one SGD step and check which weights move:

```
class PartlyRegisteredMLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.first = nn.Linear(20, 256)
        self.blocks = [nn.Linear(256, 256), nn.Linear(256, 256)]  # Plain list
        self.last = nn.Linear(256, 10)

    def forward(self, X):
        X = F.relu(self.first(X))
        for block in self.blocks:
            X = F.relu(block(X))
        return self.last(X)
```

To see which layers actually move during training, we snapshot all four weight matrices of `PartlyRegisteredMLP` in `before`, run a single backward pass on `net(X).sum()` followed by one SGD step, and compare each layer's weights against `before`:

```
net = PartlyRegisteredMLP()
layers = [net.first, *net.blocks, net.last]
before = [layer.weight.clone() for layer in layers]
optimizer = torch.optim.SGD(net.parameters(), lr=0.1)
net(X).sum().backward()
optimizer.step()
[not torch.equal(b, layer.weight) for b, layer in zip(before, layers)]
```

```
[True, False, False, True]
```

Only the outer weights updated, without a single warning or error. Backpropagation flowed through `self.blocks` to reach `self.first`, but the optimizer never saw the middle layers. Wrapping the list in `nn.ModuleList` (or `nn.ModuleDict` for dictionaries) registers every child properly:

```
class ModuleListMLP(PlainListMLP):
    def __init__(self):
        super().__init__()
        self.layers = nn.ModuleList(self.layers)

net = ModuleListMLP()
net(X).shape, sum(p.numel() for p in net.parameters())
```

```
(torch.Size([2, 10]), 7946)
```

All 7,946 parameters are now registered. Use `nn.Sequential` for straight chains, and `nn.ModuleList` or `nn.ModuleDict` whenever your `forward` method loops or indexes over children manually.

5.1.3 Forward Computation

Because a module's forward method is just Python, it can invoke arbitrary tensor operations, loops, conditionals, and skip paths. We use this flexibility to implement a residual connection, and then see what constraints arise when a framework compiles the training step.

5.1.3.1 Residual Connections

The simplest widely used architecture that is not a straight chain is a *residual connection*, which adds a block's input directly to its output (He et al. 2016b). Writing f for the block's *body* (here two fully connected layers with a ReLU between them), the block computes

$$\mathbf{y} = \mathbf{x} + f(\mathbf{x}). \tag{5.1}$$

Whenever $f(\mathbf{x}) = \mathbf{0}$, the block reduces to the identity map, allowing signals to pass cleanly through deep stacks while each block learns only a refinement $f(\mathbf{x})$ rather than rebuilding the representation from scratch. The ResNet architectures of Section 7.4 rely on this pattern. No framework needs a special built-in layer for residual connections: in a custom module, it is simply one line, `X + self.body(X)`:

```
class ResidualBlock(nn.Module):
    def __init__(self, num_hiddens):
        super().__init__()
        self.body = nn.Sequential(
            nn.Linear(num_hiddens, num_hiddens), nn.ReLU(),
            nn.Linear(num_hiddens, num_hiddens))

    def forward(self, X):
        return X + self.body(X)
```

Figure 5.2 depicts the wiring. Adding $\mathbf{x}$ and $f(\mathbf{x})$ requires their shapes to match, so a residual block takes a single hidden width `num_hiddens`.

```
block = ResidualBlock(24)
block(torch.randn(2, 24)).shape
```

```
torch.Size([2, 24])
```

A forward method can also depend on values that are neither inputs nor trainable parameters. A scalar constant can live as an ordinary Python attribute, whereas a tensor tracked from data (such as running input statistics) must be saved in checkpoints and moved across devices alongside the model without being updated by the optimizer. We cover how to register such non-trainable *buffers* in Section 5.2.1.2.

5.1.3.2 Eager and Compiled Execution

So far our forward passes have executed like regular Python, running one operation at a time. To eliminate Python overhead and fuse operations, however, frameworks can trace a computation once and replay the compiled graph on future calls. That speedup comes

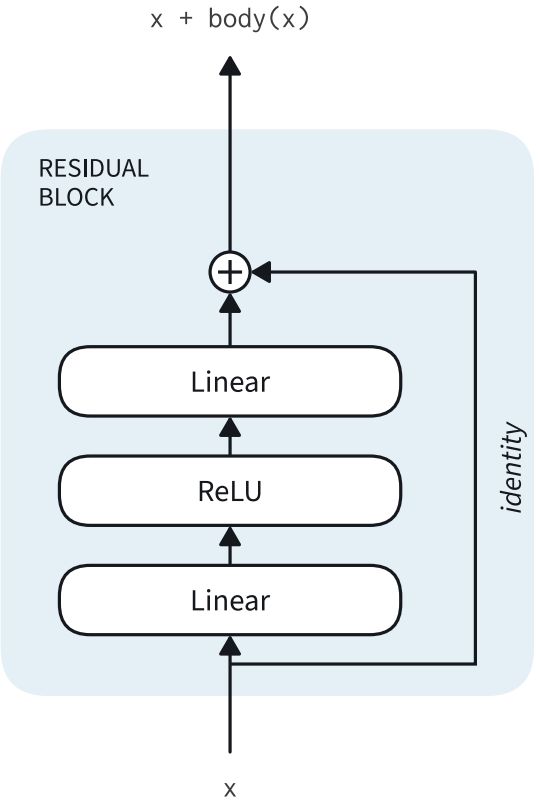


Figure 5.2: A residual block. The input splits into the body, Linear, ReLU, Linear, and an identity path, and the two rejoin by addition.

with a trade-off: compiled execution restricts data-dependent Python control flow inside the forward pass.

Both Gluon `Block` and PyTorch `nn.Module` run *eagerly* by default, executing Python statement by statement, so data-dependent control flow works just like NumPy (we cover graph compilation in Chapter 13).

5.1.4 Lazy Initialization

5.1.4.1 Shapes from Data

Many layer libraries let a layer specify only its output dimension at construction and infer its input width from the first minibatch.

Before the first forward pass, `nn.LazyLinear(256)` holds an `UninitializedParameter` without a shape:

```
net = nn.Sequential(nn.LazyLinear(256), nn.ReLU(), nn.LazyLinear(10))
net[0].weight
```

```
<UninitializedParameter>
```

Passing the first minibatch `X` through the network cascades shapes from layer to layer and materializes every weight tensor:

```
net(X)
net[0].weight.shape
```

```
torch.Size([256, 20])
```

5.1.4.2 Initialization after a Dry Run

Lazy layers free us from manually propagating input dimensions from layer to layer, which quickly becomes error-prone in deep models. The trade-off is that parameters do not exist until data flows through the network: before that first pass, we cannot inspect parameter counts or run shape-dependent initializers such as Xavier initialization (Section 4.4.2.1). A lazy model therefore needs its initializer either recorded up front or applied after a *dry run* (a single forward pass on a dummy minibatch whose output is discarded) that materializes every shape.

`d2l.Module.apply_init` runs one forward pass without tracking gradients to materialize lazy shapes, then applies an initializer callback across the module tree via `self.apply`:

```
@d2l.add_to_class(d2l.Module)
def apply_init(self, inputs, init=None):
    """Run a dry pass to fix lazy shapes, then apply `init`."""
    with torch.no_grad():
        self(*inputs)
    if init is not None:
        self.apply(init)
```

`nn.Module.apply(fn)` calls `fn` on every module of the tree, children before their parents: one more task that is a walk over the tree. Here is `apply_init` with Xavier's uniform initializer:

```
class TinyMLP(d2l.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(nn.LazyLinear(256), nn.ReLU(),
                                 nn.LazyLinear(10))

def init_xavier(module):
    if isinstance(module, nn.Linear):
        nn.init.xavier_uniform_(module.weight)

model = TinyMLP()
model.apply_init([X], init_xavier)
model.net[0].weight.shape
```

```
torch.Size([256, 20])
```

When should we use lazy layers versus explicit dimensions? For rapid prototyping on unfamiliar datasets, lazy layers are handy. For production models whose parameters we want to count, initialize programmatically, or instantiate from a configuration file without dummy data, declaring explicit input and output widths is cleaner, and that is the convention we adopt for the rest of this chapter.

5.1.5 Models from a Config

5.1.5.1 Configs and Builders

Can we reconstruct a trained model from its saved weight tensors alone? Not entirely. A trained model consists of both numerical arrays and the code that interprets them. Every width and depth we wrote earlier was hardcoded directly into constructor calls; a weights file reflects those choices only indirectly (tensor shapes reveal layer widths, but nothing in the arrays reveals a parameter-free choice such as ReLU vs. tanh).

Architectural hyperparameters that vary across experiments therefore belong in a dedicated *configuration* object (*config*) that can be logged alongside metrics and saved next to checkpoints, while the topology itself (what each block computes and how blocks wire together) lives in code. Below, a Python `dataclass` captures four architectural fields, and a `build` factory function turns any config into a complete module tree:

```
@dataclass
class MLPConfig:
    num_inputs: int = 784
    num_hiddens: int = 256
    num_blocks: int = 4
    num_outputs: int = 10

def build(cfg: MLPConfig) -> nn.Module:
    blocks = [ResidualBlock(cfg.num_hiddens) for _ in range(cfg.num_blocks)]
    return nn.Sequential(nn.Linear(cfg.num_inputs, cfg.num_hiddens),
                         *blocks, nn.Linear(cfg.num_hiddens, cfg.num_outputs))
```

A single config completely specifies an instance of the architecture: an input projection from `num_inputs` to `num_hiddens`, `num_blocks` residual blocks, and an output head. Notice that `blocks` starts as a temporary Python list, which is completely safe here because unpacking it into the sequential container registers every block immediately. Because every layer specifies both its input and output widths, we can inspect and count parameters right away without a dry run.

```
net = build(MLPConfig())
[(name, tuple(p.shape)) for name, p in net.named_parameters()][:4]
```

```
[('0.weight', (256, 784)),
 ('0.bias', (256,)),
 ('1.body.0.weight', (256, 256)),
 ('1.body.0.bias', (256,))]
```

The resulting model maps a minibatch of 784-dimensional vectors (such as flattened Fashion-MNIST images) to 10 output logits:

```
net(torch.rand(2, 784)).shape
```

```
torch.Size([2, 10])
```

5.1.5.2 Counting Parameters

A config also lets us calculate a model's exact parameter count on paper before allocating any memory. The input projection has $784 \cdot 256 + 256 = 200{,}960$ parameters, each residual block contains two dense layers totaling $2 \cdot (256 \cdot 256 + 256) = 131{,}584$ parameters, and the output layer adds $256 \cdot 10 + 10 = 2{,}570$, giving

$$200{,}960 + 4 \cdot 131{,}584 + 2{,}570 = 729{,}866 \tag{5.2}$$

parameters in all—dominated by the residual blocks. Scaling the model up now takes changing two config fields rather than editing model code, and walking the parameter tree verifies our calculation:

```
for cfg in (MLPConfig(), MLPConfig(num_hiddens=512, num_blocks=8)):
    n = sum(p.numel() for p in build(cfg).parameters())
    print(f'num_hiddens={cfg.num_hiddens}, num_blocks={cfg.num_blocks}: '
          f'{n:,} parameters')
```

```
num_hiddens=256, num_blocks=4: 729,866 parameters
num_hiddens=512, num_blocks=8: 4,609,546 parameters
```

Doubling the hidden width quadruples the parameter count inside each block, and doubling the number of blocks doubles that total, so the larger config contains more than six times as many parameters. Real-world foundation models follow the exact same pattern: our 8B language model is specified by a compact config (e.g., 32 blocks, hidden width 4,096) paired with builder code that materializes the module tree.

Keep in mind that a config stores hyperparameters, not implementation code. Restoring a model from a checkpoint requires all three pieces: the weights, the config, and the matching

`build` function and block definitions. In Section 5.6, we save the weights and config together, while version control takes care of the code.

So far we have only located and counted parameters without inspecting what lives inside them. In Section 5.2, we open up the tensors a model owns, account for how much memory training consumes, and see how to share or freeze weights.

5.1.6 Summary

Three core ideas anchor model construction. First, every neural network is a tree of modules, with each parameter uniquely addressed by its path from the root. Second, optimizers, checkpoints, and device transfers all operate by walking registered children, so hiding layers inside an unregistered container silently excludes them from training and saving. Third, factoring variable dimensions into a lightweight config paired with a `build` function makes architectures easy to log, scale, and reproduce. On top of these fundamentals, custom forward methods give us full Python flexibility (subject to tracing constraints under compiled execution), while lazy layers offer convenience for quick experiments at the cost of deferring shape creation until a dry run.

5.1.7 Exercises

1. code **Unregistered modules.** The plain list in `PlainListMLP` keeps its layers out of `net.parameters()`, `net.to(device)`, and `net.state_dict()`. Predict, then verify, what three further operations do to the model.
    1. Call `net.apply(init_xavier)` and check whether the weight of `net.layers[0]` changes.
    2. Add `nn.Dropout(0.5)` after the `nn.ReLU()` in the list, call `net.eval()`, and pass the same input through the model twice.
    3. The method `load_state_dict` is the inverse of `state_dict`: it copies a dictionary of saved tensors into a model. Call `net.load_state_dict(ModuleListMLP().state_dict())`.
    4. Which of these failures are silent and which raise an error? Explain all three from the same missing registration.
2. **Widening a residual block.** `ResidualBlock` requires its input and output widths to agree. Suppose a block maps width d to a larger width d', so its body ends in a d'-wide layer and the skip path must also produce width d'. Give one way to build that skip path with parameters and one without. For each, count the extra parameters and state which output coordinates still receive the identity path.
3. code **A module from scratch.** ● In plain Python and NumPy, without any framework base class, write a `Parameter` class that wraps an array and a minimal `Module` class whose `__setattr__` registers every assigned `Parameter` or `Module`. Give `Module` a `parameters()` method that recurses into the registered children. Write a `Linear` subclass of `Module` that holds a weight and a bias, build a two-layer MLP from two instances of it, and verify that `parameters()` returns all four arrays. Then store the two layers in a plain list, check what `parameters()` returns, and extend `Module` so that the list version finds all four arrays as well.

    Adapted from CMU 10-714, Homework 2, Question 2.

4. code **Lazy input width.** Build `net = nn.Sequential(nn.LazyLinear(256), nn.ReLU(), nn.LazyLinear(10))` and pass a batch with 20 features through it, then a batch with 30 features. Explain the outcome from what the first forward pass did to each layer. Then describe a model that accepts inputs of either width while sharing every layer after the first.

5.2 Parameters, State, and Memory

In the previous section, we organized models as trees of modules so frameworks can find every parameter automatically. Until now we could let the framework manage those tensors behind the scenes without asking what else a model holds or how much memory training requires. At small scale, those details rarely bite. At large scale, they dominate every engineering decision. Consider an 8-billion-parameter model such as the Llama 3 model from Section 5.1. Storing its weights in 32-bit floating point takes 32 GB, yet *training* it requires four times as much (128 GB) before processing a single example: every parameter needs a matching gradient, and the Adam optimizer (Section 9.5) maintains two additional tensors of the same size for running averages of the gradients and squared gradients. That already exceeds the memory of an 80 GB GPU.

Computing such budgets accurately requires knowing every tensor a model owns and the role each one plays. Everyday workflows depend on the exact same inventory: checkpoints must capture every persistent tensor the forward pass depends on, GPU transfers must move them all together, and fine-tuning must specify cleanly which tensors the optimizer may update.

Frameworks unify all of these operations through *registration* (Section 5.1): each layer registers its tensors into the module tree under structured names, allowing optimizers, checkpoint savers, and device-transfer routines to walk that single tree. This section surveys that state tree in three parts. First, we distinguish trainable *parameters* from non-trainable *buffers*. Second, we account for training memory in bytes per parameter and show how to count parameters without allocating their storage. Third, we cover two recurring parameter patterns, weight tying and parameter freezing, used throughout language modeling and transfer learning.

```
import torch
from torch import nn
```

5.2.1 The State Tree

5.2.1.1 Access by Path

Our running example is a compact residual MLP: an input layer projecting 20 features to 64 hidden units, two 64-wide residual blocks, and a 10-unit output layer. We redefine `ResidualBlock` from Section 5.1 here so the notebook stands on its own: each block adds the output of a two-layer *body* directly to its input.

```
class ResidualBlock(nn.Module):
    def __init__(self, num_hiddens):
        super().__init__()
```

```
            self.body = nn.Sequential(
                nn.Linear(num_hiddens, num_hiddens), nn.ReLU(),
                nn.Linear(num_hiddens, num_hiddens))

        def forward(self, X):
            return X + self.body(X)

    net = nn.Sequential(nn.Linear(20, 64), ResidualBlock(64),
                        ResidualBlock(64), nn.Linear(64, 10))
    X = torch.randn(2, 20)
    net(X).shape
```

```
torch.Size([2, 10])
```

Parameters sit at the leaves of the module tree. The dotted path to any leaf (Figure 5.1) mirrors how we index into it in Python, which is handy for inspecting a single layer during debugging. Optimizers and checkpoint writers, however, need *every* leaf at once, so each framework provides a full-tree traversal yielding all parameters alongside their paths.

In PyTorch it is `named_parameters()`. Each parameter is an `nn.Parameter`, a tensor that registers itself with the module system as trainable, and its gradient `.grad` stays `None` until backpropagation fills it.

This residual MLP contains 12 parameter tensors in total. Below we inspect the total count along with the four tensors belonging to the first residual block:

```
    named = list(net.named_parameters())
    len(named), [(name, p.shape) for name, p in named[2:6]]
```

```
(12,
 [('1.body.0.weight', torch.Size([64, 64])),
  ('1.body.0.bias', torch.Size([64])),
  ('1.body.2.weight', torch.Size([64, 64])),
  ('1.body.2.bias', torch.Size([64]))])
```

A name like `1.body.0.weight` traces the path from child `1` of `net` (the first residual block) through `body` and its first layer `0` to `weight` (`net[1].body[0].weight`). In Gluon, `grad_req == 'write'` marks trainable parameters; in PyTorch, `state_dict()` keys match `named_parameters()` whenever the model has no buffers:

```
    list(net.state_dict()) == [name for name, _ in net.named_parameters()]
```

```
True
```

Every framework utility traverses this same state tree under these paths. The equality above holds only because every tensor in our residual MLP happens to be trainable, which is not true of all layers.

5.2.1.2 Parameters and Buffers

Suppose a model standardizes its inputs using a feature-wise mean and standard deviation computed once from the training set. Those statistics must be saved inside checkpoints

(without them, predictions at inference time would be wrong) and moved onto the GPU alongside the weights, yet the optimizer must *never* update them via gradient descent, because they are data statistics rather than learned weights. Storing them as plain Python attributes fails, since module traversals ignore plain attributes (Section 5.1). Batch normalization (Section 7.3, (Ioffe and Szegedy 2015)) relies on the exact same category of state: running means and variances updated during training forward passes and applied during inference.

Such persistent, non-trainable tensors are commonly called *buffers*, and every framework lets us register them so checkpoints and device moves include them while optimizers skip them. This yields a simple three-way rule for every tensor inside a layer: register it as a *parameter* if the optimizer should update it, register it as a *buffer* if checkpoints and device transfers need it without gradients, and keep it as an ordinary local variable if it lives only for one forward/backward pass (such as intermediate activations cached by backpropagation in Section 4.3). Architectural constants like layer widths remain plain Python attributes. Table 5.1 collects these rules for later reference.

Table 5.1: The four kinds of value in a layer, and what the optimizer, a checkpoint and a device move do with each.

Kind	Example	Updated by the optimizer	Saved in checkpoints	Moved with the model
parameter	weights, biases	yes	yes	yes
buffer	input statistics, BatchNorm running means	no	yes	yes
plain attribute	layer widths	no	no	no
local variable	intermediate activations	no	no	no

To illustrate, we build a `Standardizer` module that normalizes its inputs with a precomputed per-feature mean and standard deviation before applying a linear layer.

`self.register_buffer(name, tensor)` registers a non-trainable buffer.

```
class Standardizer(nn.Module):
    def __init__(self, mean, std):
        super().__init__()
        self.register_buffer('mean', mean)
        self.register_buffer('std', std)
        self.out = nn.Linear(4, 2)

    def forward(self, X):
        return self.out((X - self.mean) / self.std)

sample = torch.randn(100, 4) * torch.arange(1., 5.)
standardizer = Standardizer(sample.mean(0), sample.std(0, correction=0))
list(standardizer.state_dict())
```

```
['mean', 'std', 'out.weight', 'out.bias']
```

Inspecting the trainable vs. non-trainable state confirms that checkpoints see all four tensors while optimizers see only the linear layer's weight and bias:

```
[name for name, _ in standardizer.named_parameters()]
```

```
['out.weight', 'out.bias']
```

Registration governs more than checkpointing and optimization: whole-model operations such as dtype casting (Section 5.5) and GPU transfers (Section 5.7) traverse the exact same state tree, moving parameters and buffers alike while leaving unregistered plain attributes behind.

5.2.2 Training Memory

5.2.2.1 Bytes per Parameter

How much memory does it take to train our residual MLP? Focusing on model and optimizer state alone, memory scales with three per-parameter components: the parameter itself, its gradient, and any per-parameter optimizer state. Plain minibatch SGD keeps no state, so in 32-bit floating point (fp32, 4 bytes per number) it consumes 8 bytes per parameter. Adam adds two fp32 tensors per parameter: exponential moving averages of the gradient and of the squared gradient. Because nearly all large-scale training runs use Adam or a close variant, that 16-byte figure is the practical baseline:

Training state	Precision	Bytes per parameter
Weights	fp32	4
Gradients	fp32	4
Adam's average of the gradients	fp32	4
Adam's average of their squares	fp32	4
Total		16

These 16 bytes per parameter represent a strict lower bound. They exclude intermediate *activations* saved during the forward pass for backpropagation (which grow with batch size and sequence length), temporary kernel workspaces, and allocator fragmentation, all of which we measure directly on a GPU in Section 5.7. For our residual MLP, that baseline comes out to:

```
n = sum(p.numel() for p in net.parameters())
weights, grads, adam_state = 4 * n, 4 * n, 8 * n   # bytes, all fp32
print(f'{n} parameters: {(weights + grads + adam_state) / 2**20:.2f} MiB '
      'for weights + gradients + Adam state')
```

```
18634 parameters: 0.28 MiB for weights + gradients + Adam state
```

We can double-check that count by hand: the input layer contributes $20 \times 64 + 64 = 1344$ parameters, each residual block contributes $2 \times (64 \times 64 + 64) = 8320$, and the output layer adds $64 \times 10 + 10 = 650$, totaling 18,634 parameters. Adam's optimizer state is not just a theoretical entry in a table: it is real memory allocated tensor by tensor, adding two full copies of the model:

```
adam = torch.optim.Adam(net.parameters())
net(X).sum().backward()
adam.step()
moments = sum(t.numel() for state in adam.state.values()
              for t in state.values() if t.ndim > 0)
moments == 2 * n
```

```
True
```

`Adam` allocates the two moments of each parameter on its first `step()`.

For our tiny MLP, that baseline is a negligible 0.28 MiB, which is why memory barely crossed our minds in earlier chapters. Scale the exact same arithmetic to an 8-billion-parameter model, however, and you immediately hit 128 GB regardless of batch size. You might hope that 16-bit *mixed-precision* training (Section 5.5) cuts this static floor in half. Surprisingly, it does not: mixed precision keeps fp32 master copies of the weights and fp32 Adam moments so tiny updates do not vanish under rounding (Micikevicius et al. 2018), and the 16-bit values that the forward pass uses are cast from them.

5.2.2.2 Counting Without Allocating

Every parameter count above inspected tensors that had already been allocated. For an 8B model, that gets the order backwards: checking whether 32 GB of weights fit in RAM should not require allocating 32 GB first. Fortunately, counting parameters requires only shapes and dtypes, both of which are determined purely by constructor arguments. Consider the output projection of Llama 3 8B, mapping a 4,096-dimensional hidden state to logits over 128,256 vocabulary *tokens*. Its weight matrix has $4096 \times 128{,}256 = 525{,}336{,}576$ entries (2.1 GB in fp32). Below, we instantiate and count this layer without allocating its storage.

Constructing a module inside `with torch.device('meta'):` creates *meta* tensors that carry shapes and dtypes without allocating storage:

```
with torch.device('meta'):
    head = nn.Linear(4096, 128256, bias=False)   # shapes, no data
n_head = sum(p.numel() for p in head.parameters())
print(head.weight.device, f'{n_head:,} parameters, '
      f'{4 * n_head / 1e9:.1f} GB in fp32')
```

```
meta 525,336,576 parameters, 2.1 GB in fp32
```

To materialize a meta-device model without wasting time on random initialization before loading a checkpoint, call `to_empty(device=...)` and then `load_state_dict`:

```
with torch.device('meta'):
    clone = nn.Sequential(nn.Linear(20, 64), ResidualBlock(64),
                          ResidualBlock(64), nn.Linear(64, 10))
clone = clone.to_empty(device='cpu')   # memory, no values yet
clone.load_state_dict(net.state_dict())
torch.equal(clone(X), net(X))
```

```
True
```

Running the same abstract inspection across every layer of an 8B model yields its exact 8-billion parameter count, along with the 128 GB training floor, without allocating a single byte of tensor memory.

5.2.3 Shared and Frozen Parameters

5.2.3.1 Tied Parameters

A single parameter tensor can play multiple roles inside a network. The classic example comes from language modeling (Section 8.3), where a model reads tokens from a vocabulary V and predicts the next token. Its input *embedding* is a $|V| \times d$ lookup table whose v^{th} row is the d-dimensional representation of token v, while its output head projects a d-dimensional hidden state back to $|V|$ token logits: another $|V| \times d$ table with the exact same geometry and semantic meaning. *Tying* both layers to one shared weight matrix eliminates $|V| \times d$ parameters and often improves generalization (Press and Wolf 2017; Inan et al. 2017): tokens that behave interchangeably as inputs naturally deserve similar scores as outputs, and weight tying enforces a single unified token geometry. The memory savings can be dramatic: in GPT-2 small (Radford et al. 2019), the tied 50257×768 table accounts for roughly 38.6 million of the model's 124 million parameters, nearly a third of the entire network.

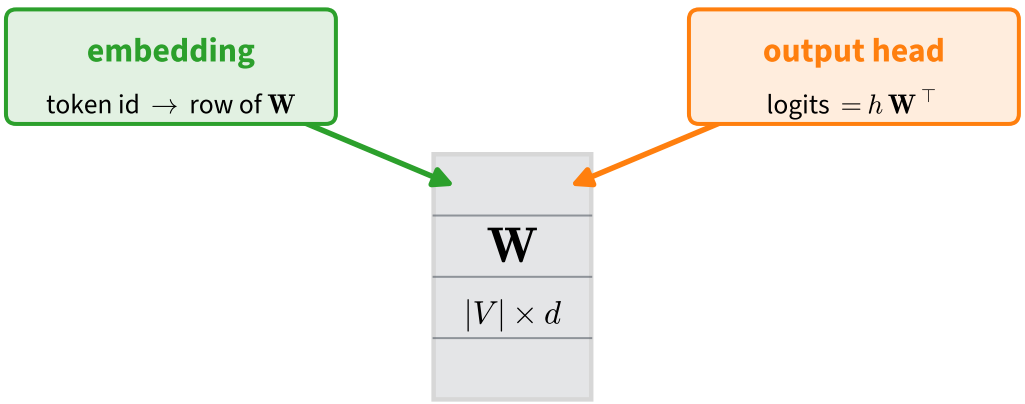


Figure 5.3: One weight matrix W, two call sites: the embedding looks up a row by token id, the output head multiplies by its transpose to produce logits. Tying means both point at the same tensor, so gradients from both uses sum into one place.

Figure 5.3 illustrates the two call sites. We verify the mechanics on a miniature language model, `ToyLM`, with a 100-token vocabulary and hidden width $d = 16$.

Assigning `self.head.weight = self.emb.weight` aliases both layers to the same `nn.Parameter` tensor:

```
class ToyLM(nn.Module):
    def __init__(self, vocab_size, d, tied=True):
        super().__init__()
        self.emb = nn.Embedding(vocab_size, d)
        self.body = nn.Linear(d, d)
        self.head = nn.Linear(d, vocab_size, bias=False)
        if tied:
            self.head.weight = self.emb.weight   # one tensor, two roles

    def forward(self, tokens):
        return self.head(torch.relu(self.body(self.emb(tokens))))

lm = ToyLM(vocab_size=100, d=16)
lm.head.weight is lm.emb.weight
```

```
True
```

Does the shared matrix get counted once or twice? We compare the tied model against an untied counterpart.

```
untied = ToyLM(vocab_size=100, d=16, tied=False)
(sum(p.numel() for p in lm.parameters()),
 sum(p.numel() for p in untied.parameters()))
```

```
(1872, 3472)
```

The tied model contains $1600 + 272 = 1872$ parameters (one embedding table plus the body layer), whereas the untied version adds a second 100×16 table for 3472 parameters in all.

Both `collect_params()` and `state_dict()` retain both keys (`emb.weight` and `head.weight`) pointing to the shared storage:

```
sd = lm.state_dict()
(sd['emb.weight'].data_ptr() == sd['head.weight'].data_ptr(),
 [name for name, _ in lm.named_parameters()])
```

```
(True, ['emb.weight', 'body.weight', 'body.bias'])
```

What happens during backpropagation when a tensor is used twice? By the multivariable chain rule, each call site contributes its own gradient term, and both contributions sum into the gradient of the shared tensor. Writing $\mathbf{W}_{\text{emb}}$ and $\mathbf{W}_{\text{head}}$ for the two tables of the untied model initialized to the same values as the tied matrix $\mathbf{W}$, the two models compute the exact same function and satisfy

$$\frac{\partial L}{\partial \mathbf{W}} = \frac{\partial L}{\partial \mathbf{W}_{\text{emb}}} + \frac{\partial L}{\partial \mathbf{W}_{\text{head}}}. \tag{5.3}$$

We verify this identity numerically by copying the tied weights into the untied model and comparing gradients on the same batch:

```
untied.load_state_dict(lm.state_dict())   # same values, separate tensors
tokens = torch.randint(0, 100, (2, 8))
lm(tokens).sum().backward()
untied(tokens).sum().backward()
torch.allclose(lm.emb.weight.grad,
               untied.emb.weight.grad + untied.head.weight.grad)
```

```
True
```

5.2.3.2 Frozen Parameters

When fine-tuning a pretrained network on a new task (Section 32.2), a common strategy is to keep the pretrained backbone fixed and train only a fresh output head, dramatically cutting both data requirements and optimizer memory. Locking parameters against gradient updates is called *freezing*. Below we freeze every layer except the output head of a fresh residual MLP, leaving just 650 of its 18,634 parameters trainable.

Setting `p.requires_grad = False` on backbone parameters excludes them from gradient accumulation:

```
finetune = nn.Sequential(nn.Linear(20, 64), ResidualBlock(64),
                         ResidualBlock(64), nn.Linear(64, 10))
for p in finetune[:-1].parameters():
    p.requires_grad = False

(sum(p.numel() for p in finetune.parameters() if p.requires_grad),
 sum(p.numel() for p in finetune.parameters()))
```

```
(650, 18634)
```

Taking one Adam step should now update only the output head while leaving all earlier layers untouched. We check every parameter before and after the step.

```
head_opt = torch.optim.Adam(
    (p for p in finetune.parameters() if p.requires_grad), lr=0.1)
before = [p.clone() for p in finetune.parameters()]
finetune(X).sum().backward()
head_opt.step()
[not torch.equal(b, p) for b, p in zip(before, finetune.parameters())]
```

```
[False,
 False,
 False,
 False,
 False,
 False,
 False,
 False,
 False,
 False,
 True,
 True]
```

Only the output head's weight and bias changed, and the optimizer allocated Adam moments for just those 650 trainable parameters. Freezing comes with two subtle pitfalls, neither of which raises an error. First, freezing a parameter does *not* reclaim optimizer state that has already been allocated. If an Adam optimizer was constructed over all parameters and has already stepped, it continues holding 8 bytes per parameter of running moments even for layers frozen later: harmless for 18,634 parameters, but costly at 8-billion-parameter scale (as explored in the exercises).

The second pitfall is even easier to overlook: freezing blocks gradients, yet some layer state mutates *without* gradients. In particular, batch normalization (Section 7.3) updates its running mean and variance during the *forward pass* whenever the layer runs in training mode, completely bypassing the optimizer. Whether freezing a BatchNorm layer's trainable weights also locks its running statistics therefore depends on the framework. Below we pass a shifted batch through a frozen BatchNorm layer in training mode, with no optimizer step at all, and check whether its running mean drifts.

```
bn = nn.BatchNorm1d(4)
for p in bn.parameters():
    p.requires_grad = False
before = bn.running_mean.clone()
bn(torch.randn(8, 4) + 3)          # training mode: the statistics update
torch.allclose(bn.running_mean, before)
```

```
False
```

The running mean moved. Freezing parameter gradients (`requires_grad=False`) is separate from switching the layer to evaluation mode (`bn.eval()`). Locking a BatchNorm layer during fine-tuning requires setting both.

Every parameter we inspected in this section (tied, frozen, or averaged) began from initial values supplied by the framework. In the next section, we examine how those initial values are chosen and how to control initialization across the entire module tree.

5.2.4 Summary

A model's persistent state forms a tree of named tensors, and registration determines which framework operations see each tensor: optimizers update *parameters*, whereas checkpoints and device transfers also include *buffers* (persistent tensors without gradients, such as running statistics), and unregistered attributes are silently skipped. Training in fp32 with Adam requires a baseline of 16 bytes per parameter before accounting for activations (weights, gradients, and two Adam moments, or 128 GB for an 8B model), and abstract shape inspection lets us compute such budgets without allocating memory. Weight tying aliases one tensor across multiple roles so it is counted once and receives the sum of gradients from all call sites; freezing halts gradient updates but does not release existing optimizer state, and need not stop forward-pass updates to BatchNorm running statistics.

5.2.5 Exercises

1. code **Running statistics.** This problem needs only one fact about the batch normalization layer of Section 7.3, the update below. In training mode the layer normalizes with the minibatch mean $\hat{\mu}_{\mathcal{B}}$ and variance, and each training step updates the running mean $\bar{\mu}$ by

$$\bar{\mu} \leftarrow \rho\,\bar{\mu} + (1-\rho)\,\hat{\mu}_{\mathcal{B}}. \tag{5.4}$$

1. A fresh layer starts from $\bar{\mu} = \mathbf{0}$. Predict $\bar{\mu}$ after the layer has processed k batches in training mode, each with minibatch mean 3 in every feature. Check the prediction for $k = 1, 5, 20$ on a fresh layer over 4 features, without any optimizer step, and read off the value of ρ the layer uses by default.
2. Suppose you registered $\bar{\mu}$ as a trainable parameter and removed the update above. What gradient does the training loss send to $\bar{\mu}$? How do the training loss and the evaluation accuracy then compare with those of an unmodified model?
3. Find a loss in $\bar{\mu}$ alone whose gradient step with learning rate η on each minibatch reproduces the removed update, and state the η that matches ρ.

2. code **Memory footprint.** Write a helper that reports, for each top-level block of `net`, its number of state entries and its bytes under fp32 Adam training with every parameter trained.
   1. Which block dominates? At what output width would the output layer overtake one residual block?
   2. Insert a batch normalization layer over the 64 hidden features between the two residual blocks. Extend the helper so that it costs the layer's running statistics correctly, and report each block's bytes per entry. For which blocks is the share of the total bytes larger than the share of the total entries, and for which is it smaller? Explain both directions.
3. code **Tying and freezing.** Consider the tied `ToyLM` of this section.
   1. Freeze `lm.emb.weight` but leave `lm.head` alone. How many trainable parameters remain, and what does this show about the interaction between tying and freezing?
   2. Starting from the values of `lm`, build a model whose embedding stays fixed while its head adapts, and verify both properties after one optimizer step. How many parameters does it store and how many does it train, compared with a fully trainable `lm`?
4. **Fine-tuning memory.** A language model has 8 billion parameters. Its output layer is a $4096 \times 128{,}256$ matrix, not tied to the embedding. All state is fp32, and the optimizer is Adam.
   1. How many bytes do weights, gradients, and Adam's running averages take when every parameter is trained? How many 80 GB GPUs does that state alone fill?
   2. Freeze everything except the output layer, and build the optimizer over the output layer only. How many bytes does the persistent state take now, and does it fit on one 80 GB GPU?
   3. Suppose instead that an optimizer built over all parameters took one step before the freeze and is kept afterwards. How many bytes remain allocated, and does the state still fit?

5.3 Initialization

Every neural network we have trained so far started from random weights, usually drawn by the framework's default initializer. Yet Section 4.4 showed just how unforgiving that starting point can be: if each layer slightly amplifies or attenuates activations, a deep stack compounds the factor exponentially, stalling training before it even begins.

That section also derived the remedy for a single layer: draw weights with standard deviation $g/\sqrt{n_{\text{in}}}$, where n_{in} is the layer's input width and the *gain* g compensates for the activation function that follows: $g = \sqrt{2}$ before a ReLU (He initialization, Section 4.4.2.2) and $g = 1$ for a linear output. Under that rule, each layer preserves the mean square of its input.

You might expect that applying this rule to every layer settles initialization once and for all. Surprisingly, it does not. In a residual network, activations still explode with depth even when every individual layer obeys He initialization. Each residual block adds the output of its body f (Section 5.1) to its input. If the body preserves the input mean square and produces an output uncorrelated with the input, their sum has *double* the mean square. Across 32 residual blocks, the activation standard deviation balloons by $2^{16} \approx 65{,}000$ before a single gradient step is taken.

Sound initialization therefore depends on the *architecture's wiring*, not just individual layer types, and we need clean mechanisms for applying custom rules across a module tree. The first half of this section covers the mechanics: applying initializers across an entire module tree, bounding extreme draws with truncated normals, writing custom initializers, and editing weights in place. Slow down for the second half, where we analyze variance growth across residual blocks and fix it by scaling down or zeroing each block's output projection.

5.3.1 Applying Initializers

As we saw in Section 4.4, default linear-layer initializers do not apply He initialization (Section 4.4.2.2). To apply He initialization (or the depth-dependent residual scaling we derive shortly) across an entire model, we walk the module tree (Section 5.1). Below we compare default versus He initialization on a 10-layer ReLU stack of width 256 (without biases) fed 1,024 standard-normal inputs.

PyTorch's default draws from $U(-1/\sqrt{n_{\text{in}}}, 1/\sqrt{n_{\text{in}}})$ (one-sixth of He's variance). `net.apply(init_he)` invokes `init_he` on every submodule, using the in-place `nn.init.kaiming_normal_` (trailing underscore) so existing `nn.Parameter` references remain intact:

```
net = nn.Sequential(*[module for _ in range(10) for module in
                      (nn.Linear(256, 256, bias=False), nn.ReLU())])
X = torch.randn(1024, 256)
print(f'default: mean square of the output {net(X).pow(2).mean():.2g}')

def init_he(module):
    if isinstance(module, nn.Linear):
        nn.init.kaiming_normal_(module.weight, nonlinearity='relu')

net.apply(init_he)
print(f'He:      mean square of the output {net(X).pow(2).mean():.2g}')
```

```
default: mean square of the output 1.9e-08
He:      mean square of the output 1.1
```

Under the default initializer, the output mean square collapses by orders of magnitude because every layer multiplies variance by a factor strictly less than one. Switching to He initialization brings the output mean square right back to within roughly a factor of two of 1 (the input mean square), with the remaining spread coming simply from random finite-width fluctuations compounding across ten layers.

This pattern is the whole interface. Later chapters package it as helper functions, such as `init_cnn` for convolutional networks, but each is a function like `init_he` handed to `apply`.

5.3.1.1 Truncated Normals

A standard Gaussian gets the variance right, but its tails extend to infinity. For a tiny layer that rarely matters; across the 10^8 weights of a large model, however, dozens of entries are virtually guaranteed to land beyond five standard deviations. To avoid extreme initial outliers, modern language and vision models such as BERT and the Vision Transformer draw from a *truncated normal* that rejects and redraws samples outside a fixed multiple of σ (Devlin et al. 2018; Dosovitskiy et al. 2021). Clipping the tails also trims the variance slightly, which some framework implementations compensate for by rescaling and others leave as is.

`nn.init.trunc_normal_` takes *absolute* cutoffs `a` and `b` (defaulting to ± 2), so truncating at $\pm 2\sigma$ for `std=0.02` requires passing `a=-0.04, b=0.04`:

```
w = torch.empty(1000, 1000)
nn.init.normal_(w, std=0.02)
print(f'normal:    std {w.std():.4f}, max weight {w.abs().max():.4f}')
nn.init.trunc_normal_(w, std=0.02, a=-0.04, b=0.04)
print(f'truncated: std {w.std():.4f}, max weight {w.abs().max():.4f}')
```

```
normal:    std 0.0200, max weight 0.0998
truncated: std 0.0176, max weight 0.0400
```

Out of one million standard normal draws with $\sigma = 0.02$, extreme weights approach $5\sigma = 0.1$, whereas the truncated normal guarantees $|w| \leq 0.04$. Clipping the tails lowers the empirical standard deviation from 0.02 to roughly 0.0176. Fixed-σ recipes like BERT simply accept that slight reduction, while variance-preserving schemes rescale after truncation to hit the exact target variance.

5.3.2 Custom Initializers and Direct Edits

Whenever built-in initializers do not match what a paper or architecture calls for, we can write our own initializer or edit parameter tensors directly. Both build on each framework's initializer interface:

In PyTorch, an initializer is a function that overwrites a parameter in place.

5.3.2.1 A Rule with a Gain of Our Choice

We now implement the gain-parameterized normal rule from the introduction: draw weights with standard deviation $g/\sqrt{n_{\text{in}}}$ for any chosen gain g. He initialization corresponds to $g = \sqrt{2}$, and our residual-network experiments below will also test $g = 1$, $g = 1/\sqrt{N}$ (for N blocks), and $g = 0$ on each block's output projection. Writing one reusable initializer handles all four cases cleanly.

The function follows the convention of `torch.nn.init`: it overwrites the tensor it is given, and its name ends in an underscore. A PyTorch weight has shape $(n_{\text{out}}, n_{\text{in}})$:

```
def scaled_normal_(weight, gain):
    nn.init.normal_(weight, std=gain / math.sqrt(weight.shape[1]))
```

5.3.2.2 Direct Edits

Occasionally we want to modify parameter values directly, either right after initialization or after loading weights. Because ReLU is positively homogeneous, halving the final layer's weights in our ten-layer stack halves every output activation and should therefore reduce the output mean square by a factor of four.

Wrap in-place edits on leaf `nn.Parameter` tensors in `with torch.no_grad():` so autograd does not reject them.

```
before = net(X).pow(2).mean()
with torch.no_grad():
    net[-2].weight *= 0.5
print(f'{before:.2g} before, {net(X).pow(2).mean():.2g} after')
```

```
1.1 before, 0.28 after
```

Sure enough, the output mean square drops to exactly one-quarter of its prior value.

5.3.3 Initialization of Residual Networks

5.3.3.1 Variance Growth across Blocks

Earlier we claimed that a deep residual stack explodes at initialization even when every individual layer follows He's rule. To prove it, note that a residual block computes $\mathbf{x}_{k+1} = \mathbf{x}_k + f_k(\mathbf{x}_k)$. Take the two-layer body from Section 5.1, expanded fourfold in the middle as in many modern residual blocks and transformers: a linear layer from width d to $4d$, a ReLU, and a linear *output projection* from $4d$ back to d. All biases start at zero. The first layer uses He initialization ($g = \sqrt{2}$) because a ReLU follows it, while the output projection uses standard deviation $g/\sqrt{4d}$ for a gain g that we leave free ($g = 1$ being the standard linear-layer rule from Section 4.4).

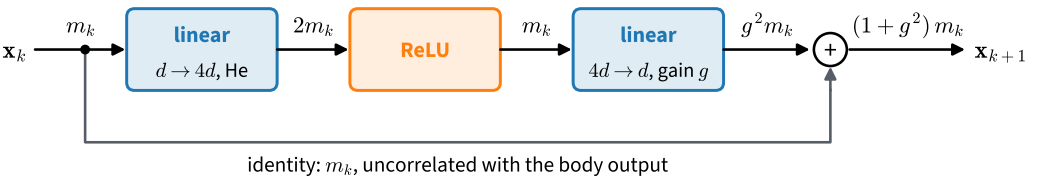


Figure 5.4: The mean square of the signal at each point of one residual block. The He layer doubles m_k, the ReLU halves it again, and the output projection multiplies it by g^2. The identity path carries m_k around the body, and the two mean squares add at the sum.

Let m_k denote the mean square of the entries of $\mathbf{x}_k$; Figure 5.4 follows it through the block. Passing through the first linear layer doubles m_k, and the ReLU halves it back to m_k (Section 4.4.2.2). The output projection then scales the mean square by $4d \cdot g^2/(4d) = g^2$, so the body output $f_k(\mathbf{x}_k)$ has mean square $g^2 m_k$. Now comes the crucial step: because the

output projection's weights are zero-mean and independent of $\mathbf{x}_k$, the body output $f_k(\mathbf{x}_k)$ is uncorrelated with the skip signal $\mathbf{x}_k$. Their mean squares therefore add:

$$m_{k+1} = m_k + g^2 m_k = (1 + g^2)\, m_k. \tag{5.5}$$

Every block inflates the activation mean square by $1 + g^2 > 1$ whenever $g \neq 0$. At $g = 1$, each block doubles the variance, so across N blocks the activation standard deviation grows by $2^{N/2}$: $2\times$ after 2 blocks, $16\times$ after 8 blocks, and $65{,}536\times$ after 32 blocks. Every single layer respected the single-layer variance rule, yet the network as a whole blew up because that local rule is blind to the skip connection adding the input back at every block.

5.3.3.2 Scaling Down the Output Projection

Across N blocks, the per-block factor compounds to $(1 + g^2)^N$. Keeping activations well scaled at arbitrary depth therefore requires shrinking g as N grows. Choosing $g^2 = 1/N$, scaling every body's output projection by $1/\sqrt{N}$, bounds the compounding cleanly:

$$m_N = \left(1 + \frac{1}{N}\right)^N m_0 \leq e\, m_0, \tag{5.6}$$

because $(1 + 1/N)^N$ increases monotonically to e. The activation standard deviation now grows by at most $\sqrt{e} \approx 1.65$ regardless of depth (1.5 after 2 blocks, 1.60 after 8, and 1.64 after 32). The same $1/\sqrt{N}$ scaling is widely used in practice: GPT-2 (Radford et al. 2019), for example, shrinks the initial standard deviation of the output projection of every residual *branch* by $1/\sqrt{N}$, where N is the total number of residual branches in the network. Its blocks normalize their inputs, so the growth is additive rather than multiplicative (see the exercises), and the same scaling keeps the sum bounded. Fixup (Hongyi Zhang et al. 2019) derives a depth-dependent scaling of this kind for residual networks that have no normalization layers at all.

5.3.3.3 Zero Output Projections

What if we push this idea to its limit and set $g = 0$? At first glance, zero initialization sounds dangerous: zeroing *every* weight traps all hidden units in identical symmetry forever (Section 4.4). Zeroing *only* the output projection of each residual body is completely different. At step zero, every body outputs zero, so every block acts as an exact identity map and the entire network starts out cleanly behaved while depth engages gradually during training. Symmetry never freezes learning because the body's first layer retains its random weights: on the first backward pass, the output projection receives a nonzero gradient (since its ReLU inputs are nonzero), moving away from zero immediately, and from the second step onward gradients flow freely into the first layer as well. Figure 5.5 lays out the three stages and contrasts them with the all-zeros trap.

Zero-initialized branches have a clear lineage. Goyal et al. (2017) zeroed the final scale factor of each residual block to train ResNets with large minibatches. Fixup combines zero output projections with its depth-dependent scaling. Later work replaces the zero matrix with a single learnable scalar per branch that starts at zero (De and Smith 2020; Bachlechner et al. 2021).

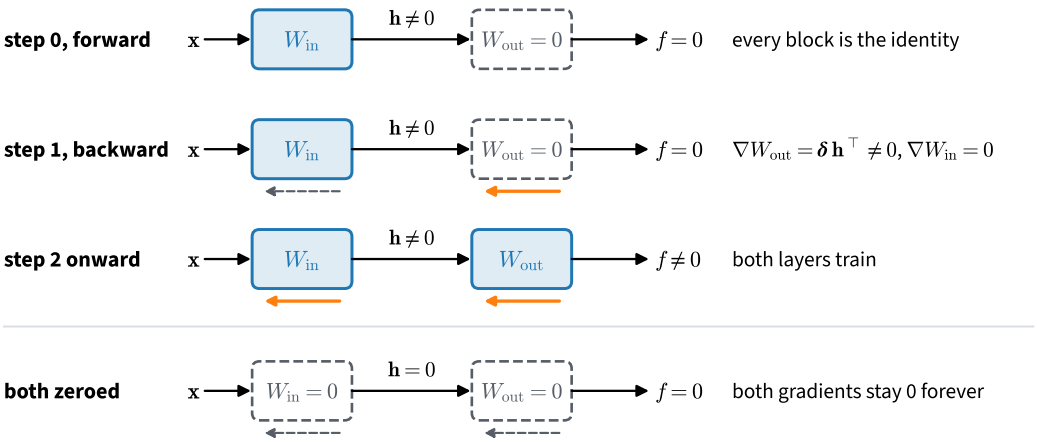


Figure 5.5: Zeroing only the output projection W_{out} of a residual body. In the first forward pass the body outputs $f = 0$, so the block is the identity, but the hidden activations $\mathbf{h}$ are nonzero. The first backward pass therefore gives W_{out} the nonzero gradient $\delta\mathbf{h}^\top$, where δ is the gradient arriving at the body output, while W_{in} receives none. From the next step on both layers train. Bottom row: with both layers zeroed, $\mathbf{h} = 0$ and neither layer ever receives a gradient.

5.3.3.4 Measuring Variance across Depth

We now test all three predictions empirically. `WideResidualBlock` implements our $d \to 4d \to d$ residual block using our custom gain-scaled normal initializer, with gain $\sqrt{2}$ on the expansion layer and a configurable `out_gain` on the output projection:

```
class WideResidualBlock(nn.Module):
    def __init__(self, d, out_gain):
        super().__init__()
        self.body = nn.Sequential(nn.Linear(d, 4 * d), nn.ReLU(),
                                  nn.Linear(4 * d, d))
        for layer, gain in ((self.body[0], math.sqrt(2)),
                            (self.body[2], out_gain)):
            scaled_normal_(layer.weight, gain)
            nn.init.zeros_(layer.bias)

    def forward(self, X):
        return X + self.body(X)
```

Next, `std_across_depth` builds a stack of `num_blocks` blocks of width 64, feeds in 256 $\mathcal{N}(0, 1)$ inputs, and tracks the activation standard deviation at the input and after every block:

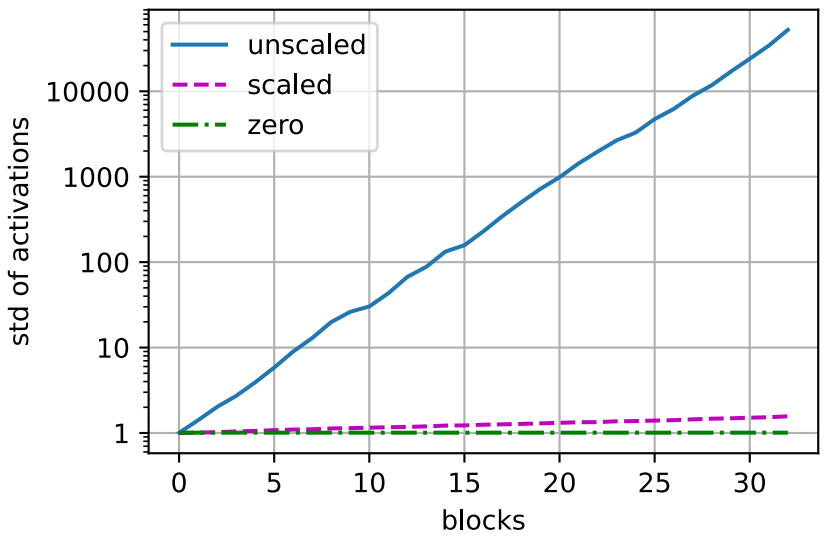
```
def std_across_depth(num_blocks, out_gain):
    net = nn.Sequential(*[WideResidualBlock(64, out_gain)
                          for _ in range(num_blocks)])
    X = torch.randn(256, 64)
    stds = [X.std().item()]
```

```
    with torch.no_grad():
        for block in net:
            X = block(X)
            stds.append(X.std().item())
    return stds
```

We compare all three gain schedules: unscaled ($g = 1$), scaled ($g = 1/\sqrt{N}$), and zero ($g = 0$), printing final standard deviations for $N = 2, 8, 32$ and plotting the full depth profile at $N = 32$:

```
gains = {'unscaled': lambda n: 1.0, 'scaled': lambda n: n ** -0.5,
         'zero': lambda n: 0.0}
profiles = {n: [std_across_depth(n, gain(n)) for gain in gains.values()]
            for n in (2, 8, 32)}
print(f'{"N":>3}' + ''.join(f'{name:>10}' for name in gains))
for n, stds in profiles.items():
    print(f'{n:>3}' + ''.join(f'{s[-1]:>10.3g}' for s in stds))
d2l.plot(list(range(33)), profiles[32], 'blocks', 'std of activations',
         legend=list(gains), yscale='log', figsize=(4.5, 3))
```

```
  N  unscaled    scaled      zero
  2      2.04      1.49     0.996
  8      13.8      1.55     0.988
 32  5.24e+04      1.57      1.01
```



The empirical curves confirm our analysis. The unscaled stack grows by roughly $\sqrt{2}$ per block (a straight line on the log scale), reaching anywhere from about 20,000 to 200,000 after 32 blocks (most individual runs sit slightly below the theoretical $\sqrt{\mathbb{E}[m_{32}]} = 65{,}536$ because multiplying random positive factors produces a right-skewed distribution whose median falls below the square root of its mean). With initial activations already in the tens of thousands, there is little point even attempting to train the unscaled stack. By contrast, the $1/\sqrt{N}$-scaled stack stays comfortably between 1.4 and 1.9 at every depth, matching our

1.5–1.65 prediction, while zeroing the output projections holds the standard deviation flat at 1 across all 32 blocks.

Running a quick forward-pass diagnostic like this catches broken initializations immediately before wasting GPU time on training. Notice, though, that it cannot tell you whether $1/\sqrt{N}$ scaling or zero initialization will optimize faster once training begins: both preserve signal scale at step zero, differing only in whether the residual branches start active or wake up over the first few updates.

Finally, remember that initialization controls only step zero. As gradient steps update the weights, nothing in initialization alone guarantees that activations stay well scaled, which is where normalization layers (Section 7.3) step in.

5.3.4 Summary

Framework default initializers do not use He initialization, so deep networks without normalization layers require explicit initialization (applied via constructor arguments, tree-wide callbacks, or direct in-place edits), and truncated normals prevent rare extreme outliers in large models. Crucially, per-layer variance rules (Section 4.4) are not enough for residual architectures: adding each block's body back to its input multiplies the activation mean square by $1 + g^2$, causing $g = 1$ to blow up by $\sqrt{2}$ per block. Scaling each block's output projection by $1/\sqrt{N}$ caps activation growth at $\sqrt{e}$ for any depth N, while zeroing the output projection starts every block as the identity without trapping the network in symmetry.

5.3.5 Exercises

1. **Truncated normal.** For a normal distribution truncated at $\pm 2\sigma$: what fraction of draws does the cut discard, and by how much does it shrink the standard deviation? Neither quantity has an elementary closed form; express both in terms of the standard normal density and distribution function and evaluate them using tabulated values. Verify the standard deviation against the printed output of the demonstration in Section 5.3.1.1, and the discarded fraction by counting the entries of its plain normal draw that lie beyond $\pm 2\sigma$.

2. code **Normalized inputs.** Many residual networks normalize the input of each body: the body reads $\mathbf{x}_k / r_k$, where r_k is the root mean square of the entries of $\mathbf{x}_k$ for each example, instead of $\mathbf{x}_k$.

    1. Show that the body output then has mean square g^2, whatever the scale of $\mathbf{x}_k$, and that with $m_0 = 1$ the mean square after N blocks is $1 + Ng^2$. How does the growth with depth compare with the unnormalized blocks of this section?
    2. Which gain keeps the mean square at the end of the stack equal to 2 for every N? Compare it with the scaling of GPT-2.
    3. Add the normalization to `WideResidualBlock`, measure the standard deviation across depth for $g = 1$ and $g = 1/\sqrt{N}$ at $N = 32$, and compare with your predictions and with the measured plot in Section 5.3.3.4.

3. code **Kaiming initialization.** Implement the Kaiming (He) initializers of Section 4.4.2.2, uniform and normal, with an explicit `nonlinearity` argument that switches the gain between `'relu'` and `'linear'`.

1. Check both on the weight of a layer with 512 inputs and 256 outputs by mean, standard deviation, and range: the normal initializer against the scaled normal initializer of this section, and the uniform initializer against its predicted bound and standard deviation. Explain any difference in range.
2. Apply your normal initializer to the stack of Section 5.3.3.4 with the `'relu'` gain on both layers of every block. Predict the factor by which each block multiplies the mean square, and check it against a measured curve added to the plot of Section 5.3.3.4.

Adapted from CMU 10-714, Homework 2 and Homework 4.

4. code **Gradients across depth.** ● The backward pass through a block multiplies the gradient by the transposed Jacobian of $\mathbf{x} \mapsto \mathbf{x} + f(\mathbf{x})$.
   1. Repeat the argument of this section for the backward pass. Show that the body multiplies the mean square of the gradient by g^2, although the backward pass sums over the outputs of each layer rather than its inputs, so that each block again multiplies it by $1 + g^2$.
   2. Verify the prediction for the three choices of gain at $N = 32$: compute the gradient of the sum of the outputs of the stack with respect to its input, and compare its mean square with that of the gradient at the output.
5. code **Data-driven calibration.** ● Initialize the stack of Section 5.3.3.4 with the unscaled choice of gain, then calibrate it block by block on one minibatch of unit-variance inputs. For each block in turn, run the minibatch forward to that block, measure the standard deviation of its body output $f_k(\mathbf{x}_k)$, and rescale the block's output projection so that this standard deviation becomes $1/\sqrt{N}$. This adapts the layer-sequential unit-variance initialization of Mishkin and Matas (2016) to a residual stack.
   1. Explain why the blocks must be calibrated in order rather than all at once from a single forward pass.
   2. Explain why the calibration targets the body output rather than the block output $\mathbf{x}_k + f_k(\mathbf{x}_k)$.
   3. Predict the standard deviation at the end of the calibrated stack. Plot the calibrated depth-versus-standard-deviation curve next to that of the scaled choice of gain, and explain the difference.

5.4 Custom Layers and Functions

A deep learning framework's built-in library covers the standard layers everyone needed yesterday: linear projections, activation functions, dropout. Research moves faster than release cycles, however, and a newly proposed layer can take years to appear in official libraries. *RMSNorm*, a normalization layer that rescales each activation vector to unit root-mean-square magnitude, is a case in point: published in 2019 (Zhang and Sennrich 2019), it took years to reach the standard layer libraries. Until then, every codebase that wanted RMSNorm either implemented it by hand or imported a third-party version. To behave like a first-class built-in layer, any custom layer we write must plug seamlessly into the framework: its parameters must reach the optimizer, its state must serialize into checkpoints and migrate onto GPUs, and it must slot into sequential and custom containers alongside built-in layers.

Sometimes the obstacle is not the forward operation at all, but its derivative. To squeeze

an 8-billion-parameter model (Section 5.1) into tight memory, practitioners often quantize weights down to 4-bit integers, cutting storage to just 4 GB (a quarter of 16-bit floating point). Quantizing a finished model needs no gradients. *Training* with quantized weights in the forward pass does, and there the rounding is in the way: it has derivative zero almost everywhere, so any weight quantized inside the forward pass receives zero gradient and stops learning.

In this section, we build both kinds of extensions. Because every framework operation (optimization, checkpointing, and device movement) walks registered state in the module tree (Section 5.1), subclassing the base module and registering our tensors is all it takes for a custom layer to participate automatically. We progress through four concrete examples: a stateless layer, `RMSNorm` with a learned parameter, a random-features layer holding fixed non-trainable buffers, and a custom backward pass (the *straight-through estimator*) that lets gradients flow through rounding.

5.4.1 Layers without Parameters

We start with the simplest possible custom layer: one that holds no state at all. `CenteredLayer` subtracts the global mean from its input tensor. To build it, we subclass the framework's base module and implement only the forward pass.

```
class CenteredLayer(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, X):
        return X - X.mean()
```

The vector `[1, 2, 3, 4, 5]` has mean 3, so `CenteredLayer` shifts it to run from -2 to 2:

```
layer = CenteredLayer()
layer(torch.tensor([1.0, 2, 3, 4, 5]))
```

```
tensor([-2., -1.,  0.,  1.,  2.])
```

Because `CenteredLayer` shares the exact same module interface as built-in layers, we can drop it directly into a sequential container after a linear layer to center its outputs. The resulting mean should vanish; in floating-point arithmetic, seeing a tiny residual around 10^{-8} or smaller is ordinary rounding error rather than a bug.

```
net = nn.Sequential(nn.Linear(8, 128), CenteredLayer())
Y = net(torch.rand(4, 8))
Y.mean()
```

```
tensor(2.5611e-09, grad_fn=<MeanBackward0>)
```

5.4.2 Layers with Parameters

In Section 4.4, we saw how activation scales can drift across depth depending on layer weights. A *normalization layer* removes that sensitivity by rescaling each activation vector to a controlled magnitude and giving the network a learned scale parameter to adjust

it instead. We explore this family in depth starting in Section 7.3; here, we implement its cleanest member, *RMSNorm*.

5.4.2.1 RMSNorm

Given an input vector $\mathbf{x} \in \mathbb{R}^d$ with root mean square $\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2}$, RMSNorm divides $\mathbf{x}$ by its RMS and scales the result elementwise by a learned *gain* vector $\mathbf{g} \in \mathbb{R}^d$:

$$\mathrm{RMSNorm}(\mathbf{x}) = \frac{\mathbf{x}}{\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2 + \epsilon}} \odot \mathbf{g}.$$

A small constant ϵ (such as 10^{-6}) guards against division by zero. With $\mathbf{g}$ initialized to $\mathbf{1}$, the output always has root mean square close to 1 regardless of the input's scale. For example, for $\mathbf{x} = (3, 4)$, the mean square is $(9 + 16)/2 = 12.5$ and the RMS is 3.536, yielding $(0.849, 1.131)$, and rescaling the input to $(300, 400)$ produces the exact same output. Geometrically (Figure 5.6), RMSNorm preserves the direction of $\mathbf{x}$ while projecting it onto the sphere of radius $\sqrt{d}$, after which $\mathbf{g}$ stretches each coordinate axis.

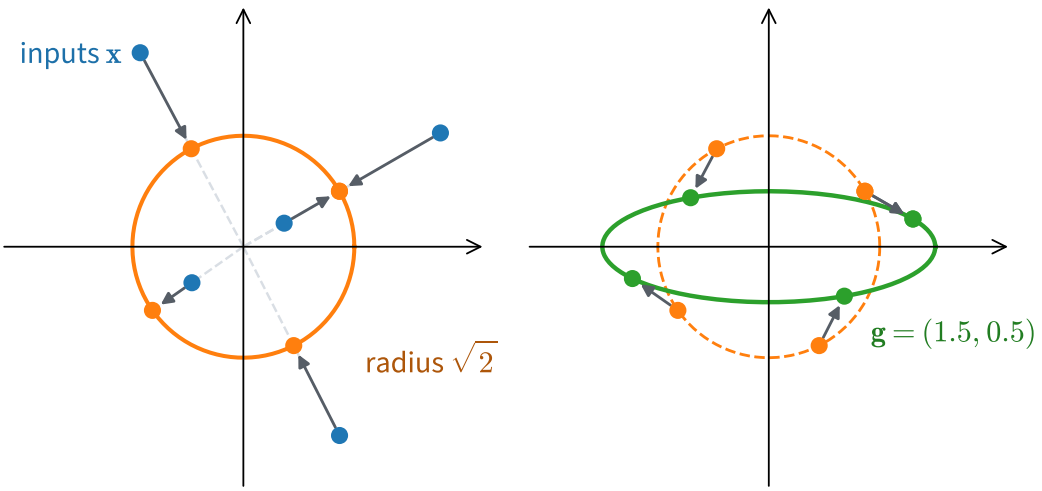


Figure 5.6: RMSNorm in two dimensions. Left: with gain 1, every input (blue) moves along its ray from the origin onto the circle of radius $\sqrt{2}$ (orange); two inputs on the same ray, at lengths 0.6 and 2.9, land on the same point. Right: the gain $\mathbf{g} = (1.5, 0.5)$ stretches the circle into an ellipse with semi-axes $1.5\sqrt{2}$ and $0.5\sqrt{2}$.

RMSNorm streamlines *layer normalization* (Ba et al. 2016), which additionally subtracts each example's mean (like a per-row `CenteredLayer`) and adds a learned bias after rescaling. Zhang and Sennrich showed that dropping mean-centering rarely hurts performance while cutting computation, which is why many modern language models, including Llama 3 8B (Grattafiori et al. 2024) (Section 5.1), adopt RMSNorm throughout.

Because $\mathbf{g}$ must be trained, the layer registers it as a parameter (Section 5.2); the rest of the implementation mirrors the formula directly.

In PyTorch, assigning a tensor wrapped in `nn.Parameter` to an attribute registers it.

```
class RMSNorm(nn.Module):
    def __init__(self, d, eps=1e-6):
        super().__init__()
        self.gain = nn.Parameter(torch.ones(d))
        self.eps = eps

    def forward(self, X):
        rms = X.pow(2).mean(-1, keepdim=True).add(self.eps).sqrt()
        return self.gain * X / rms
```

We test `RMSNorm` on $(3, 4)$ and on the 100-fold rescaled vector $(300, 400)$; both rows should map to $(0.849, 1.131)$:

```
norm = RMSNorm(2)
norm(torch.tensor([[3.0, 4.0], [300.0, 400.0]]))
```

```
tensor([[0.8485, 1.1314],
        [0.8485, 1.1314]], grad_fn=<DivBackward0>)
```

5.4.2.2 Registration

Why wrap RMSNorm inside a module subclass rather than calling a plain function with a standalone gain tensor? Because module registration automatically grants our custom layer all four capabilities of built-in layers: parameter discovery by optimizers, composability inside containers, checkpoint serialization, and device migration (Section 5.1, Section 5.2). We verify this end to end by sandwiching `RMSNorm` between two linear layers, taking one gradient step so the gain moves off ones, and copying the full state into a freshly initialized clone, after which both models should produce identical outputs (Section 5.6 extends this to disk checkpoints).

```
net = nn.Sequential(nn.Linear(20, 8), RMSNorm(8), nn.Linear(8, 2))
X = torch.randn(4, 20)
optimizer = torch.optim.SGD(net.parameters(), lr=0.1)
net(X).pow(2).sum().backward()
optimizer.step()
clone = nn.Sequential(nn.Linear(20, 8), RMSNorm(8), nn.Linear(8, 2))
clone.load_state_dict(net.state_dict())
net[1].gain.data, torch.equal(net(X), clone(X))
```

```
(tensor([0.9681, 0.9702, 0.9415, 0.7670, 1.0491, 0.9845, 0.9079, 0.9954]),
 True)
```

Sure enough, the gain updates away from ones and the clone matches the trained model bit for bit, with zero boilerplate inside `RMSNorm` beyond registering `self.gain`.

5.4.2.3 Comparison with Framework Implementations

Passing the integration checks proves our layer plugs into the framework properly, not that its math is bug-free. Whenever a library offers a native equivalent, testing our custom im-

plementation against it with non-trivial weights provides an immediate numerical sanity check.

We set a non-trivial gain (`linspace(0.5, 1.5, 8)`) on both our custom `RMSNorm` and the native `nn.RMSNorm` and verify that their outputs match:

```
ours, native = RMSNorm(8), nn.RMSNorm(8, eps=1e-6)
with torch.no_grad():
    ours.gain.copy_(torch.linspace(0.5, 1.5, 8))
    native.weight.copy_(ours.gain)
X = torch.randn(16, 8)
torch.allclose(ours(X), native(X))
```

```
True
```

Whenever a built-in library layer matches your specification, prefer it in production for maintenance and fused-kernel speed; write custom layers for novel operations or modifications the library lacks.

5.4.3 Layers with Fixed State

Some layers depend on arrays that are neither learned by gradient descent nor recomputed from scratch on every call: they are generated once at construction and must remain fixed for the lifetime of the model. A classic example is a *random features* layer (Rahimi and Recht 2007), which projects a minibatch $\mathbf{X}$ through a fixed standard-normal matrix $\mathbf{W} \in \mathbb{R}^{d \times D}$, shifts by random phases $\mathbf{b} \in \mathbb{R}^D$ drawn uniformly from $[0, 2\pi]$, and applies cosine elementwise:

$$\mathbf{Z} = \sqrt{2/D}\, \cos(\mathbf{X}\mathbf{W} + \mathbf{b}). \tag{5.7}$$

Each row of $\mathbf{Z}$ provides D fixed nonlinear features—remarkably, even without ever updating $\mathbf{W}$ or $\mathbf{b}$, a single trainable linear layer placed on top can approximate rich nonlinear functions.

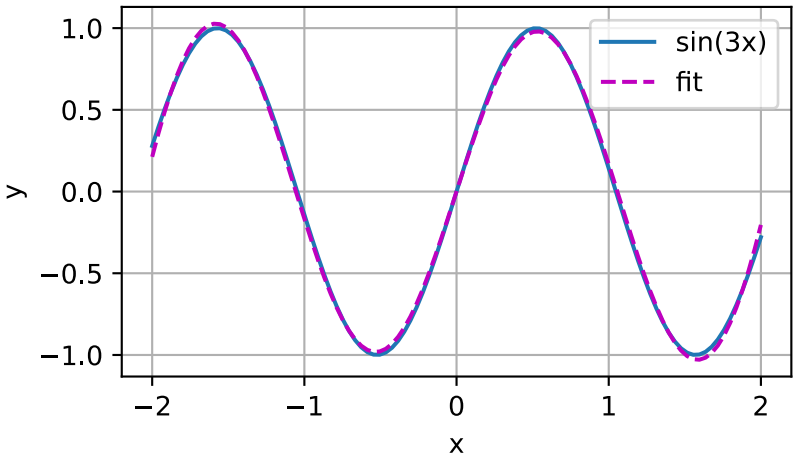
Just like the running statistics in `Standardizer` (Section 5.2.1.2), $\mathbf{W}$ and $\mathbf{b}$ must stay fixed during optimization yet travel with the model in checkpoints and across devices. We therefore register them as buffers:

```
class RandomFeatures(nn.Module):
    def __init__(self, d, num_features):
        super().__init__()
        self.register_buffer('W', torch.randn(d, num_features))
        self.register_buffer('b', 2 * torch.pi * torch.rand(num_features))

    def forward(self, X):
        return (2 / self.b.numel()) ** 0.5 * torch.cos(X @ self.W + self.b)
```

We now put `RandomFeatures` to work by fitting $y = \sin(3x)$ across 100 evenly spaced points in $[-2, 2]$ using 500 random features feeding a single linear layer trained with Adam. Because $\mathbf{W}$ and $\mathbf{b}$ are registered as buffers rather than parameters, handing the whole model to the optimizer updates only the linear head:

```
x = torch.linspace(-2, 2, 100)
X, y = x.reshape(-1, 1), torch.sin(3 * x)

def mse(net, X, y):
    return ((net(X)[:, 0] - y) ** 2).mean()

net = nn.Sequential(RandomFeatures(1, 500), nn.Linear(500, 1))
optimizer = torch.optim.Adam(net.parameters(), lr=0.05)
for _ in range(500):
    optimizer.zero_grad()
    mse(net, X, y).backward()
    optimizer.step()
print(f'mean squared error {mse(net, X, y):.2g}')
d2l.plot(x, [y, net(X)[:, 0].detach()], 'x', 'y',
         legend=['sin(3x)', 'fit'], figsize=(4.5, 2.5))
```

```
mean squared error 0.00054
```



The learned combination tracks $\sin(3x)$ closely, driving the mean squared error below 0.01 (compared with 0.5 for a constant zero baseline).

Now watch what happens if a checkpoint saves only the trained linear head: a freshly instantiated model draws a completely different $\mathbf{W}$ and $\mathbf{b}$, feeding the linear layer features it has never seen. Below we compare restoring just the linear head against restoring the full state, buffers included:

```
fresh = nn.Sequential(RandomFeatures(1, 500), nn.Linear(500, 1))
fresh[1].load_state_dict(net[1].state_dict())
print(f'linear layer only: mean squared error {mse(fresh, X, y):.2g}')
fresh.load_state_dict(net.state_dict())
print(f'full state:        mean squared error {mse(fresh, X, y):.2g}')
```

```
linear layer only: mean squared error 0.96
full state:        mean squared error 0.00054
```

Loading only the linear weights causes the MSE to explode by more than two orders of mag-

nitude (no better than predicting zero), whereas restoring the full state including the buffers recovers the trained fit exactly.

This exact same buffer mechanism stores non-trainable tables and running statistics across later chapters, such as the running mean and variance of batch normalization (Section 7.3).

5.4.4 Functions with Custom Gradients

Up to now we have written forward passes and trusted automatic differentiation to supply the backward pass. Autograd differentiates every operation faithfully, which is occasionally the very thing that gets in our way. Return to the 4-bit quantization example from the opening: rounding weights to integers in the forward pass is piecewise constant, so its true derivative is zero almost everywhere—leaving every weight behind a rounding step with zero gradient:

```
w = torch.tensor([0.9, 1.4, 2.6], requires_grad=True)
w.round().sum().backward()
w.grad
```

```
tensor([0., 0., 0.])
```

Mathematically, a zero gradient is exact; practically, it halts learning dead in its tracks. The *straight-through estimator* (STE) (Bengio, Léonard, et al. 2013) sidesteps this by decoupling the two directions: the forward pass still rounds to integers, while the backward pass pretends rounding were the identity function and passes the upstream gradient through unchanged (derivative 1). Although deliberately *not* the true derivative, it works well in practice because the staircase round(x) never deviates by more than $1/2$ from the identity (Figure 5.7). Since autograd cannot invent a surrogate derivative that differs from the forward function, we supply the backward rule explicitly.

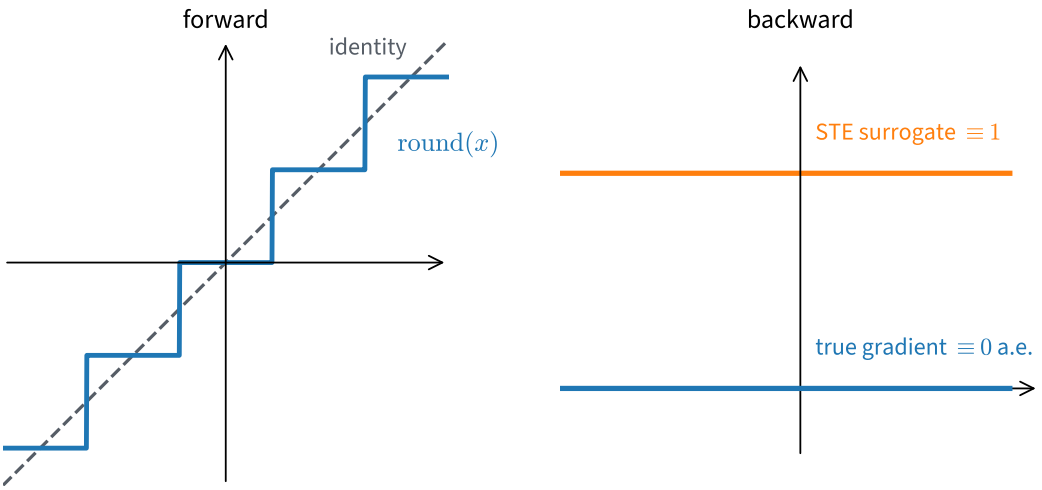


Figure 5.7: The straight-through estimator. Forward: the staircase round(x) stays close to the identity (dashed). Backward: the true derivative of rounding is zero almost everywhere; the estimator uses the derivative of the identity, 1, instead.

In PyTorch the tool is a subclass of `torch.autograd.Function` with the two directions as static methods:

```
class RoundSTE(torch.autograd.Function):
    @staticmethod
    def forward(ctx, X):
        return X.round()

    @staticmethod
    def backward(ctx, grad_output):
        # Treat the forward pass as the identity: pass the gradient on
        return grad_output
```

Always invoke `RoundSTE.apply(X)` (calling `forward` directly bypasses autograd registration); if `backward` needs forward tensors, save them via `ctx.save_for_backward`.

We verify our custom backward rule by multiplying the rounded weights by $(1, 2, 3)$ and summing: the straight-through estimator should deliver $(1, 2, 3)$ directly as the gradient with respect to $\mathbf{w}$:

```
w.grad = None
y = (RoundSTE.apply(w) * torch.tensor([1.0, 2.0, 3.0])).sum()
y.backward()
w.grad
```

```
tensor([1., 2., 3.])
```

The gradient passes straight back to $\mathbf{w}$ as if rounding were the identity, while the forward pass still operates on integer-rounded weights. Can gradient descent actually learn through this surrogate? Below we fit an eight-weight linear model whose weights are rounded to integers during the forward pass on synthetic data generated by weights drawn uniformly from $[-7, 7]$ (the 4-bit signed range). Starting from $\mathbf{w} = \mathbf{0}$, we compare plain rounding against the straight-through estimator, alongside the initial loss and the irreducible rounding floor achieved by rounding the true weights directly:

```
X = torch.randn(256, 8)
w_true = torch.empty(8).uniform_(-7, 7)
y = X @ w_true

def fit(rnd, X, y, num_steps=100, lr=0.05):
    w = torch.zeros(8, requires_grad=True)
    optimizer = torch.optim.SGD([w], lr=lr)
    for _ in range(num_steps):
        loss = ((X @ rnd(w) - y) ** 2).mean()
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    return loss.item()

print(f'start {(y ** 2).mean():.3g}, '
      f'rounding {fit(torch.round, X, y):.3g}, '
```

```
        f'straight-through {fit(RoundSTE.apply, X, y):.3g}, '
        f'rounded target {((X @ w_true.round() - y) ** 2).mean():.3g}')
```

```
start 82.9, rounding 82.9, straight-through 1.04, rounded target 0.557
```

With plain rounding, every gradient is zero and the loss never budges from its starting value. With the straight-through estimator, the loss plunges from tens or hundreds down to 1 or 2, within a small factor of the loss of the rounded ground-truth weights. A deliberately approximate surrogate gradient successfully trains a quantized model.

The identity is just one possible surrogate. A full 4-bit quantizer also clamps its inputs to $[-8, 7]$, so practical implementations often pass gradients only inside the representable range and zero them out in the saturated exterior (see the exercises). Choosing the right surrogate is a modeling decision tailored to the operation it approximates.

When the surrogate *is* the identity, there is an elegant one-line trick that avoids writing a custom autograd function altogether, and it is what you will most often encounter in production code. Using a *stop-gradient* operator (which acts as the identity in the forward pass and has zero derivative in the backward pass), writing $X + \text{stop}(\text{round}(X) - X)$ evaluates to $\text{round}(X)$ forward while differentiating solely through the leading X backward. We still need the full custom-backward mechanism for non-identity surrogates like clipped quantizers, but for plain straight-through rounding, both definitions match identically in values and gradients:

In PyTorch the stop-gradient operation is the method `detach()`, which returns a tensor with the same values that autograd treats as a constant.

```
def round_ste2(X):
    return X + (X.round() - X).detach()

def grad_of(f):
    w.grad = None
    (f(w) * torch.tensor([1.0, 2.0, 3.0])).sum().backward()
    return w.grad

(torch.allclose(RoundSTE.apply(w), round_ste2(w)),
 torch.allclose(grad_of(RoundSTE.apply), grad_of(round_ste2)))
```

```
(True, True)
```

A backward rule written by hand can be wrong, and nothing in the forward pass will show it. The standard test is a *gradient check*: perturb each input by a small amount, measure how much the output changes, and compare this finite-difference slope with the gradient that the rule returns. Below we check a smooth built-in function and our straight-through rule:

```
def check(f):
    x = torch.tensor([0.9, 1.4, 2.6], dtype=torch.float64, requires_grad=True)
    return torch.autograd.gradcheck(f, (x,), raise_exception=False)

check(torch.tanh), check(RoundSTE.apply)
```

```
(True, False)
```

The built-in function passes and the straight-through estimator fails. This failure is by design. The forward pass is a staircase, whose slope is zero wherever a finite difference can measure it, while our backward rule reports a slope of one. A gradient check therefore tells us whether a rule is the derivative of its forward pass, not whether the rule is useful. Run it on every rule that is meant to be exact, such as the clamp in the exercises.

Keep in mind that a custom gradient overrides the derivative of an expression built from existing tensor primitives; writing custom GPU kernels from scratch is a separate systems discipline outside our scope, though Chapter 13 covers how to maximize speed using the primitives frameworks already supply.

One vital question remains about layers like `RMSNorm`: is their floating-point arithmetic actually safe? Squaring activations and summing them can easily overflow in 16-bit formats. In Section 5.5, we examine numerical formats and mixed-precision training directly.

5.4.5 Summary

A custom layer is simply a subclass of the framework's base module implementing a forward pass; registering its tensors handles optimization, containers, checkpointing, and device placement automatically. Learned weights are registered as parameters (as with the gain of `RMSNorm`), while fixed non-trainable tensors (such as the random projections in `RandomFeatures`) are registered as buffers so checkpoints preserve them without exposing them to the optimizer. When an operation's true derivative is uninformative—such as rounding—a custom backward rule like the straight-through estimator (or its stop-gradient shortcut) lets training proceed. Whenever a built-in library layer matches your specification, prefer it for maintenance and fused-kernel speed.

5.4.6 Exercises

1. code **Scale invariance of RMSNorm.** ● Set $\epsilon = 0$.
    1. Show that $\mathrm{RMSNorm}(c\mathbf{x}) = \mathrm{RMSNorm}(\mathbf{x})$ for every $c > 0$.
    2. Differentiate both sides with respect to c at $c = 1$ and conclude that the Jacobian $\mathbf{J}$ of RMSNorm at $\mathbf{x}$ satisfies $\mathbf{J}\mathbf{x} = \mathbf{0}$.
    3. Conclude that the gradient of any loss with respect to the input $\mathbf{x}$ of RMSNorm is orthogonal to $\mathbf{x}$, and check this with automatic differentiation for a random $\mathbf{x}$ and a random linear loss. What does a small gradient step do to the length of $\mathbf{x}$?
2. code **RMSNorm bias.** Add an optional learned bias $\mathbf{b}$ to `RMSNorm` (a shift applied after the gain, restoring part of what layer normalization had), and verify that the state dict of a model containing it grows by the expected entry. Then place the layer directly before a linear layer with weight $\mathbf{W}$ and bias $\mathbf{c}$, and show that $\mathbf{b}$ can be folded into $\mathbf{c}$ without changing the outputs. What is the folded bias?
3. code **Random features approximate a kernel.** Let $\mathbf{w}$ have independent standard normal entries and b be uniform on $[0, 2\pi]$.
    1. Using $\cos\alpha\cos\beta = \frac{1}{2}(\cos(\alpha-\beta) + \cos(\alpha+\beta))$ and $\mathbb{E}[\cos(\mathbf{w}^\top\mathbf{u})] = \exp(-\|\mathbf{u}\|^2/2)$, show that the features $\mathbf{z}(\mathbf{x})$ of `RandomFeatures` satisfy $\mathbb{E}[\mathbf{z}(\mathbf{x})^\top\mathbf{z}(\mathbf{x}')] = \exp(-\|\mathbf{x}-\mathbf{x}'\|^2/2)$ for every number D of features.
    2. Check this numerically for two inputs at distance 1, with $D = 10$, 100, 1,000, and 10,000, averaging the error over several layers for each D. How does the error shrink as D grows?

3. Repeat the fit of $\sin(3x)$ from this section with $D = 5$ and $D = 50$ features, and with a linear model on x alone in place of the random features. What error does each reach, and why?

Adapted from (Rahimi and Recht 2007).

4. code **A clamp with a custom gradient.** Implement an `autograd.Function` whose forward is `X.clamp(lo, hi)` and whose backward passes the gradient only where the input lay strictly inside the clamp range. Compare its gradients with those of the native `torch.clamp` for inputs inside, outside, and exactly on the boundary. Which convention does the native operation use at the boundary?

Adapted from the PyTorch tutorial Learning PyTorch with Examples and the JAX notebook Custom derivative rules.

5.5 Numerical Formats

Every tensor in our models so far has used 32-bit floating point (fp32), without questioning whether neural networks actually need that many bits. Cutting the bit width in half pays off immediately in both memory and throughput: the 8-billion-parameter model running through this chapter takes 32 GB for its weights in 32-bit precision but only 16 GB in 16-bit, and modern GPUs execute 16-bit matrix multiplications at least twice as fast as 32-bit ones. Can we train in 16 bits and still recover the same model?

Not naively. Sixteen-bit formats are noticeably coarser, opening the door to two silent failure modes that raise no runtime error. In fp16, 300^2 already overflows to `inf`, flooding training with `inf` and NaN values. In bf16, adding 0.001 to a weight of 1 rounds right back to 1, silently stalling learning even while gradients look healthy.

This section shows how to train reliably in 16 bits while avoiding both traps. Every tensor carries a *dtype* (its numerical format), and every format trades off *dynamic range* (how large or small representable numbers can be) against *precision* (how finely adjacent numbers are spaced). *Mixed-precision training* keeps an fp32 master copy of the weights so tiny gradient updates accumulate faithfully, while executing expensive matrix multiplications and storing activations in 16 bits, paired with *loss scaling* when using fp16 so small gradients do not underflow to zero. Whereas Section 5.2 counted bytes per parameter, here we control bytes per number. The second subsection (dtype promotion and model casting) is practical reference material you can skim quickly. All examples run on CPU or GPU; actual 16-bit speedups require GPU matrix units, which we measure in Section 5.7.

```
import random
```

5.5.1 Floating-Point Formats

5.5.1.1 Range and Precision

Simply squaring 300 in both 16-bit formats reveals the two distinct ways low precision can bite (300 is hardly exotic inside a neural network, where an unnormalized logit or intermediate variance sum can easily reach that size):

```
x = torch.tensor(300.0)
x.to(torch.float16)**2, x.to(torch.bfloat16)**2
```

```
(tensor(inf, dtype=torch.float16), tensor(90112., dtype=torch.bfloat16))
```

In fp16 (*half precision*), 300^2 overflows straight to `inf` because its largest finite value is just 65504. In bf16 (*brain floating point*), the result stays finite (90112 instead of 90000), yet already incurs rounding error in the third significant digit. Both formats fit in 16 bits; they simply partition those bits differently. Any normal floating-point number stores a sign bit s, an exponent e, and a t-bit mantissa encoding a fraction $f \in [0, 1)$, representing

$$x = (-1)^s \cdot 2^e \cdot (1 + f). \tag{5.8}$$

Exponent bits govern dynamic range (the largest and smallest magnitudes $|x|$ the format can represent), while mantissa bits govern relative precision: adjacent numbers between 1 and 2 sit 2^{-t} apart, and between 2^e and 2^{e+1} they sit 2^{e-t} apart, maintaining uniform relative spacing across scales. The three formats we work with allocate their bits as follows:

format	sign	exponent	mantissa
fp32	1	8	23
bf16	1	8	7
fp16	1	5	10

Notice the trade-off immediately. bf16 preserves all 8 exponent bits of fp32 while trimming the mantissa down to 7 bits, delivering fp32's full dynamic range at coarser resolution. fp16 does the opposite, retaining 10 mantissa bits but only 5 exponent bits: finer local steps, far narrower range. We can check both outputs above by hand: with 5 exponent bits, e spans $[-14, 15]$, capping fp16 at $2^{15}(2 - 2^{-10}) = 65504$. With 7 mantissa bits, bf16 values between 2^{16} and 2^{17} are spaced by $2^9 = 512$, so 90000 falls between $175 \cdot 512 = 89600$ and $176 \cdot 512 = 90112$ and rounds to the closer value, 90112.

Figure 5.8 visualizes these bit budgets side by side (see Section 35.8 for a deeper treatment of IEEE floating-point arithmetic).

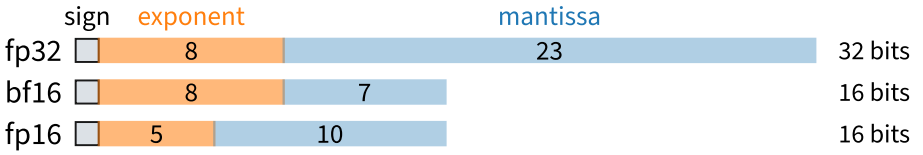


Figure 5.8: Bit layouts of the formats in the table, aligned at the sign bit, so that a longer bar means more bits: bf16 keeps the 8-bit exponent of fp32, and fp16 and bf16 split the same 16 bits in opposite ways.

We can query each format's exact numerical limits via `finfo`, focusing on three fields: `max` (largest finite value), `tiny` (smallest positive *normal* value), and `eps` (the gap 2^{-t} just above 1). Below `tiny`, where the exponent hits its floor, the format degrades gracefully through evenly spaced *subnormal* numbers down to zero: in fp16 these are multiples of 2^{-24}, so a

value near 3×10^{-5} still preserves 9 significant bits, whereas one near 10^{-7} retains only a single bit.

```
for dtype in (torch.float32, torch.bfloat16, torch.float16):
    fi = torch.finfo(dtype)
    print(f'{str(dtype):15} max {fi.max:10.3e}  tiny {fi.tiny:9.3e}'
          f'  eps {fi.eps:8.3e}')
```

```
torch.float32   max  3.403e+38  tiny 1.175e-38  eps 1.192e-07
torch.bfloat16  max  3.390e+38  tiny 1.175e-38  eps 7.812e-03
torch.float16   max  6.550e+04  tiny 6.104e-05  eps 9.766e-04
```

Because bf16 shares fp32's 8-bit exponent, it matches fp32's `tiny` and nearly matches its `max`, while its `eps` of 0.0078 gives two to three decimal digits of precision. fp16 resolves three to four decimal digits, but overflows above 65504, slips into subnormals below 6.1×10^{-5}, and cannot represent any positive value below $2^{-24} \approx 6.0 \times 10^{-8}$. Which shortage hurts deep learning more? Activations and gradients routinely span many orders of magnitude; running out of range catastrophically turns numbers into `inf` or zero, whereas slightly coarser precision merely introduces mild rounding noise. That is why bf16 has become the default 16-bit training format on modern hardware, leaving fp16 primarily for older GPUs and inference.

5.5.1.2 Rounding of Small Updates

Where bf16's coarse spacing *does* hurt is during parameter updates. A single gradient descent step shifts a weight by ηg, often 10^{-3} or less of the weight's magnitude. Under round-to-nearest, any step smaller than half the local spacing vanishes completely. Below we add 10^{-3} and 10^{-4} to a weight of 1 in each format:

```
w = torch.tensor(1.0)
for dtype in (torch.float32, torch.bfloat16, torch.float16):
    print(f'{str(dtype):14}: {float(w.to(dtype) + 1e-3):.7f}, '
          f'{float(w.to(dtype) + 1e-4):.7f}')
```

```
torch.float32 : 1.0010000, 1.0001000
torch.bfloat16: 1.0000000, 1.0000000
torch.float16 : 1.0009766, 1.0000000
```

In fp32, both increments register cleanly. In bf16, whose spacing near 1 is $2^{-7} \approx 0.0078$, both round right back to 1, leaving the weight completely frozen. In fp16 (spacing $2^{-10} \approx 0.00098$), the 10^{-3} step survives (rounded to one grid step) while 10^{-4} disappears. Figure 5.9 draws both grids to one scale: an update registers only if it carries the weight more than half a grid spacing away. Storing weights directly in 16 bits can therefore stall optimization even when every gradient is accurate, which is why training keeps master weights in fp32.

5.5.2 Dtype Promotion and Model Casting

5.5.2.1 Promotion

What dtype results when an expression combines mismatched operands? We test adding an fp16 tensor to an fp32 tensor, a Python float, and an integer tensor:

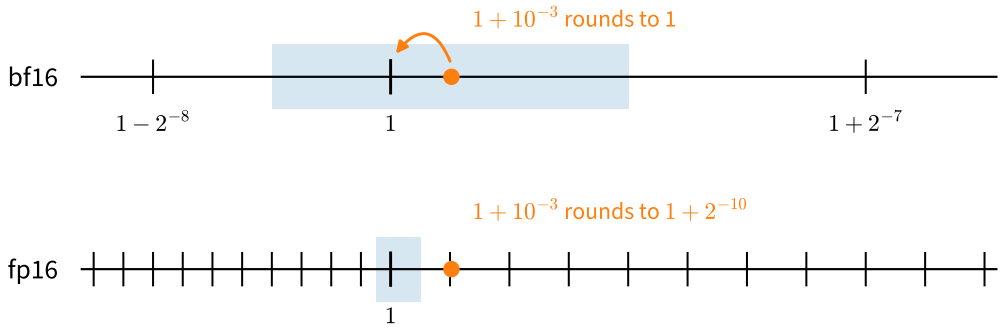


Figure 5.9: The bf16 grid (top) and the fp16 grid (bottom) near a weight of 1, drawn to one scale. A shaded interval holds the values that round back to 1; it reaches half a grid spacing to each side, and the spacing below 1 is half the spacing above it. In bf16 the update $1+10^{-3}$ lies inside the interval and is lost. In fp16 it lies outside and rounds to the next grid point, $1 + 2^{-10}$.

```
x16 = torch.ones(3, dtype=torch.float16)
x32 = torch.ones(3, dtype=torch.float32)
(x16 + x32).dtype, (x16 + 1.0).dtype, (x16 + torch.arange(3)).dtype
```

```
(torch.float32, torch.float16, torch.float16)
```

Combining two float tensors promotes to the wider format (and fp16 + bf16 promotes to fp32), so a single stray fp32 tensor silently promotes an entire downstream 16-bit computation to fp32. Python float literals and integer tensors do not promote float tensors.

5.5.2.2 Casting a Model

Since parameters are just tensors, we can cast an entire model to 16 bits. Throughout this section we use our Fashion-MNIST MLP from Section 4.2: 784 inputs, a 256-unit ReLU hidden layer, and 10 outputs, with $\mathcal{N}(0, 0.01^2)$ weights and zero biases (203,530 parameters occupying 814,120 bytes in fp32).

```
def mlp():
    net = nn.Sequential(nn.Flatten(), nn.Linear(784, 256), nn.ReLU(),
                        nn.Linear(256, 10))
    for layer in net[1], net[3]:
        nn.init.normal_(layer.weight, std=0.01)
        nn.init.zeros_(layer.bias)
    return net

def param_bytes(net):
    return sum(p.numel() * p.element_size() for p in net.parameters())

net = mlp()
param_bytes(net)
```

```
814120
```

`net.to(dtype)` (or `net.bfloat16()`) casts all parameters and buffers; layers reject inputs whose dtype does not match their weights.

```
net = net.bfloat16()
X = torch.randn(8, 1, 28, 28)
try:
    net(X)
except RuntimeError as e:
    print(e)
print(net(X.bfloat16()).dtype, param_bytes(net))
```

```
mat1 and mat2 must have the same dtype, but got Float and BFloat16
torch.bfloat16 407060
```

Casting halves parameter memory immediately, though frameworks differ in whether a 16-bit model casts an fp32 input automatically or rejects the dtype mismatch.

For *inference*, casting a trained model to 16 bits immediately halves its weight footprint (subject to verifying that task accuracy holds up). For *training*, however, pure 16-bit weights run straight into the small-update rounding trap we just saw, as we confirm empirically below.

5.5.3 Mixed-Precision Training

5.5.3.1 Master Weights and 16-Bit Compute

We appear to face a dilemma: reliable weight updates demand 32-bit storage, yet hardware speedups live in 16-bit arithmetic. *Mixed-precision training* (Micikevicius et al. 2018) resolves this tension by decoupling parameter storage from computation. We keep fp32 *master weights* so every small update accumulates cleanly, cast them to 16 bits for the compute-heavy matrix multiplications and activations of the forward and backward passes, and keep sensitive reductions such as the loss in fp32. Figure 5.10 summarizes a single training step.

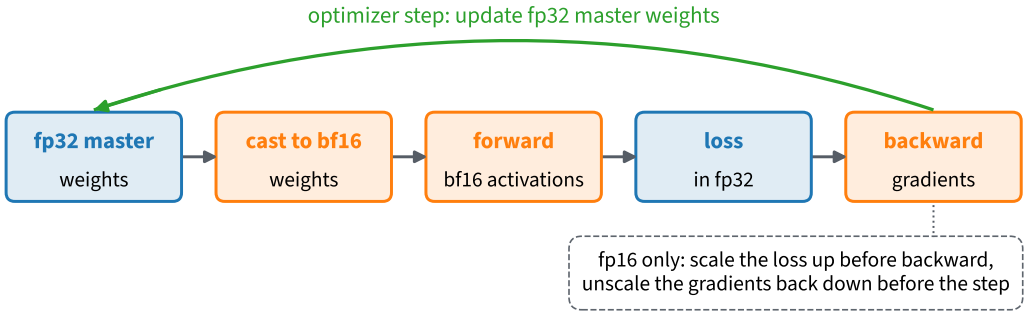


Figure 5.10: One step of mixed-precision training: the fp32 master weights are cast to bf16 for the forward pass, which produces bf16 activations; the loss is computed in fp32; the backward pass produces gradients, and the optimizer applies them to the fp32 master weights. With fp16, the loss is also scaled up before the backward pass and the gradients are scaled back down before the update.

`torch.autocast` leaves parameters in fp32 and automatically casts eligible ops (matrix multiplies, convolutions) to 16 bits while keeping reductions and losses in fp32.

```
net = mlp()
with torch.autocast('cpu', dtype=torch.bfloat16):
    Y = net(X)
Y.dtype, net[1].weight.dtype
```

```
(torch.bfloat16, torch.float32)
```

5.5.3.2 Training in fp32 and in 16 Bits

How much does 16-bit arithmetic affect optimization, and what goes wrong when we drop the fp32 master copy? Below we train our MLP three times with plain SGD on Fashion-MNIST minibatches of 256 images: full fp32, pure bf16 weights (with no fp32 master copy), and mixed precision. All three runs start from identical random weights and consume the exact same minibatches in the same order.

Notice the learning rate: we set $\eta = 0.01$ (one-tenth the rate used in Section 4.2) and run for 1000 steps (about four epochs). Our rounding analysis explains why η matters so dramatically: an update ηg disappears whenever it falls below half the local bf16 grid spacing, so a smaller learning rate loses more updates. A small rate is no artificial setting. Every real training run reaches one once its learning rate has decayed. Just before training, we will measure how much is lost at each of the two rates.

```
data = d2l.FashionMNIST(batch_size=256)
batches = list(torch.utils.data.DataLoader(data.train, batch_size=256,
                                           drop_last=True))  # One epoch
val_batch = next(iter(torch.utils.data.DataLoader(data.val, batch_size=1024)))
```

With the training minibatches and a 1,024-image validation slice preloaded in memory, `train` runs $1{,}000$ SGD steps at $\eta = 0.01$ and returns the per-step losses alongside final validation accuracy:

```
def train(net, mixed=False, num_steps=1000):
    dtype = next(net.parameters()).dtype  # The inputs follow the weights
    optimizer = torch.optim.SGD(net.parameters(), lr=0.01)
    losses = []
    for step in range(num_steps):
        X, y = batches[step % len(batches)]
        with torch.autocast('cpu', dtype=torch.bfloat16, enabled=mixed):
            loss = F.cross_entropy(net(X.to(dtype)).float(), y)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        losses.append(loss.detach())
    Xv, yv = val_batch
    with torch.no_grad():
        accuracy = (net(Xv.to(dtype)).argmax(1) == yv).float().mean().item()
    return torch.stack(losses).tolist(), accuracy
```

One implementation detail in `train` is a good systems habit independent of precision: we keep per-step loss tensors on-device during the loop and convert them to Python floats once at the end, avoiding a CPU–GPU synchronization stall on every single step (Section 5.7).

Before training, we check the rounding argument on the first minibatch. Of the nonzero first-layer updates ηg at initialization, what fraction would bf16 storage round away at $\eta = 0.01$ and at $\eta = 0.1$, and what share of the total update magnitude do the lost updates carry?

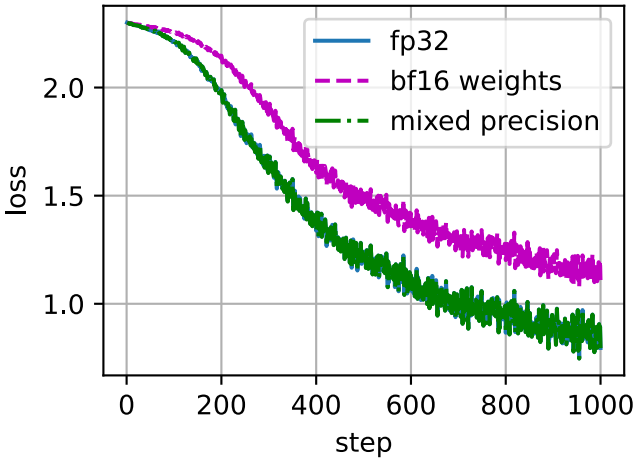
```
net = mlp()
X, y = batches[0]
F.cross_entropy(net(X), y).backward()
w, g = net[1].weight.detach(), net[1].weight.grad
w16, moved = w.bfloat16(), g != 0  # Count only the nonzero updates
for lr in (0.01, 0.1):
    lost = ((w16.float() - lr * g).bfloat16() == w16) & moved
    share = lost.sum() / moved.sum()
    mass = g[lost].abs().sum() / g.abs().sum()
    print(f'lr {lr}: {share:.0%} of updates rounded away, '
          f'{mass:.0%} of the update mass')
```

```
lr 0.01: 92% of updates rounded away, 80% of the update mass
lr 0.1: 52% of updates rounded away, 16% of the update mass
```

At $\eta = 0.01$ about nine in ten updates are rounded away, and they carry about four-fifths of the total update magnitude (the "mass" in the printout): most of the gradient step never reaches the weights. At $\eta = 0.1$ about half of the updates are still lost, but they are the small ones and carry only about 15% of the magnitude. Most of the step then lands, and the failure stays hidden until the learning rate decays. Now we train all three variants at $\eta = 0.01$.

```
def seeded_mlp(seed):
    torch.manual_seed(seed)  # The same seed gives the same initial weights
    return mlp()

seed = random.randrange(2**31)  # One random draw, shared by all runs
loss32, acc32 = train(seeded_mlp(seed))
loss_bf16, acc_bf16 = train(seeded_mlp(seed).bfloat16())
loss_mixed, acc_mixed = train(seeded_mlp(seed), mixed=True)
print(f'validation accuracy: fp32 {acc32:.3f}, bf16 weights {acc_bf16:.3f}, '
      f'mixed {acc_mixed:.3f}')
d2l.plot(list(range(1, 1001)), [loss32, loss_bf16, loss_mixed], 'step',
         'loss', legend=['fp32', 'bf16 weights', 'mixed precision'])
```

```
validation accuracy: fp32 0.684, bf16 weights 0.639, mixed 0.685
```



The fp32 and mixed-precision loss curves sit virtually on top of each other, with validation accuracies matching within about one percentage point. Pure bf16 weights, by contrast, lag behind within the first hundred steps: after 1000 steps, its loss has only reached about 1.15 versus 0.85, and validation accuracy trails at about 64% versus 68%. Just as our scalar experiment predicted, storing weights directly in bf16 rounds away most first-layer updates, whereas mixed precision applies them cleanly to fp32 master weights. None of the three runs is trained to convergence; what matters is the apples-to-apples comparison on identical data and step counts.

Notice two important caveats about what mixed precision does and does not buy us. First, we do not see a wall-clock speedup on a tiny CPU MLP; 16-bit speedups come from dedicated GPU tensor cores working on large matrix multiplications, which we benchmark in Section 5.7.2.1. Second, mixed precision does *not* shrink the static per-parameter memory floor from Section 5.2, because master weights and optimizer moments remain in fp32. That 16 GB footprint from the opening applies to an 8B model cast for *inference*; during *training*, mixed precision saves activation memory (halving stored 16-bit activations) and matrix-multiplication time.

5.5.3.3 Loss Scaling for fp16

With bf16, master weights complete the recipe. With fp16, however, its narrow exponent range creates a second hazard at the bottom end: small gradients *underflow*. Magnitudes below $2^{-14} \approx 6.1 \times 10^{-5}$ drop into subnormals and shed significant bits, and anything below half the smallest subnormal ($\approx 3 \times 10^{-8}$) rounds straight to zero. A gradient of 10^{-8} vanishes in fp16, yet multiplying it by 2^{16} lifts it safely into normal range:

```
g = torch.tensor(1e-8)
g.half(), (g * 2**16).half()
```

```
(tensor(0., dtype=torch.float16), tensor(0.0007, dtype=torch.float16))
```

How common are tiny gradients in practice? Below we compute first-layer fp32 gradients for our MLP on the first minibatch and measure what fraction of nonzero entries fall into fp16's subnormal or zero regimes, both unscaled and scaled by 2^{16}:

```
net = mlp()
X, y = batches[0]
F.cross_entropy(net(X), y).backward()
g = net[1].weight.grad.flatten()
g = g[g != 0]
for scale in (1, 2**16):
    print(f'scale {scale:5}: '
          f'{((g * scale).abs() < 2**-14).float().mean():.1%} subnormal, '
          f'{((g * scale).half() == 0).float().mean():.2%} zero')
```

```
scale     1: 32.5% subnormal, 0.21% zero
scale 65536: 0.0% subnormal, 0.00% zero
```

Roughly a third of the nonzero first-layer gradients land in fp16's subnormal range, and about two in a thousand underflow all the way to zero. On this small MLP the impact is mild because only the tiniest entries vanish; in fact, it is hard to construct a toy model where fp16 fails outright and loss scaling saves it. Severe underflow hits large, deep models where losses average over many tokens or pixels and gradients attenuate across dozens of layers: Micikevicius et al. (2018) found that while several ImageNet classifiers trained fine in unscaled fp16, an object detector diverged and a large recurrent language model matched fp32 accuracy only once loss scaling was enabled.

Multiplying the gradients by 2^{16} eliminates every subnormal entry while staying comfortably below fp16's overflow ceiling of 65504. That is the essence of *loss scaling*: before the backward pass, multiply the scalar loss by a large constant S. By linearity of differentiation, every intermediate gradient shifts up by S during backpropagation. Before stepping the optimizer, divide the gradients back down by S in fp32, where tiny values are represented faithfully, leaving the mathematical parameter update completely unchanged while keeping intermediate fp16 gradients out of the underflow zone.

Choosing S statically is brittle: too small and gradients underflow, too large and big gradients overflow to `inf`. *Dynamic loss scaling* automates the choice: start with a large S, inspect the gradients on every step, and whenever an `inf` or NaN appears, skip the optimizer update and halve S; after a streak of clean steps, double S to stay close to the top of fp16's range. Figure 5.11 shows one step of this loop.

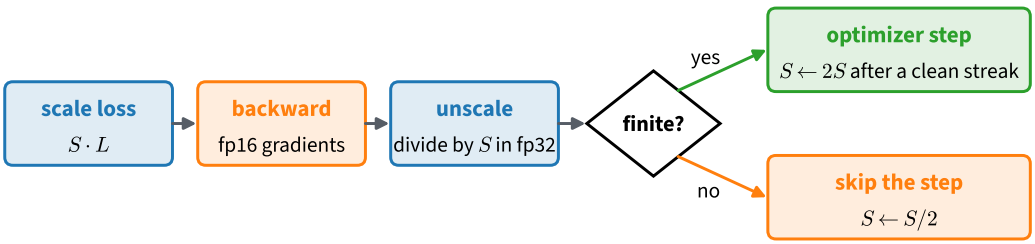


Figure 5.11: One step of dynamic loss scaling. The loss is multiplied by S, the backward pass runs in fp16, and the gradients are divided by S in fp32. If all of them are finite, the optimizer takes its step, and S doubles after a streak of such steps. If any is `inf` or NaN, the step is skipped and S is halved.

`torch.amp.GradScaler` automates dynamic loss scaling with fp16 `autocast`:

```
net = mlp()
optimizer = torch.optim.SGD(net.parameters(), lr=0.1)
scaler = torch.amp.GradScaler('cpu')
for X, y in batches[:20]:
    with torch.autocast('cpu', dtype=torch.float16):
        loss = F.cross_entropy(net(X), y)
    optimizer.zero_grad()
    scaler.scale(loss).backward()  # Gradients of S * loss
    scaler.step(optimizer)         # Divide by S, skip the step on inf/NaN
    scaler.update()                # Adjust S
print(f'loss {loss.item():.3f}, loss scale {scaler.get_scale():.0f}')
```

```
loss 2.030, loss scale 65536
```

Consequently, on modern hardware with native bf16 support (recent GPUs, TPUs, and server CPUs), bf16 mixed precision without loss scaling is the standard recipe; fp16 with dynamic loss scaling remains essential on older GPUs.

One final hardware feature accelerates fp32 models without changing any tensor dtypes at all. Starting with NVIDIA's Ampere architecture, fp32 matrix multiplications can run on tensor cores in *tf32* mode: inputs are truncated to fp16's 10-bit mantissa while keeping fp32's 8-bit exponent, and inner products accumulate in full fp32, delivering substantial speedups while matching strict fp32 to three or four significant digits.

PyTorch defaults to strict fp32 matmuls; `torch.set_float32_matmul_precision('high')` (Section 5.7) enables tf32.

5.5.4 Numerical Failure Modes

Three concrete hazards explain the vast majority of numerical bugs in 16-bit code: an upstream overflow that surfaces downstream as NaN, an unshifted exponential that blows up before the final logarithm can shrink it, and a long reduction that stops growing once individual terms fall below the accumulator's rounding threshold.

5.5.4.1 From Overflow to NaN

A NaN loss is rarely the root cause: it is usually the aftermath of an earlier overflow to `inf`, because indeterminate arithmetic like `inf - inf` or `inf / inf` produces NaN:

```
s = torch.tensor(60000., dtype=torch.float16) * 2  # Overflows
s, s - s
```

```
(tensor(inf, dtype=torch.float16), tensor(nan, dtype=torch.float16))
```

That initial overflow may occur layers upstream before NaN ever reaches the loss, and once a single parameter update carries a NaN, every subsequent step is poisoned. Whenever a loss turns NaN, check value ranges first (is any operation running in fp16 near its 65504 ceiling?) before blaming the learning rate or architecture.

5.5.4.2 Log-Sum-Exp

Softmax cross-entropy requires computing $\log \sum_i \exp(x_i)$. Evaluating that expression literally overflows in fp16 even when the final answer is small:

```
x = torch.tensor([12.0, 11.0, 10.0], dtype=torch.float16)
x.exp().sum().log(), torch.logsumexp(x, dim=0)
```

```
(tensor(inf, dtype=torch.float16), tensor(12.4062, dtype=torch.float16))
```

Although the true answer is a modest 12.4, the intermediate term $e^{12} \approx 163{,}000$ easily blows past fp16's 65504 limit. The fix is the exact shift we introduced in Section 3.3.5.2: subtract the maximum entry $m = \max_i x_i$ before exponentiating and add m back afterward,

$$\log \sum_i \exp(x_i) = m + \log \sum_i \exp(x_i - m). \tag{5.9}$$

Every exponent on the right is nonpositive, making overflow impossible, and the largest term is exactly $e^0 = 1$, preventing underflow to $\log 0$. In practice you rarely need to code this shift yourself: built-in log-softmax and cross-entropy routines apply it automatically.

5.5.4.3 Accumulation in fp32

Long sums accumulated in 16 bits stall silently: once the running total grows large enough, each small summand falls below half the local grid spacing and rounds to zero. Below we add 2^{-10} (represented *exactly* in fp16) 4096 times into an fp16 accumulator versus an fp32 accumulator:

```
total16, total32 = np.float16(0), np.float32(0)
term = np.float16(2**-10)
for _ in range(4096):
    total16 += term  # fp16 accumulator
    total32 += term  # fp32 accumulator, same fp16 terms
total16, total32
```

```
(np.float16(2.0), np.float32(4.0))
```

The exact total is 4, which the fp32 accumulator hits on the nose. The fp16 accumulator freezes at 2 after 2048 steps: between 2 and 4, adjacent fp16 values lie 2^{-9} apart, so $2 + 2^{-10}$ lands right on the midpoint and round-to-nearest-even snaps it back to 2 forever. Batch averages, epoch loss totals, and variance calculations in normalization layers all share this hazard, which is why mixed-precision policies keep reductions and normalizations in fp32, and why any custom reduction over 16-bit tensors should accumulate in fp32.

All three failure modes trace back to the same root: an intermediate quantity outstripping the range or precision of a 16-bit format even when the final answer fits comfortably. Remember also that mixed precision and tf32 alter the lowest bits of arithmetic by design, so they will never match pure fp32 bit for bit. When runs *can* be made bit-for-bit repeatable, and how to achieve it, is the focus of Section 5.8.

5.5.5 Summary

Floating-point formats trade dynamic range (exponent bits) against precision (mantissa bits): fp16 gives finer local steps but a narrow range (overflowing past 65504 and underflowing tiny gradients), whereas bf16 keeps fp32's full 8-bit exponent at just two to three decimal digits of precision. Storing weights directly in 16 bits rounds away small gradient updates and stalls training, so mixed-precision training keeps fp32 master weights while running matrix multiplications and storing activations in 16 bits, matching fp32 accuracy on our Fashion-MNIST MLP where pure bf16 weights fall behind. With fp16, dynamic loss scaling lifts gradients out of the subnormal/underflow zone during backpropagation; bf16 avoids this step altogether. Finally, guard low-precision code against overflow-induced NaNs by using shifted log-sum-exp operations and fp32 accumulators for long reductions and normalizations.

5.5.6 Exercises

1. **Mixed-precision memory.** Training with Adam in fp32 stores 16 bytes per parameter (Section 5.2). Suppose that `train` used Adam instead of SGD.
    1. In which dtype does the mixed-precision recipe of your framework store the weights, the gradients, and Adam's running averages between steps? Does mixed precision reduce the bytes per parameter?
    2. Count as stored activations the input of each linear layer and the logits. For the MLP of `train`, convert the per-example count to bytes in fp32 and in the 16-bit format `train` uses. At which batch size does the activation saving equal the persistent training state of this MLP?
2. **Loss scaling on overflow.** Under dynamic loss scaling (Section 5.5.3.3), a step whose unscaled gradients contain an `inf` or NaN is skipped and the scale is halved. After N consecutive clean steps, the scale is doubled.
    1. A skipped step must leave Adam's running averages untouched. Explain what would go wrong if the step instead updated them with the non-finite gradient, and what would go wrong if it updated them with the non-finite entries replaced by zeros.
    2. Explain why the scale must grow again after a run of clean steps instead of only shrinking.
    3. Suppose that every step overflows when the scale is above a fixed threshold and that no step overflows below it. Once the scale has settled, what fraction of steps is skipped as a function of N? Without the assumption of a fixed threshold, what does a larger N cost?
3. code **Normalization in fp16.** Mixed-precision recipes run normalization layers in fp32. To see why, consider RMSNorm with its gain set to one (Section 5.4 implements it as a module). It rescales an input row $\mathbf{x}$ of width d by its root mean square,

$$\frac{\mathbf{x}}{\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2 + \epsilon}}, \tag{5.10}$$

    where the small constant ϵ guards against division by zero. Compute it entirely in fp16 for a batch of 64 rows of width $d = 1024$ whose entries have a standard deviation of about 100.

1. Which intermediate quantity is the first to become `inf`, and what does the layer return for the affected rows?
2. Take the squares as exact. At what input standard deviation does their mean exceed the fp16 maximum, and at what standard deviation does their sum over the d entries exceed it?

4. code **The e4m3fn format.** ● Print `torch.finfo(torch.float8_e4m3fn)` and compare its `max`, `tiny` and `eps` with those of fp16 and bf16. Explain the name `e4m3fn`. A format with a 4-bit exponent and a 3-bit mantissa that followed the IEEE convention would reserve the all-ones exponent for `inf` and NaN. Compute that format's largest finite value, and explain how `e4m3fn` reaches a `max` of 448 instead.

5.6 Saving, Loading, and Pretrained Weights

A trained neural network is surprisingly fragile: it exists only in the memory of the process that trained it. The moment that process terminates (whether because your script finished, hit an error, tripped over a server's power cord, or ran on a preemptible cloud instance), every learned weight vanishes unless saved to disk. For the toy models of earlier chapters, losing progress costs minutes. At modern scale, it can erase weeks of compute, and hardware failures are not rare exceptions: they are routine. Over a 54-day window of pre-training Llama 3 405B on up to 16,384 GPUs, Meta recorded 419 unexpected interruptions, roughly one every three hours (Grattafiori et al. 2024). Without reliable checkpointing and resumption, a run of that scale could never complete.

Saving weights serves a second, equally ubiquitous purpose: sharing trained models so others can run inference or fine-tune them without training from scratch (Section 32.2). Both workflows raise the same practical questions: what exactly belongs on disk, which file formats are safe to load, and how should we verify that weights match our architecture?

The answer to *what* to save follows directly from Section 5.2. A model consists of code plus state: the code defines the architecture and forward pass, while the *state* is the collection of named tensors (parameters and buffers alike) produced by training. We serialize only the state, mapping names to tensors, and reconstruct the model by instantiating the architecture in code and loading those tensors back in. Resuming an interrupted training run demands still more: besides model weights, we must save optimizer state (such as Adam's running moments), random-number generator states, the step counter, and the architecture configuration, making a training checkpoint several times larger than an inference weight file.

In this hands-on section, we first inspect and serialize model state, then contrast unsafe pickle files with the safe `safetensors` format, build an atomic training checkpoint capable of bit-for-bit resumption, and finish by loading pretrained ResNet-50 weights into a modified architecture while verifying missing and unexpected keys.

```
import json
import os
import struct
from collections import Counter
from dataclasses import asdict, dataclass
import numpy as np
import torch
```

```
from torch import nn
from torch.nn import functional as F
from torchvision.models import resnet50, ResNet50_Weights
from safetensors.torch import load_file, save_file
```

5.6.1 Model State

Because a model contains dozens or hundreds of tensors, saving it requires storing a mapping from names to tensors. Each framework generates this mapping automatically by traversing the module tree (Section 5.1), keying every tensor by its path from the root. Below we inspect the state of a simple single-hidden-layer MLP with 256 hidden units.

`collect_params()` and `state_dict()` map dotted paths (`hidden.weight`, `output.bias`) to parameters and buffers:

```
class MLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.hidden = nn.Linear(20, 256)
        self.output = nn.Linear(256, 10)

    def forward(self, x):
        return self.output(F.relu(self.hidden(x)))

net = MLP()
X = torch.randn(2, 20)
Y = net(X)
{name: tuple(t.shape) for name, t in net.state_dict().items()}
```

```
{'hidden.weight': (256, 20),
 'hidden.bias': (256,),
 'output.weight': (10, 256),
 'output.bias': (10,)}
```

Notice that this state dictionary records names, shapes, and numerical values, and nothing else. It contains no trace of the `MLP` class, its forward logic, or its activation function. Restoring the model therefore requires Python code that recreates the exact matching names and shapes: even renaming an attribute from `output` to `head` breaks the mapping despite computing the identical function. We show how to catch and diagnose such mismatches later in this section.

5.6.2 Pickle and safetensors

When checkpoint files never leave your own disk, almost any serialization format suffices. The moment you download pretrained weights from a public hub or accept a checkpoint from someone else, however, a security question arises: can loading that file execute arbitrary code on your machine? Unfortunately, with Python's default serialization format, the answer is yes.

5.6.2.1 Code Execution on Load

Python's built-in `pickle` module does not store passive data: it stores a bytecode program that reconstructs Python objects, and unpickling *executes* that program, allowing it to call any importable function. Any class can customize what pickle executes via `__reduce__`, returning a callable and its arguments. Our `Tripwire` example below benignly invokes `print`; a malicious checkpoint could just as easily invoke `os.system`.

```
class Tripwire:
    def __reduce__(self):  # pickle calls print(...) to rebuild the object
        return (print, ('*** payload executed while loading ***',))
```

`torch.save` writes a pickle archive; loading with `weights_only=False` (the default prior to PyTorch 2.6) executes the payload immediately:

```
torch.save(Tripwire(), 'tripwire.pt')
_ = torch.load('tripwire.pt', weights_only=False)  # the payload runs here
```

```
*** payload executed while loading ***
```

By default, the loaders reject arbitrary Python objects:

`torch.load` defaults to `weights_only=True` since PyTorch 2.6.

```
try:
    torch.load('tripwire.pt')  # weights_only=True, the default
except Exception as e:
    print(type(e).__name__, str(e).split('.')[0])
```

```
UnpicklingError Weights only load failed
```

Restricted unpicklers only protect you insofar as their allowlists have no bugs, and historical vulnerabilities have bypassed even hardened restricted loaders. A file format that simply *cannot* encode executable instructions eliminates the entire attack surface by design.

PyTorch's `weights_only` mode is such a restricted loader.

5.6.2.2 The safetensors Format

The *safetensors* format stores strictly raw tensor data and metadata—never code. Its binary layout has just three contiguous segments (Figure 5.12): an 8-byte little-endian integer giving the header length, a UTF-8 JSON header mapping each tensor name to its dtype, shape, and byte offsets, and the raw tensor buffers packed back to back. Loading executes zero code; the reader merely validates the JSON header and checks that every byte range stays within the file. As an added systems bonus, knowing exact byte offsets upfront enables memory-mapping (`mmap`) and random access to individual tensors without reading the entire file. Framework-agnostic and fast, safetensors has become the standard weight distribution format on the Hugging Face Hub.

We save our MLP's state to `mlp.safetensors`, load it into a fresh clone, and verify that both instances produce identical outputs.

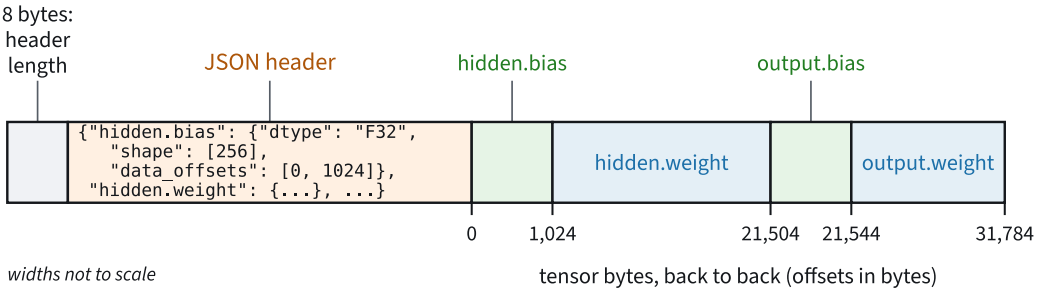


Figure 5.12: The safetensors file of the MLP below as one strip of bytes: an 8-byte length, the JSON header with each tensor's dtype, shape, and byte range, and the raw bytes of the four tensors back to back, in the order of their names. The offsets under the data are the ranges that the header records.

```
save_file(net.state_dict(), 'mlp.safetensors')
clone = MLP()
clone.load_state_dict(load_file('mlp.safetensors'))
torch.equal(clone(X), Y)
```

```
True
```

The round trip is bit-for-bit exact. Even better, because the header is plain JSON at a fixed offset, we can inspect the file's contents using nothing beyond Python's standard library:

```
with open('mlp.safetensors', 'rb') as f:
    n = struct.unpack('<Q', f.read(8))[0]  # header length, little-endian
    header = json.loads(f.read(n))
print('header length:', n, 'bytes')
for name, entry in sorted(header.items(), key=lambda e: e[1]['data_offsets']):
    print(f"{name:18s} {entry['dtype']} {str(entry['shape']):10s} "
          f"bytes {entry['data_offsets']}")
```

```
header length: 296 bytes
hidden.bias        F32 [256]      bytes [0, 1024]
hidden.weight      F32 [256, 20]  bytes [1024, 21504]
output.bias        F32 [10]       bytes [21504, 21544]
output.weight      F32 [10, 256]  bytes [21544, 31784]
```

Every byte interval equals the tensor's element count times 4 bytes (`F32` = float32): the 256-entry hidden bias occupies the first 1,024 bytes, the $256 \times 20 = 5{,}120$-entry hidden weight occupies the next 20,480 bytes, and each block begins immediately where the previous one ends—pure tensor data with zero hidden code.

For private scratch checkpoints on your own machine, native framework formats are perfectly fine. Whenever weights cross a trust boundary—published to users or downloaded from the web—use a code-free format like safetensors paired with a cryptographic checksum so recipients can verify integrity.

5.6.3 Training Checkpoints

A bare weights file suffices for inference, but *not* for resuming training. The reason is stateful optimization. Most modern deep learning runs use *Adam* (Section 9.5), which tracks two per-parameter running averages (Section 5.2): $\mathbf{m}$ for the gradient $\mathbf{g}$ and $\mathbf{v}$ for its elementwise square $\mathbf{g} \odot \mathbf{g}$, updating each parameter opposite its smoothed gradient scaled down where recent gradients have been large:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \frac{\hat{\mathbf{m}}}{\sqrt{\hat{\mathbf{v}}} + \epsilon}. \tag{5.11}$$

Here η is the learning rate, ϵ is a small constant (typically 10^{-8}), division and square root act elementwise, and the hats indicate bias-correction factors compensating for zero initialization (derived in Section 9.5). Crucially, $\mathbf{m}$ and $\mathbf{v}$ encode the optimization history: discard them, and Adam behaves as if training had just started from scratch.

A truly resumable training checkpoint therefore packs five distinct components (Figure 5.13): (1) model state (parameters and buffers), (2) optimizer state ($\mathbf{m}$ and $\mathbf{v}$), (3) random-number generator states so minibatch shuffling and dropout continue seamlessly, (4) the step counter governing Adam's bias correction and any learning-rate schedule, and (5) the model configuration needed to rebuild the architecture. Gradients are omitted because the next step recomputes them immediately. That leaves 12 of the 16 bytes per parameter held during fp32 Adam training (Section 5.2), namely weights plus two moments, so a resumable checkpoint for an 8-billion-parameter model weighs in at 96 GB: three times its 32 GB fp32 weights and six times its 16 GB bf16 inference file (Section 5.5).

Checkpointing must also be *crash-safe*: overwriting an existing checkpoint in place risks losing both old and new versions if the job dies mid-write. Instead, we write to a temporary file and atomically rename it over the destination once finished, guaranteeing that any reader sees either the complete previous checkpoint or the complete new one. The rename protects against a crash of the process. Surviving a power loss also takes an `os.fsync` on the file and on its directory, which the exercises work out.

We pack model state, optimizer state, step, `asdict(cfg)`, and CPU/CUDA/`torch.Generator` RNG states into a pure-tensor/primitive dict compatible with `weights_only=True`, saving via a `.tmp` file and `os.replace`:

```
def save_checkpoint(path, model, optimizer, step, cfg=None, rng=None):
    """Atomically write a resumable training checkpoint."""
    ckpt = {'model': model.state_dict(),
            'optimizer': optimizer.state_dict(),
            'step': step,
            'cpu_rng': torch.get_rng_state()}
    if cfg is not None:
        ckpt['cfg'] = asdict(cfg)
    if rng is not None:                          # a torch.Generator
        ckpt['rng'] = rng.get_state()
    if torch.cuda.is_available():
        ckpt['cuda_rng'] = torch.cuda.get_rng_state_all()
    tmp = path + '.tmp'
    torch.save(ckpt, tmp)
    os.replace(tmp, path)
```

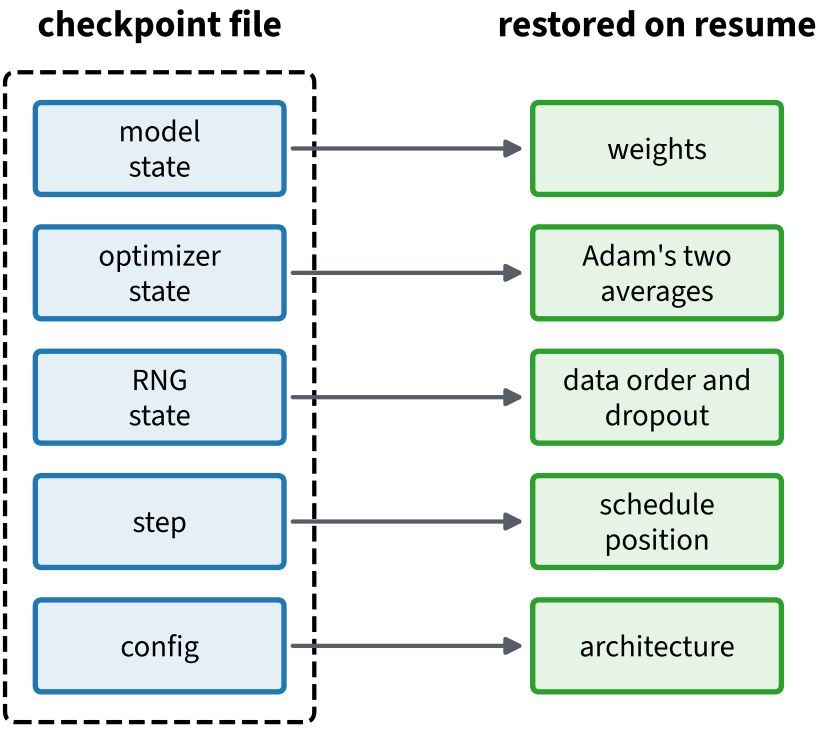


Figure 5.13: A checkpoint file's five compartments, each paired by an arrow with what it restores on resume: model state with the weights, optimizer state with Adam's two running averages, RNG state with the data order and dropout, step with the position in the schedule, and config with the architecture.

```
def load_checkpoint(path, model, optimizer=None, rng=None):
    """Restore the state written by save_checkpoint; return the raw dict."""
    ckpt = torch.load(path, weights_only=True)
    model.load_state_dict(ckpt['model'])
    if optimizer is not None:
        optimizer.load_state_dict(ckpt['optimizer'])
    if rng is not None:
        rng.set_state(ckpt['rng'])
    torch.set_rng_state(ckpt['cpu_rng'])
    if torch.cuda.is_available() and 'cuda_rng' in ckpt:
        torch.cuda.set_rng_state_all(ckpt['cuda_rng'])
    return ckpt
```

To test checkpoint resumption, we train a small 64-hidden-unit MLP regressor with Adam ($\eta = 0.05$) on 256 noisy linear examples in 20 dimensions, drawing minibatches of 64 examples using a dedicated random generator seeded at 0 so that minibatch ordering becomes part of the checkpointed state. As in Section 5.1, a `Config` dataclass captures all architectural and training hyperparameters so they travel directly inside the checkpoint.

```
@dataclass
class Config:
    num_inputs: int = 20
    num_hiddens: int = 64
    num_outputs: int = 1
    batch_size: int = 64
    lr: float = 0.05
```

We train for 200 steps, write a checkpoint, and immediately continue the same run for 5 more steps to create our uninterrupted 205-step ground truth. Each call to `train` records the minibatch loss before each update step.

```
def build(cfg):
    net = nn.Sequential(nn.Linear(cfg.num_inputs, cfg.num_hiddens),
                        nn.ReLU(),
                        nn.Linear(cfg.num_hiddens, cfg.num_outputs))
    for layer in (net[0], net[2]):
        nn.init.xavier_uniform_(layer.weight)
        nn.init.zeros_(layer.bias)
    return net

cfg = Config()
data = torch.randn(256, cfg.num_inputs)
target = data @ torch.randn(cfg.num_inputs, 1) + 0.1 * torch.randn(256, 1)
loss = nn.MSELoss()

def train(model, opt, rng, num_steps):
    losses = []
    for _ in range(num_steps):
        idx = torch.randint(len(data), (cfg.batch_size,), generator=rng)
        opt.zero_grad()
```

```
            l = loss(model(data[idx]), target[idx])
            l.backward()
            opt.step()
            losses.append(l.detach())
        return [l.item() for l in losses]

    net = build(cfg)
    opt = torch.optim.Adam(net.parameters(), lr=cfg.lr)
    rng = torch.Generator().manual_seed(0)      # draws the minibatches
    train(net, opt, rng, 200)
    path = 'run.pt'
    save_checkpoint(path, net, opt, step=200, cfg=cfg, rng=rng)
    uninterrupted = train(net, opt, rng, 5)
```

Now we simulate restarting in a fresh process: `resume` rebuilds the model and optimizer from the saved config, optionally restores the optimizer and RNG states alongside the weights, and runs the remaining steps up to `num_steps`, while `max_diff` reports the largest elementwise weight discrepancy between two models:

```
    def resume(path, num_steps, restore_optimizer=True, restore_rng=True):
        cfg = Config(**torch.load(path, weights_only=True)['cfg'])  # saved config
        model = build(cfg)
        opt = torch.optim.Adam(model.parameters(), lr=cfg.lr)
        rng = torch.Generator().manual_seed(0)  # seeded as at the start
        ckpt = load_checkpoint(path, model, opt if restore_optimizer else None,
                               rng=rng if restore_rng else None)
        return model, train(model, opt, rng, num_steps - ckpt['step'])

    def max_diff(a, b):
        return max((p - q).abs().max().item()
                   for p, q in zip(a.parameters(), b.parameters()))
```

We compare restoring the full checkpoint against omitting only the random generator (`restore_rng=False`, as if a script simply re-seeded from `0`) and measure the maximum weight discrepancy against the uninterrupted 205-step `net`:

```
    for name, kwargs in (('full checkpoint', {}),
                         ('without the generator', {'restore_rng': False})):
        resumed, _ = resume(path, 205, **kwargs)
        print(f'{name:22s} max |difference| {max_diff(resumed, net):.3g}')
```

```
full checkpoint        max |difference| 0
without the generator  max |difference| 0.00397
```

Restoring the full checkpoint reproduces the uninterrupted weights with zero error (bit for bit). Omitting the RNG state causes the resumed loop to replay the very first minibatches instead of drawing the next ones, introducing weight differences of roughly 0.004 to 0.02 (note that bit-for-bit equality also assumes deterministic operations, which we examine in Section 5.8).

Dropping the *optimizer* state is far more destructive. Below we resume with the saved weights and RNG state intact, but with a freshly constructed Adam optimizer, and compare

the five post-resume minibatch losses side by side:

```
_, full = resume(path, 205)
_, fresh = resume(path, 205, restore_optimizer=False)
print('saved optimizer:', [round(l, 4) for l in full])
print('fresh optimizer:', [round(l, 4) for l in fresh])
```

```
saved optimizer: [0.0009, 0.0017, 0.0022, 0.001, 0.0019]
fresh optimizer: [0.0009, 2.1294, 0.6979, 0.7146, 0.9511]
```

With the saved optimizer state, training picks up smoothly with losses hovering near 0.01 or below. With a fresh Adam optimizer, a *single* step blows the loss up roughly a thousandfold, and even four steps later it remains over a hundred times higher than before.

Why does one step of a fresh Adam wreck a nearly converged model? Equation Equation 5.11 tells the whole story. Starting from $\mathbf{m} = \mathbf{0}$ and $\mathbf{v} = \mathbf{0}$, Adam's bias correction on step 1 divides out the exact initialization factor, leaving $\hat{\mathbf{m}} = \mathbf{g}$ and $\hat{\mathbf{v}} = \mathbf{g} \odot \mathbf{g}$. Every parameter with any nonzero gradient therefore steps by $\eta\,|g_i|/(|g_i| + \epsilon) \approx \eta = 0.05$: the full learning rate, no matter how tiny its gradient. By contrast, the saved $\mathbf{v}$ remembers the much larger squared gradients from earlier in training, keeping $\sqrt{\hat{\mathbf{v}}}$ large relative to the tiny late-stage gradients so updates remain gentle. The closer a model is to convergence, the more violent an uninitialized Adam restart becomes. Weights alone are never enough to resume training.

5.6.4 Pretrained Weights

We now turn to the second major use of saved state: initializing from weights trained by someone else. A backbone trained on ImageNet (1.2 million images across 1,000 categories) learns rich visual representations that transfer cleanly to new tasks. Adapting it to a 10-class problem is simple: load every layer except the final classifier *head*, and swap in a fresh 10-output linear layer. Below we demonstrate this with ResNet-50 (~25 million parameters, or 23.5 million excluding its 1,000-class head). We will study convolutional networks in Chapter 6 and Chapter 7; for weight loading, ResNet-50 is simply a state dictionary of a few hundred named tensors.

```
net = resnet50(weights=ResNet50_Weights.DEFAULT)  # ~100 MB once
net.fc = nn.Linear(net.fc.in_features, 10)        # new 10-class head
net.fc
```

```
Linear(in_features=2048, out_features=10, bias=True)
```

External weight files rarely match a custom target model one-to-one, and silently ignoring mismatches is a notorious source of bugs. To see how a careful loader audits partial transfers, we strip two groups of tensors from our pretrained ResNet-50 state: the classification head (which we *intend* to replace) and the entire final stage of the backbone (simulating a corrupted or mismatched checkpoint). Loading the remaining tensors into a fresh 10-class ResNet-50 lets us inspect two diagnostic sets: *missing* keys (model parameters absent from the file, left randomly initialized) and *unexpected* keys (file entries unused by the model).

`load_state_dict(..., strict=False)` loads matching keys and returns `(missing_keys, unexpected_keys)` directly:

```
damaged = {k: v for k, v in net.state_dict().items()
           if not k.startswith(('fc.', 'layer4.'))}
target = resnet50(weights=None)
target.fc = nn.Linear(target.fc.in_features, 10)
report = target.load_state_dict(damaged, strict=False)
print('missing by block:', dict(Counter(
    k.split('.')[0] for k in report.missing_keys)))
print('unexpected:', report.unexpected_keys)
```

```
missing by block: {'layer4': 50, 'fc': 2}
unexpected: []
```

Missing keys for the new 10-class head's weight and bias are expected, since that head starts from random initialization. The missing keys from the final backbone stage are a red flag: had we loaded blindly with partial matching, a huge chunk of our "pretrained" ResNet-50 would silently remain random noise. Here the unexpected set is empty; a nonempty unexpected set typically signals a renamed module whose weights were never loaded. In production code, always assert explicitly that the missing and unexpected sets match *only* the layers you intended to replace or drop, raising an error on anything else.

Once verified, transfer learning typically begins by *freezing* the pretrained backbone (Section 5.2.3.2) so initial updates train only the new head, before selectively unfreezing deeper layers (Section 32.2).

Everything in this chapter so far has lived in host CPU memory. In Section 5.7, we move both state and computation onto GPUs.

5.6.5 Summary

Persisting a model saves its *state* (a dictionary mapping module-tree paths to parameter and buffer tensors), while the architecture itself lives in code. Because Python's `pickle` executes arbitrary code on load, weights shared across trust boundaries should use a code-free format such as `safetensors` (an 8-byte length prefix, a JSON header, and packed tensor bytes). Crash-safe training checkpoints write atomically via temporary files and capture five components: model state, optimizer state, RNG state, step counter, and config. Restoring all five resumes training bit for bit; dropping RNG state drifts the data stream, and dropping Adam's moments causes the first resumed step to move every parameter by the full learning rate, spiking the loss by orders of magnitude. When loading pretrained weights into a modified architecture, always audit the missing and unexpected key sets against your expectations rather than ignoring mismatches silently.

5.6.6 Exercises

1. code **Safetensors header.** The header length of the MLP's safetensors file is the 8-byte little-endian integer at the start of the file (Figure 5.12).
    1. Predict, then measure, how that length changes when the hidden width is doubled and when a third layer is added. Explain which parts of the header depend on tensor sizes at all.
    2. Read the hidden layer's bias from the file using only the header and file seeks, without the safetensors library. Decode the bytes with the dtype and shape that

the header records, and check that the result equals the tensor that `load_file` returns.
3. Explain why a pickle-based checkpoint cannot offer the same single-tensor read without deserializing the whole object graph.

2. code **Partial loading.** Define `MLP2`, a copy of `MLP` that inserts a second hidden layer `hidden2` of width 256 before the output layer and renames the attribute `output` to `head`.
   1. Load `mlp.safetensors` into `MLP2` with `load_state_dict(..., strict=False)`. Read the returned report and explain each missing and each unexpected key.
   2. Rename the file's keys to the names that `MLP2` uses before loading. Confirm that only `hidden2` keeps its fresh initialization, and make the load raise on any unexpected key and on any missing key outside `hidden2`.

   Adapted from Matthew Inkawhich, Saving and Loading Models, "Warmstarting Model Using Parameters from a Different Model".

3. code **Dtype round trip.** ● A `float16` number stores 5 exponent bits and 10 mantissa bits, against 8 and 23 in `float32` (Section 5.5). Cast the parameters of an `MLP` to `float16`, save them, and load them back into a `float32` `MLP`.
   1. Predict the largest relative error of a restored parameter from the mantissa width. Measure the maximum absolute difference and, over the nonzero parameters, the maximum relative difference between the restored and the original values. Explain any parameter whose relative error exceeds the prediction.
   2. Compare the outputs of the original and the restored `MLP` on `X`, and decide whether the round trip is acceptable for inference.
   3. Adam keeps a running average of squared gradients. Compute what `float16` stores for the square of a gradient entry of 10^{-4}, and explain why a resumable checkpoint keeps the optimizer state in `float32`.

4. [extended] **Crash-safe manifest.** ●
   1. Build a utility that writes model state, optimizer state, and a JSON metadata file under a fresh step-numbered prefix, then publishes the three by renaming into place one manifest that records each file's name, size, and checksum.
   2. Simulate two failures: a crash after the three files are written but before the manifest rename, and the truncation of one file of a published checkpoint. Show that reloading resumes from the previous complete checkpoint in the first case and rejects the corrupt checkpoint in the second.
   3. A rename is atomic, so a reader sees either the old manifest or the new one. POSIX does not require the data of the renamed files to reach the disk before the rename does. `os.fsync` forces the buffered writes of a file, or of a directory's entries, to disk. State where the utility must call `os.fsync` so that a power loss leaves either the previous checkpoint or the new one intact, and explain each call.

5.7 GPUs, Devices, and Memory

Since Section 2.4, our `Trainer` has accepted a `num_gpus` argument and politely ignored it. For the tiny models of earlier chapters that hardly mattered, since CPU training finished in seconds. Once models scale up, however, device placement makes the difference between

seconds and hours: a modern desktop GPU multiplies two 4096×4096 matrices in under 2 milliseconds, whereas a CPU needs a fifth of a second or more (as we benchmark in Section 5.7.2.1). That raw speed comes with a strict hardware constraint: the GPU has its own dedicated memory (say 24 GB on a desktop card), and every tensor it operates on must reside there.

Will a target model even fit? Our running 8-billion-parameter model needs 16 GB for its weights in bf16, which fits easily for inference. Training it with Adam in fp32, however, requires 128 GB of static state (Section 5.2) before storing a single activation, far more than any single 24 GB or 80 GB card. Whether a model fits on a device, and whether the GPU stays busy computing or stalls waiting for data, comes down to where every tensor lives and how many bytes it consumes.

In this section, we answer three core questions: how tensors and models are placed on and copied between CPUs and GPUs; how asynchronous GPU execution works (and how to benchmark GPU kernels accurately); and what actually fills GPU memory during training, including how *activation checkpointing* trades extra compute for dramatic memory savings. We then wire everything into the device-aware `Trainer` used throughout the rest of the book. Whereas Section 5.2 and Section 5.5 accounted for bytes per parameter and per number, here we put those bytes onto real hardware (deferring multi-GPU training to Chapter 13). Every code cell degrades gracefully to CPU if no GPU is available, confirming correctness even if timings require a GPU.

```
import time
from torch.utils.checkpoint import checkpoint
```

```
torch.set_float32_matmul_precision('high')  # allow tf32 in fp32 matmuls
```

5.7.1 Devices and Placement

5.7.1.1 Device Objects

Every tensor lives on a specific *device*: either the CPU paired with host RAM, or an individual GPU paired with its onboard memory. Running `nvidia-smi` in a terminal lists every NVIDIA GPU on your machine, its total and occupied memory, and active processes. While each framework names devices slightly differently, all four follow one governing principle: an operation executes on a single device, so all of its operands must live on that device.

`torch.device('cpu')` and `torch.device(f'cuda:{i}')` denote the CPU and GPU i.

We wrap device lookup in three small helpers that the rest of the book imports: one for CPU, one for GPU i, and one that counts available GPUs.

```
def cpu():
    """Get the CPU device."""
    return torch.device('cpu')

def gpu(i=0):
    """Get a GPU device."""
    return torch.device(f'cuda:{i}')
```

```
def num_gpus():
    """Get the number of available GPUs."""
    return torch.cuda.device_count()
```

Constructing a device object always succeeds; hardware existence is checked only when a tensor is placed on it.

Building on `num_gpus`, we define two fallback helpers that select a GPU when present and fall back to CPU otherwise, allowing identical notebook code to run unchanged on both laptops and GPU servers.

```
def try_gpu(i=0):
    """Return gpu(i) if exists, otherwise return cpu()."""
    if num_gpus() >= i + 1:
        return gpu(i)
    return cpu()

def try_all_gpus():
    """Return all available GPUs, or [cpu(),] if no GPU exists."""
    devices = [gpu(i) for i in range(num_gpus())]
    return devices if devices else [cpu()]

try_gpu(), try_gpu(10), try_all_gpus()
```

```
(device(type='cuda', index=0),
 device(type='cpu'),
 [device(type='cuda', index=0), device(type='cuda', index=1)])
```

Other accelerators, such as Apple-silicon GPUs (`mps`) and Google TPUs, use their own device identifiers, though the placement principles below apply equally to all of them. Throughout this book we focus on CPUs and CUDA GPUs.

5.7.1.2 Arrays on Devices

Every tensor exposes its location via its `device` attribute:

PyTorch tensors default to the CPU unless `device=` is passed.

Allocating directly on the target device avoids an extra CPU-to-GPU copy. Below we place `X` on the first GPU and `Y` on the second GPU (falling back to CPU when fewer than two GPUs are present) and try adding them:

```
X = torch.ones(2, 3, device=try_gpu())
Y = torch.rand(2, 3, device=try_gpu(1))
try:
    print(X + Y)
except RuntimeError as e:
    print(str(e).split(', but')[0])
```

```
Expected all tensors to be on the same device
```

Adding tensors on different devices raises an error immediately, preventing accidental cross-device transfers.

5.7.1.3 Copying Between Devices

Moving tensors across devices is vastly slower than computing on them (Section 5.7.2.1 measures the gap), so silently copying mismatched operands would hide severe performance bugs. Instead, we copy `X` explicitly onto `Y`'s device (Figure 5.14) before adding them.

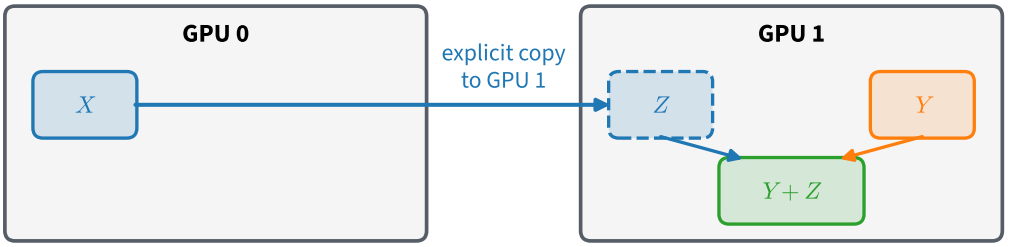


Figure 5.14: X lives on GPU 0 and Y on GPU 1. The copy Z of X on GPU 1 (dashed) lets Y + Z run entirely on GPU 1.

```
Z = X.to(try_gpu(1))
(Y + Z).device
```

```
device(type='cuda', index=1)
```

Now the sum runs and lands on the second GPU alongside both operands. The practical rule is simple: transfer data once at the boundary of your computation (moving a tensor to a device it already occupies is essentially a no-op) and keep every inner-loop operation on that single device.

5.7.1.4 Models on a Device

Because a forward pass combines inputs with model parameters, the entire model must live on the same device as its minibatches. Thanks to the module tree from Section 5.1, every framework moves or initializes all parameters in one call.

```
net = nn.Sequential(nn.Linear(3, 1))
net = net.to(device=try_gpu())
net(X)
```

```
tensor([[-0.2254],
        [-0.2254]], device='cuda:0', grad_fn=<AddmmBackward0>)
```

With inputs and parameters colocated on the same device, the forward pass runs there and returns its output on that device as well.

Keep two placement rules in mind: move or initialize the model on its target device *before* constructing the optimizer (so optimizer moments are allocated on the same device), and

allocate any new tensors created inside `forward` on `X.device` rather than defaulting to the CPU.

5.7.2 Asynchronous Execution

5.7.2.1 The Execution Queue

A GPU does not execute operations synchronously when Python invokes them. Instead, Python enqueues each kernel launch onto the device's command queue and returns immediately, letting the CPU prepare subsequent work while the GPU crunches through its queue in order. This pipelining hides Python overhead, but it comes with a crucial timing catch: measuring how long a Python loop takes measures *enqueueing*, not *execution*. Below we enqueue 32 products of 4096×4096 matrices and check the wall clock twice: right after Python finishes queuing them, and again once the device finishes computing them.

```
if torch.cuda.is_available():
    A = torch.randn(4096, 4096, device=gpu())
    A @ A
    torch.cuda.synchronize()  # Warm up and wait
    t = time.perf_counter()
    for _ in range(32):
        B = A @ A
    print(f'time to queue 32 matrix products: {time.perf_counter() - t:.4f} sec')
    torch.cuda.synchronize()
    print(f'time until they all finished:     {time.perf_counter() - t:.4f} sec')
else:
    print('No CUDA device: operations on the CPU run synchronously.')
```

```
time to queue 32 matrix products: 0.0010 sec
time until they all finished:     0.0535 sec
```

GPU kernels run asynchronously while CPU operations run synchronously; `torch.cuda.synchronize()` waits for all queued GPU work to finish.

Python finishes enqueuing all 32 matrix multiplications in a tiny fraction of the actual compute time. When does the CPU actually block and wait? Only when Python needs a concrete value back on the host—printing a tensor, converting a scalar to a Python float, converting to NumPy, or branching on a tensor in an `if` statement. Every such *synchronization point* stalls Python until the GPU drains its queue up to that value, leaving the GPU idle while Python catches up (Figure 5.15). Logging the loss as a Python number after every minibatch inserts a synchronization bubble on every step, which is why our `Trainer` keeps losses on-device inside a `d2l.Averager` and converts them to Python floats only once per epoch (Section 2.4).

Asynchronous dispatch also dictates four rules for trustworthy GPU benchmarks: (1) run warm-up calls first to exclude one-time allocation, library setup, and JIT compilation; (2) synchronize the device both immediately *before* starting the timer and immediately *before* stopping it; (3) average over multiple repetitions; and (4) use `time.perf_counter` for sub-millisecond resolution. (The demonstration cell above warmed up and synchronized but reported the 32-product total; our `Benchmark` helper below automates all four rules for this and later chapters.)

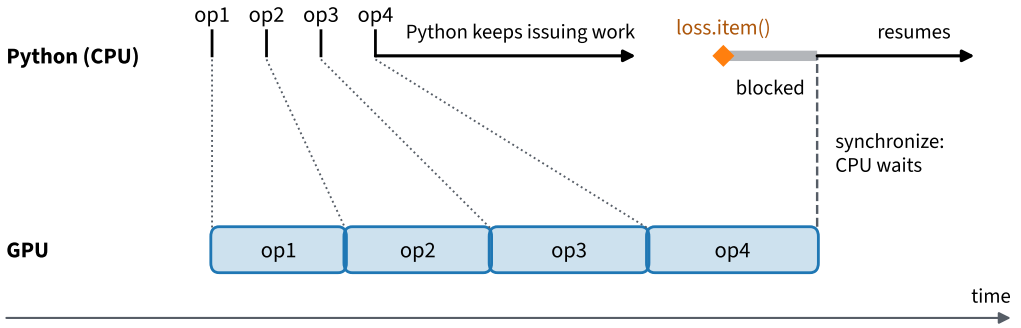


Figure 5.15: Python queues operations op1 through op4 and continues while the GPU runs them in order. Reading the loss as a Python number is a synchronization point: the CPU blocks until the GPU has drained its queue; Python then resumes, and the GPU idles until it receives more work.

```
class Benchmark:
    """Time a callable: warmup, then device-synchronized average seconds.
    `devices` lists the GPUs to wait for, by default the current one."""
    def __init__(self, f, warmup=3, repeats=10, desc='time', devices=None):
        def wait():  # CUDA runs asynchronously; the CPU does not
            if torch.cuda.is_available():
                for device in devices or [None]:
                    torch.cuda.synchronize(device)
        self.desc = desc
        for _ in range(warmup):
            f()
        wait()
        t0 = time.perf_counter()
        for _ in range(repeats):
            f()
        wait()
        self.time = (time.perf_counter() - t0) / repeats

    def __repr__(self):
        return f'{self.desc}: {1000 * self.time:.2f} ms/call'
```

We put `Benchmark` to work immediately on two claims from earlier: how much faster a GPU is than a CPU on large matrix multiplications, and how expensive host-to-device transfers are relative to compute. Below we time a 4096×4096 product on CPU (fp32), on GPU in tf32 (Section 5.5) and 16-bit precision, alongside copying a single 4096×4096 matrix from CPU RAM to the GPU.

```
if torch.cuda.is_available():
    A_cpu = A.cpu()  # The same matrix in main memory
    A16 = A.bfloat16()
    print(Benchmark(lambda: A_cpu @ A_cpu, warmup=1, repeats=3,
```

```
                    desc='product on the CPU, fp32'))
    print(Benchmark(lambda: A @ A, desc='product on the GPU, tf32'))
    print(Benchmark(lambda: A16 @ A16, desc='product on the GPU, bf16'))
    print(Benchmark(lambda: A_cpu.to(gpu()),
                    desc='copy from CPU to the GPU'))
else:
    print('No CUDA device: there is nothing to compare the CPU with.')
```

```
product on the CPU, fp32: 1423.37 ms/call
product on the GPU, tf32: 1.68 ms/call
product on the GPU, bf16: 0.91 ms/call
copy from CPU to the GPU: 5.32 ms/call
```

On a modern GPU, the 4096×4096 product finishes in under 2 ms, whereas the CPU requires about 100 ms to 2 s (roughly a $100\times$ to $1000\times$ speedup depending on CPU core count). Dropping to 16-bit precision cuts GPU time roughly in half again, delivering the tensor-core speedup promised in Section 5.5.

Pay special attention to the final line: merely *copying* the matrix from CPU memory to the GPU takes three to six times *longer* than multiplying it once it arrives, even though the multiplication executes 4096 multiply-adds per copied element. Standard host copies first stage data through *pinned* (page-locked) host memory so the GPU's DMA engine can pull it directly; pre-pinning batches eliminates that extra copy and allows transfers to overlap with compute (Section 13.2).

5.7.3 GPU Memory

5.7.3.1 Three Views of Memory

Ask how much GPU memory a training script uses and you can get three different numbers, depending on which layer of the stack you query. Requesting memory directly from the CUDA driver is slow, so frameworks run an internal *caching allocator* that grabs large slabs from the driver and sub-allocates them to tensors, retaining freed slabs for reuse rather than releasing them immediately. This creates three nested memory views (Figure 5.16): the bytes currently occupied by live tensors, the larger pool reserved by the framework's allocator, and the total memory assigned by the CUDA driver to the process (the figure displayed by `nvidia-smi`).

`torch.cuda.memory_allocated()` reports live tensor bytes, while `torch.cuda.memory_reserved()` reports the caching allocator's pool.

We allocate a 1 GiB float32 tensor and immediately delete it to watch these counters respond:

```
if torch.cuda.is_available():
    def report(tag):
        print(f'{tag}: {torch.cuda.memory_allocated() / 2**20:7.1f} MiB '
              f'allocated, {torch.cuda.memory_reserved() / 2**20:7.1f} MiB '
              'reserved')
    big = torch.zeros(256, 1024, 1024, device=gpu())  # 1 GiB of float32
    report('after creating big')
    del big
    report('after deleting it ')
```

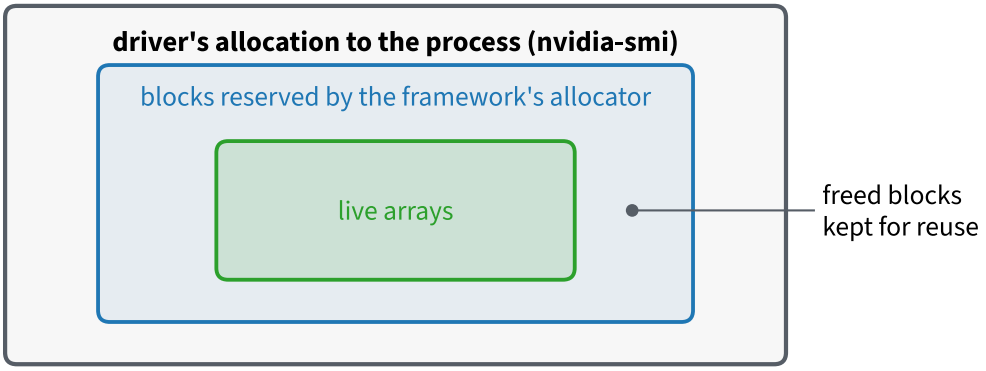


Figure 5.16: The three memory views nest: the driver's allocation to the process, which nvidia-smi reports, contains the allocator's reserved blocks, which contain the live arrays.

```
    else:
        print('No CUDA device: both counters read 0 on the CPU.')
```

```
after creating big:  1192.1 MiB allocated,  1302.0 MiB reserved
after deleting it :   168.1 MiB allocated,  1302.0 MiB reserved
```

Deleting `big` frees 1 GiB of live tensors, yet the framework's caching allocator retains the freed 1 GiB slab in its reserved pool so `nvidia-smi` does not budge. (Never force slabs back to the CUDA driver inside a training loop, as this adds slow driver allocations on every step.) The outermost driver view sets the hard out-of-memory ceiling, since the driver cannot hand out more RAM than the card physically holds. To diagnose *where* memory goes inside our code, we inspect live-byte counters, especially the *peak* memory reached during a call, which determines whether a step succeeds or crashes with out-of-memory (OOM).

`torch.cuda.reset_peak_memory_stats()` and `torch.cuda.max_memory_allocated()` measure the exact peak live memory delta of any call:

```
def peak_memory(f, *args):
    """Extra peak GPU memory allocated by f(*args), in bytes."""
    torch.cuda.synchronize()
    torch.cuda.reset_peak_memory_stats()
    base = torch.cuda.memory_allocated()
    f(*args)
    torch.cuda.synchronize()
    return torch.cuda.max_memory_allocated() - base
```

5.7.3.2 Memory During Training

In Section 5.2, we established the static memory floor for fp32 Adam training: $16N$ bytes for N parameters ($4N$ weights, $4N$ gradients, and $8N$ Adam moments). During an actual training step, a fourth term joins them: *activations*. Backpropagation must retain intermediate forward-pass outputs until the backward pass consumes them, and unlike the static

$16N$, activation memory scales linearly with minibatch size and layer width. We can calculate it before measuring: consider a three-hidden-layer MLP of width 4096 taking 1024-dimensional inputs ($N \approx 3.8 \times 10^7$ parameters, so $4N \approx 144$ MiB) at minibatch size 4096. The three hidden-layer outputs are each 4096×4096 fp32 tensors of 64 MiB (192 MiB in total), already a third larger than the weights themselves.

```
if torch.cuda.is_available():
    start = torch.cuda.memory_allocated()
    def allocated():  # Memory in use beyond that at the start of the cell
        return f'{(torch.cuda.memory_allocated() - start) / 2**20:7.1f} MiB'
    net = nn.Sequential(nn.Linear(1024, 4096), nn.ReLU(),
                        nn.Linear(4096, 4096), nn.ReLU(),
                        nn.Linear(4096, 4096), nn.ReLU(),
                        nn.Linear(4096, 10)).to(gpu())
    optimizer = torch.optim.Adam(net.parameters())
    X = torch.randn(4096, 1024, device=gpu())
    y = torch.randint(0, 10, (4096,), device=gpu())
    print('weights           ', allocated())
    loss = F.cross_entropy(net(X), y)
    print('+ activations     ', allocated())
    loss.backward()
    print('+ gradients       ', allocated())
    optimizer.step()
    print('+ optimizer state ', allocated())
else:
    print('No CUDA device: run this cell on a GPU to see the four readings.')
```

```
weights              160.2 MiB
+ activations        353.4 MiB
+ gradients          313.6 MiB
+ optimizer state    602.0 MiB
```

The readings trace the exact lifecycle of a training step: weights ($4N$ bytes) plus inputs sit in memory first; the forward pass adds the stored activations (~192 MiB, exceeding the weights); the backward pass consumes and frees those activations while materializing gradients ($4N$); and the first optimizer step allocates Adam's two moments ($8N$). Figure 5.17 draws the four stages without the 16 MiB input batch. From step two onward, memory rises and falls within every step: the forward pass stores the activations on top of the weights and Adam's moments, and the backward pass replaces them with gradients. The peak is at most the static $16N$ plus the activations. It is lower, by up to $4N$, when the loop frees the old gradients before the next forward pass. Shrinking the minibatch size is the quickest lever when activations push memory over the edge, since it cuts activation footprint without altering the architecture. By contrast, if memory climbs steadily *from step to step*, look for a leak keeping old computation graphs alive, the classic culprit being appending raw loss tensors to a Python list without detaching them first.

`optimizer.zero_grad()` frees the gradients by default (`set_to_none=True`), so a loop that calls it before the forward pass holds $12N$ bytes, not $16N$, while the activations are stored.

5.7.3.3 Activation Checkpointing

There is a second, architectural lever for activation memory. Standard backpropagation stores every layer's intermediate output only to read it once on the backward sweep. *Activa-*

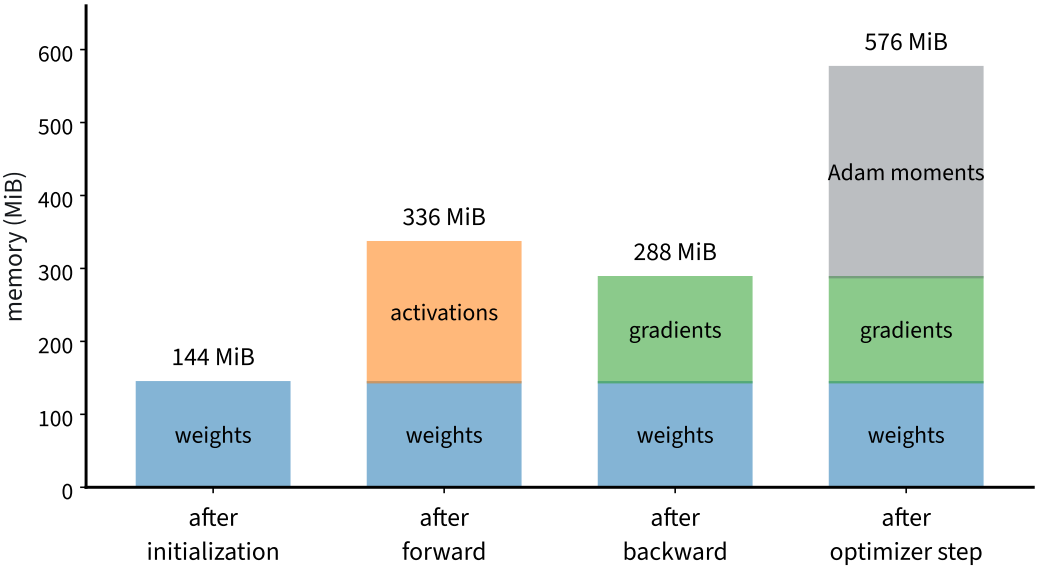


Figure 5.17: Memory in use after each stage of the first training step, for the MLP of width 4096 at minibatch size 4096. The input batch is not counted. The forward pass adds the three stored activations to the weights ($4N$). The backward pass frees the activations and leaves the gradients ($4N$). The optimizer step adds Adam's two moments ($8N$).

tion checkpointing (also called *gradient checkpointing* or *rematerialization*, (Chen et al. 2016)) discards most intermediate activations during the forward pass, retaining only the inputs at the boundaries of chosen segments, and recomputes each segment's internal activations on the fly during the backward pass. (Despite the shared word, this in-memory technique is unrelated to the disk checkpoints of Section 5.6.)

Partitioning K uniform layers into segments of s layers stores only K/s segment boundaries plus s live activations inside the active segment (about $K/s + s$ tensors instead of K). Choosing $s = \sqrt{K}$ drops peak activation memory from $O(K)$ down to $O(\sqrt{K})$. The trade-off is one extra forward pass per segment during backpropagation (Figure 5.18). Since a linear layer's backward pass computes two matrix products (input gradient and weight gradient) while its forward pass computes one, adding an extra forward pass increases total step compute from roughly 3 forward-pass equivalents to 4: about a 33% rough upper bound, depending on hardware and layer mix.

Activation checkpointing shines on deep stacks of identical blocks, such as our residual MLP (Section 5.1) and modern transformers. Below we define `ResidualBlock` with $\mathcal{N}(0, 0.01^2)$ weights and zero biases, along with a `run_stack` helper that groups blocks into segments with optional checkpointing.

```
class ResidualBlock(nn.Module):
    def __init__(self, num_hiddens):
        super().__init__()
        self.body = nn.Sequential(
            nn.Linear(num_hiddens, num_hiddens), nn.ReLU(),
```

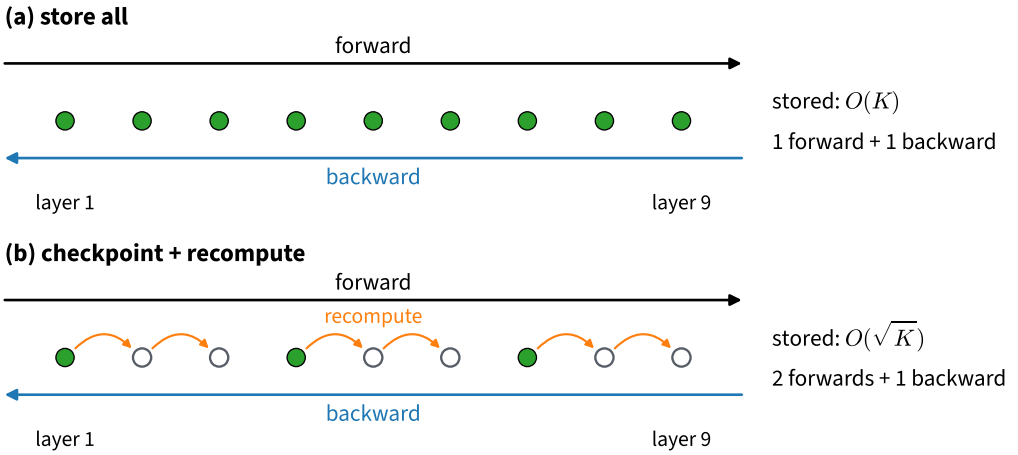


Figure 5.18: (a) Standard backpropagation stores the activations of every layer. (b) Checkpointing stores only the inputs of the segments (filled) and recomputes the activations inside each segment (open) during the backward pass. With K layers in segments of $\sqrt{K}$ layers, stored activations fall from $O(K)$ to $O(\sqrt{K})$ at the cost of one extra forward pass.

```
            nn.Linear(num_hiddens, num_hiddens))
        for layer in self.body[0], self.body[2]:
            nn.init.normal_(layer.weight, std=0.01)
            nn.init.zeros_(layer.bias)

    def forward(self, X):
        return X + self.body(X)

def run_stack(blocks, X, use_checkpoint=False, segment_size=4):
    for start in range(0, len(blocks), segment_size):
        stop = min(start + segment_size, len(blocks))
        # Bind this segment's range now: the backward pass calls segment
        # again after the loop has moved on
        def segment(X, start=start, stop=stop):
            for i in range(start, stop):
                X = blocks[i](X)
            return X
        X = checkpoint(segment, X, use_reentrant=False) \
            if use_checkpoint else segment(X)
    return X
```

`torch.utils.checkpoint.checkpoint(segment, X, use_reentrant=False)` discards intermediate activations inside `segment`, saves and restores RNG state (keeping dropout consistent), and reruns `segment` during `backward()`.

First, a sanity check: recomputing activations must never alter the math. We verify on an eight-block stack split into two checkpointed segments that all gradients match standard backpropagation.

```
blocks = nn.Sequential(*[ResidualBlock(64) for _ in range(8)])
X = torch.randn(32, 64)

def stack_grads(use_checkpoint):
    blocks.zero_grad(set_to_none=True)
    run_stack(blocks, X, use_checkpoint).sum().backward()
    return [p.grad.clone() for p in blocks.parameters()]

for g, g_ckpt in zip(stack_grads(False), stack_grads(True)):
    assert torch.allclose(g, g_ckpt)
print('checkpointed gradients match the ordinary ones')
```

```
checkpointed gradients match the ordinary ones
```

That correctness check runs anywhere; to measure the memory–compute trade-off, we move to a GPU with a deeper stack of 16 blocks (width 1024, batch size 8192) grouped into four-block segments, reporting both *step memory* (peak memory beyond parameters and inputs) and wall-clock time via `Benchmark`.

```
if torch.cuda.is_available():
    blocks = nn.Sequential(*[ResidualBlock(1024) for _ in range(16)]).to(gpu())
    X = torch.randn(8192, 1024, device=gpu())
    def step(use_checkpoint):
        blocks.zero_grad(set_to_none=True)
        run_stack(blocks, X, use_checkpoint).sum().backward()
    for use_checkpoint in (False, True):
        timing = Benchmark(lambda: step(use_checkpoint))
        blocks.zero_grad(set_to_none=True)  # Gradients count as step memory
        peak = peak_memory(step, use_checkpoint)
        print(f'checkpointing={use_checkpoint!s:5}  step memory '
              f'{peak / 2**20:5.0f} MiB, {timing.time * 1e3:5.1f} ms per step')
else:
    print('No CUDA device: the peak-memory comparison needs a GPU.')
```

```
checkpointing=False  step memory  1096 MiB,  25.5 ms per step
checkpointing=True   step memory   452 MiB,  32.3 ms per step
```

Without checkpointing, the backward pass holds activations for all 16 blocks at once; grouping into four-block segments keeps only four segment boundaries plus one active segment, cutting step memory from roughly 1.1 GiB down to 0.4–0.5 GiB for roughly a quarter more compute time (within our back-of-the-envelope 33% bound). That favorable exchange makes activation checkpointing standard practice when training large transformers and deep vision models.

5.7.4 Device-Aware Trainer

Now we can finally wire `num_gpus` into `d2l.Trainer`. Putting our rules into practice takes just two steps: place the model parameters on the target device before training begins, and move each minibatch to that device at the start of every step.

```
@d2l.add_to_class(d2l.Trainer)
def __init__(self, max_epochs, num_gpus=0, gradient_clip_val=0):
    self.save_hyperparameters()
    self.gpus = [d2l.gpu(i) for i in range(min(num_gpus, d2l.num_gpus()))]
    self.board = d2l.ProgressBoard(xlabel='epoch', ylabel='loss')

@d2l.add_to_class(d2l.Trainer)
def prepare_batch(self, batch):
    if self.gpus:
        batch = [d2l.to(a, self.gpus[0]) for a in batch]
    return batch

@d2l.add_to_class(d2l.Trainer)
def prepare_model(self, model):
    self.model = model
    if self.gpus:
        model.to(self.gpus[0])
        # Pick convolution algorithms by timing, unless determinism is wanted
        torch.backends.cudnn.benchmark = not (
            torch.are_deterministic_algorithms_enabled()
            or torch.backends.cudnn.deterministic)
    # Lazy parameters get their shapes from a dry pass on the device, then
    # the model's weight_init, if it has one, before the optimizer exists
    if any(isinstance(p, nn.parameter.UninitializedParameter)
           for p in model.parameters()):
        loader = (self.train_dataloader if self.train_dataloader is not None
                  else self.val_dataloader)  # `evaluate` has no training data
        batch = self.prepare_batch(next(iter(loader)))
        model.apply_init(batch[:-1], getattr(model, 'weight_init', None))
```

Using `min(num_gpus, d2l.num_gpus())` treats `num_gpus` as an upper bound rather than a hard requirement: on CPU-only machines, `self.gpus` is empty and training runs seamlessly on the CPU, the exact fallback behind every `Trainer(num_gpus=1)` call throughout the rest of the book.

How much faster does a GPU actually train a small model? While our 4096×4096 matrix product sped up by over $100\times$, an end-to-end training loop includes data loading, Python dispatch, and smaller layers. Below we measure a Fashion-MNIST classifier built from two 256-wide residual blocks (Section 5.7.3.3) with $\mathcal{N}(0, 0.01^2)$ weights and zero biases.

```
class ResMLPClassifier(d2l.Classifier):
    def __init__(self, num_hiddens=256, num_blocks=2, lr=0.1):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(
            nn.Flatten(), nn.Linear(784, num_hiddens), nn.ReLU(),
            *[ResidualBlock(num_hiddens) for _ in range(num_blocks)],
            nn.Linear(num_hiddens, 10))
        for layer in self.net[1], self.net[-1]:  # The blocks set their own
            nn.init.normal_(layer.weight, std=0.01)
            nn.init.zeros_(layer.bias)
```

We compare one post-warmup epoch on Fashion-MNIST with `num_gpus=0` versus `num_gpus=1`:

```
data = d2l.FashionMNIST(batch_size=256)

def epoch_time(num_gpus):
    model = ResMLPClassifier()
    trainer = d2l.Trainer(max_epochs=1, num_gpus=num_gpus)
    trainer.board.display = False
    trainer.fit(model, data)  # The first epoch warms up
    return Benchmark(trainer.fit_epoch, warmup=0, repeats=1).time

for gpus in (0, 1):
    print(f'num_gpus={gpus}: {epoch_time(num_gpus=gpus):.1f} sec per epoch')
```

```
num_gpus=0: 2.9 sec per epoch
num_gpus=1: 1.1 sec per epoch
```

The speedup is a useful reality check: on this tiny Fashion-MNIST network, a GPU epoch runs at most a few times faster than CPU, nowhere near the $100\times$ gain on 4096×4096 matrix multiplies. Here the largest matrix is only 784×256, leaving the GPU vastly underutilized while data loading and kernel-launch overhead dominate the clock. GPUs pay off once large convolutions, attention heads, and wide linear layers dominate the workload, as they do starting in the next chapter.

From a memory perspective, this small SGD model fits comfortably with gigabytes to spare. Return to our 8-billion-parameter model, however: shrinking batch size and enabling activation checkpointing tame activation memory, yet the 128 GB static floor of fp32 weights, gradients, and Adam moments remains untouched, and remember that `Trainer` cannot warn you ahead of time before hitting OOM. Back-of-the-envelope memory budgeting remains essential, and scaling past a single device's memory brings us to multi-GPU parallelism in Chapter 13.

5.7.5 Summary

Every tensor and model parameter resides on a specific device, and every operation requires its operands on the same device. Because host–device transfers are far slower than GPU compute, place models once upfront, transfer batches at step boundaries, and keep inner loops on-device. GPUs execute asynchronously from a command queue, meaning Python stalls only at synchronization points (such as converting a scalar to a Python float, or NumPy conversions) and reliable benchmarks require warm-up, explicit synchronization, and repeated trials. Training memory consists of static model/optimizer state plus batch-dependent activations; activation checkpointing cuts activation memory from $O(K)$ to $O(\sqrt{K})$ at the cost of roughly one extra forward pass. Finally, our device-aware `Trainer` automates single-GPU placement while falling back cleanly to CPU when no GPU is present.

5.7.6 Exercises

1. code **Peak memory.** Take a four-layer MLP with widths 1024, 4096, 4096, 4096 and 10 outputs, trained with Adam, as in Section 5.7.3.2. Using the accounting of that subsection, predict the peak memory of one training step at batch sizes 64, 256, and 1024, counting weights, gradients, optimizer state, and the activations the backward pass keeps. Then measure the peak on a GPU. Where does the prediction break down, and what did it omit?

2. **Checkpointing arithmetic.** Work out the schedule of Figure 5.18 on paper for a stack of sixteen `ResidualBlock`s of width 1024 at batch size 8192.
    1. Name the arrays inside one block that the backward pass must keep, and count their entries.
    2. Compute the activation memory held at the peak of the backward pass without checkpointing and with segments of one, two, four, and eight blocks. State how much forward computation each schedule repeats.
    3. Which segment size would you choose here, and how does it relate to the rule of segments of about $\sqrt{K}$ layers in Section 5.7.3.3?
3. code **Asynchronous dispatch.** ● Use the timing rules of Section 5.7.2.1 for both parts.
    1. For n from 256 to 8192, time a round trip of an $n \times n$ matrix from CPU to GPU and back, and time one $n \times n$ matrix product on the device. Find the n at which the two times are equal, extrapolating from how each time grows with n if they do not cross in this range. Predict this crossover from your GPU's published host-to-device bandwidth and matrix-product throughput, and explain any gap between prediction and measurement.
    2. If you have two GPUs, time 1000 matrix products of two 4096×4096 matrices executed on one GPU, then 500 on each of two GPUs issued from the same loop. Measure the scaling, explain how asynchronous dispatch lets Python issue work to both devices, and identify sources of deviation from linear scaling. Guard the experiment with `num_gpus() >= 2`.
4. code **Synchronization points.** ● Consider the training run of `ResMLPClassifier` in Section 5.7.4 on a GPU.
    1. List every synchronization point in one training step, reading `Trainer.fit_epoch`, `Trainer.prepare_batch`, and `Averager` as well as the model code. Classify each as unavoidable for a correct loop or as an avoidable instrumentation artifact, and justify each classification.
    2. Suppose the loop reads the loss into a Python float after every step. Predict from your list how much this readout slows down the default `ResMLPClassifier` and `ResMLPClassifier(num_hiddens=2048, num_blocks=8)`. Then time one epoch of each model with and without the readout, and explain the four timings.

Adapted from the PyTorch documentation CUDA semantics.

5.8 Reproducibility and Inspection

Suppose you tweak a single line in a training script and validation accuracy on Fashion-MNIST edges up by 0.3 percentage points. Before celebrating, you need to know how much two runs of the *unchanged* script vary on their own. In Section 5.8.1.5, we run softmax regression five times differing only in their random seed and watch final accuracies span one to four percentage points simply from different initial weights and batch shuffling, swamping a 0.3 point gain completely.

Debugging a crashing training run presents the mirror-image problem: if the loss explodes to NaN after hours of training, we want to reproduce the exact sequence of steps leading up to the crash and then peer inside the running network to pinpoint which layer emitted the first non-finite value.

This section addresses both halves together. First, we make runs repeatable by controlling pseudorandom number generators—global seeds, private generator objects, and multi-worker data loaders—and then tackle the trickier GPU wrinkle: floating-point non-associativity under parallel atomic reductions. Second, we inspect live models without modifying their code using *hooks* (and their framework equivalents) to log activation scales, catch the first NaN layer, and monitor gradient norms. Both halves reinforce each other: internal diagnostics are only comparable across two code versions when nothing else changes between runs. On a first pass, feel free to skim the subsections on generator objects and data-loader workers, revisiting them once your data pipelines perform random augmentations.

```
from torch.utils.data import DataLoader, Dataset
```

5.8.1 Seeds and Random Number Generators

Randomness sneaks into a training pipeline from far more places than most practitioners realize: weight initialization (Section 5.3), dropout masks at every step, epoch-by-epoch minibatch shuffling, random data augmentations (crops, flips, color jitter), and parallel data-loader worker processes. Every random number comes from a *pseudorandom number generator* (PRNG), a deterministic algorithm whose entire sequence is determined by its initial *seed*. Seeding every generator the pipeline touches therefore locks down every random draw.

5.8.1.1 Global Seeds

Deep learning frameworks either pull random numbers from a shared global generator or require passing generator state explicitly as an argument. Below we demonstrate seeding on a small five-step training loop where weights and synthetic inputs all come from one seeded source.

`torch.manual_seed(k)` seeds the framework's global CPU and GPU generators (note that Python's `random` and NumPy's `np.random` must be seeded separately if used).

```
def train_once(seed):
    torch.manual_seed(seed)  # seeds the CPU and every GPU
    net = nn.Sequential(nn.Linear(20, 32), nn.ReLU(), nn.Linear(32, 1))
    opt = torch.optim.SGD(net.parameters(), lr=0.1)
    X, y = torch.randn(128, 20), torch.randn(128, 1)
    for _ in range(5):
        opt.zero_grad()
        loss = ((net(X) - y) ** 2).mean()
        loss.backward()
        opt.step()
    return loss.item(), net[0].weight.detach().clone()

loss_a, w_a = train_once(seed=0)
loss_b, w_b = train_once(seed=0)
loss_c, _ = train_once(seed=1)
print('same seed, identical loss and weights:',
      loss_a == loss_b, torch.equal(w_a, w_b))
```

```
print('different seed:', loss_a, 'vs', loss_c)
```

```
same seed, identical loss and weights: True True
different seed: 0.8643209338188171 vs 0.9312677383422852
```

Both runs with `seed=0` match *bit for bit* (every weight and the final loss are identical) because identical initial weights, data, and deterministic arithmetic produce identical gradients (we examine operation order in Section 5.8.2). Changing the seed to `1` yields a distinct trajectory and loss.

5.8.1.2 Generator Objects

Relying on a single global PRNG couples every piece of code that draws from it: adding a single extra layer or a temporary random print during debugging shifts the entire downstream stream, silently altering train/validation splits, weight initializations, and dropout masks all at once. A private *generator object* isolates a stream behind its own seed so unrelated draws elsewhere can never perturb it.

`torch.Generator().manual_seed(42)` creates an isolated PRNG stream unaffected by global draws:

```
g = torch.Generator().manual_seed(42)
split = torch.randperm(10, generator=g)
_ = torch.randn(1000)  # unrelated draws from the global generator
split_again = torch.randperm(10, generator=g.manual_seed(42))
print(torch.equal(split, split_again), split)
```

```
True tensor([2, 6, 1, 8, 4, 5, 0, 9, 3, 7])
```

5.8.1.3 Data Loader Workers

When data loaders prepare minibatches in parallel *worker* processes, random data augmentation introduces two silent failure modes. If workers are spawned via OS `fork` as clones of the main process without re-seeding, every worker inherits the exact same generator state and generates identical "random" augmentations. Conversely, if each worker initializes from OS entropy, no two runs will ever match.

`DataLoader` seeds each worker's PyTorch, `random`, and global NumPy RNGs from `base_seed + worker_id`, but a private `np.random.default_rng()` attribute on the `Dataset` is cloned identically across forked workers unless re-seeded via `worker_init_fn` using `torch.utils.data.get_worker_info().seed`:

```
class NoisyData(Dataset):
    """Eight examples whose values are random draws, as in augmentation."""
    def __init__(self):
        self.rng = np.random.default_rng()  # the dataset's own generator

    def __len__(self):
        return 8

    def __getitem__(self, i):
```

```
        return self.rng.random()

def seed_worker(worker_id):
    info = torch.utils.data.get_worker_info()
    info.dataset.rng = np.random.default_rng(info.seed)  # this worker's seed

for init_fn in (None, seed_worker):
    loader = DataLoader(NoisyData(), batch_size=4, num_workers=2,
                        worker_init_fn=init_fn,
                        multiprocessing_context='fork',  # copies of this one
                        generator=torch.Generator().manual_seed(0))
    print([[round(x, 3) for x in batch.tolist()] for batch in loader])
```

```
[[0.61, 0.817, 0.884, 0.742], [0.61, 0.817, 0.884, 0.742]]
[[0.777, 0.641, 0.352, 0.467], [0.653, 0.267, 0.932, 0.823]]
```

Without per-worker or per-example re-seeding, both workers output identical "random" numbers in the first line; proper seeding gives distinct, reproducible draws in the second line.

5.8.1.4 Randomness as a Value

Every worker bug above stems from hidden mutable generator state being copied or shared implicitly behind the scenes. Functional random APIs eliminate hidden state by treating randomness as an explicit value: every random call takes a key, calling it twice with the same key returns identical numbers, and independent streams come from explicitly splitting keys. While you can still reuse a key accidentally, the reuse is right there in the code. The price is plumbing keys through every stochastic function. A global seed is simply the implicit shorthand for that same contract, and private generator objects recover explicit isolation wherever needed.

5.8.1.5 The Spread across Seeds

Now we measure the seed-to-seed variance mentioned at the start of the section. Below we train softmax regression (Section 3.3.5) on Fashion-MNIST for three epochs on a GPU across five seeds (0 through 4), seeding every generator involved and returning each run's final validation accuracy and weights.

```
def train_classifier(seed, data):
    torch.manual_seed(seed)  # initialization and shuffle order
    model = d2l.SoftmaxRegression(num_outputs=10, lr=0.1)
    trainer = d2l.Trainer(max_epochs=3, num_gpus=1)
    trainer.board.display = False
    trainer.fit(model, data)
    return trainer.history['val_acc'][-1], model.net[1].weight.detach().cpu()

data = d2l.FashionMNIST(batch_size=256)
runs = [train_classifier(seed, data) for seed in range(5)]
accs = 100 * np.array([acc for acc, _ in runs])
print('validation accuracy (%):', ' '.join(f'{a:.1f}' for a in accs))
print(f'mean {accs.mean():.1f}, max - min {np.ptp(accs):.1f} points')
```

```
validation accuracy (%): 79.8 81.8 81.9 78.3 82.1
mean 80.8, max - min 3.7 points
```

Even though all five runs execute identical code and hyperparameters, their final accuracies spread by one to four percentage points around 81%, confirming that a 0.3-point jump from a single run is pure noise. Sound empirical comparisons always report the mean and spread across multiple seeds.

To verify that seed choice alone drove those differences, we rerun `seed=0` and compare both accuracy and every weight against the first run:

```
acc, w = train_classifier(0, data)
print('same accuracy:', acc == runs[0][0],
      '| same weights:', torch.equal(w, runs[0][1]),
      f'| largest difference {(w - runs[0][1]).abs().max().item():.2g}')
```

```
same accuracy: True | same weights: True | largest difference 0
```

The rerun matches the first run's validation accuracy identically. Whether every single weight matches down to the last bit depends on whether the GPU sums floating-point numbers in a fixed order, something seeds alone do not control (Section 5.8.2).

5.8.2 Deterministic Arithmetic

Seeding fixes *which* random numbers a run draws, not *how* floating-point arithmetic combines them. Because every floating-point addition rounds to the nearest representable value, floating-point addition is *not* associative: changing summation order can change the result.

Three float32 numbers make the effect unmistakable. Near 10^8, adjacent float32 numbers lie 8 apart (Section 5.5), so $-10^8 + 1$ rounds right back to -10^8:

```
a, b, c = [torch.tensor(v, dtype=torch.float32) for v in (1e8, -1e8, 1.0)]
print(float((a + b) + c), float(a + (b + c)))  # same numbers, regrouped
```

```
1.0 0.0
```

Simply shifting parentheses changes the answer from `1.0` to `0.0`. In long reductions the discrepancy is subtler but pervasive: summing a million floats forward versus backward routinely differs in the lowest bits.

On a CPU with a fixed thread count, reduction order usually stays constant, so seeded runs repeat cleanly. On a GPU, thousands of threads race in parallel, and several fast kernels (including embedding gradients, scatter-adds, certain convolution backward passes, and reductions) accumulate partial results via *atomic additions* that land in whatever order threads finish. Two identically seeded runs on the same GPU can therefore diverge in their lowest bits, which chaotic gradient steps eventually amplify into visibly distinct loss curves. Every framework provides a determinism mode (or alternative kernel) that forces a fixed summation order, typically at some speed cost. Below we observe this directly on a GPU *scatter-add* summing one million random numbers into ten bins twice in a row.

`torch.use_deterministic_algorithms(True)` switches operations like `index_add_` to deterministic kernels (or raises a `RuntimeError` if none exists):

```
if torch.cuda.is_available():
    src = torch.randn(2**20, device='cuda')  # a million numbers
    idx = torch.randint(0, 10, (2**20,), device='cuda')  # their bins
    for deterministic in (False, True):
        torch.use_deterministic_algorithms(deterministic)
        a, b = [torch.zeros(10, device='cuda').index_add_(0, idx, src)
                for _ in range(2)]
        print(f'deterministic {deterministic!s:5}: bitwise equal '
              f'{torch.equal(a, b)}, largest difference '
              f'{(a - b).abs().max().item():.2g}')
    torch.use_deterministic_algorithms(False)
else:
    print('no GPU: on a CPU this sum adds in a fixed order')
```

```
deterministic False: bitwise equal False, largest difference 0.0051
deterministic True : bitwise equal True, largest difference 0
```

Without determinism enabled, two consecutive scatter-adds on the exact same GPU data differ by a few thousandths on sums of a few hundred; with a deterministic kernel, both calls match bit for bit.

Remember the proper scope of bitwise reproducibility: it holds only on a single machine, software stack, and hardware configuration; no framework promises identical bits across library upgrades, different GPU architectures, or CPU vs. GPU. Bitwise determinism is an invaluable *debugging* tool for bisecting the exact step where two runs diverge. Scientific reproducibility is both looser and stronger: any genuine improvement ("this architectural change improves accuracy") must hold robustly across multiple seeds, reported as mean and spread just as in Section 5.8.1.5 (~81%, spanning 1–4 points). As with 16-bit formats (Section 5.5), any claim that flips under a new seed or last-bit rounding needs stronger evidence.

5.8.3 Model Inspection with Hooks

Repeating a bug reliably proves it is real, not *where* inside the network it originates. Inserting print statements into every layer's `forward` method is clumsy, and impractical for third-party library models whose source code you do not want to fork. A *hook* solves this cleanly: it is a callback registered from outside the model that the framework invokes automatically before or after a layer executes.

`module.register_forward_hook(f)` runs `f(module, inputs, output)` inside `__call__` after `forward` (and returning a new tensor replaces the output).

For our inspection examples, we build an eight-block residual stack of width 64 between a 20-input layer and a 10-output head (Section 5.1), each block computing $\mathbf{h} = \mathbf{x} + \text{body}(\mathbf{x})$ with Xavier-initialized weights (Section 4.4.2.1) and zero biases. We deliberately leave this network unseeded so each run draws fresh random weights, which is why the diagnostic measurements below are cited as typical ranges.

The seeded runs above left PyTorch's global generator at a fixed state, which would fix this model's weights too, so the cell first reseeds the generator from the operating system's entropy with `torch.seed()`.

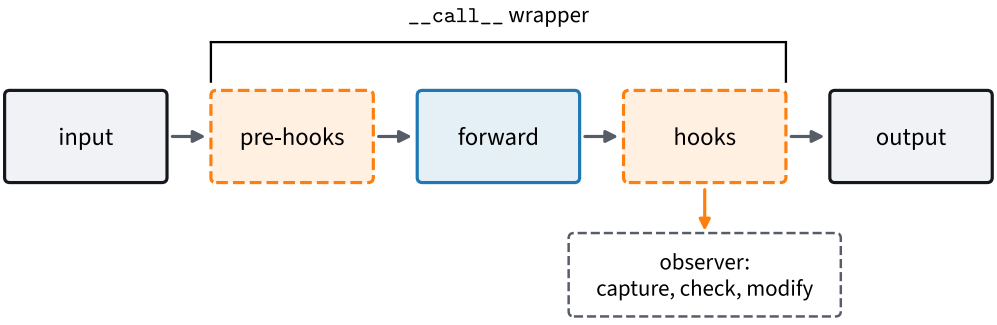


Figure 5.19: The `__call__` wrapper as a pipeline: input flows through pre-hooks, then forward, then hooks, to the output, with the two hook stages dashed and orange against forward's solid blue, and a side arrow from the hooks stage to an observer that can capture, check, or modify.

```
torch.seed()  # a fresh seed from the operating system

class ResidualBlock(nn.Module):
    def __init__(self, num_hiddens):
        super().__init__()
        self.body = nn.Sequential(nn.Linear(num_hiddens, num_hiddens),
                                  nn.ReLU(),
                                  nn.Linear(num_hiddens, num_hiddens))

    def forward(self, X):
        return X + self.body(X)

net = nn.Sequential(nn.Linear(20, 64),
                    *[ResidualBlock(64) for _ in range(8)],
                    nn.Linear(64, 10))
for m in net.modules():
    if isinstance(m, nn.Linear):  # Xavier weights, zero biases
        nn.init.xavier_uniform_(m.weight)
        nn.init.zeros_(m.bias)
```

5.8.3.1 Activation Statistics

In Section 5.3, we manually looped through blocks to measure activation standard deviations across depth. With hooks, we can collect the exact same profile from an unmodified model in just a few lines on a minibatch of 256 standard normal inputs:

```
stats, handles = [], []

def record(module, inputs, output):
    stats.append((type(module).__name__, output.detach().std().item()))
```

```
    for m in net:
        handles.append(m.register_forward_hook(record))
    net(torch.randn(256, 20))
    for h in handles:
        h.remove()
    for name, std in stats:
        print(f'{name:15s} std {std:.2f}')
```

```
Linear          std 0.72
ResidualBlock   std 0.88
ResidualBlock   std 1.09
ResidualBlock   std 1.31
ResidualBlock   std 1.62
ResidualBlock   std 1.98
ResidualBlock   std 2.28
ResidualBlock   std 2.93
ResidualBlock   std 3.58
Linear          std 3.51
```

Activation standard deviations climb steadily with depth, from roughly 0.7 right after the input layer to around 3–4.5 after the eighth block, matching theory closely. Xavier initialization gives the $20 \to 64$ input layer weight variance $2/84$, yielding output variance $20 \cdot 2/84 \approx 0.48$ (standard deviation 0.69). Each residual block then adds an approximately uncorrelated body output whose variance is half its input variance (the first linear layer preserves variance, ReLU halves the mean square, and the second linear layer preserves it), multiplying total variance by 1.5 (and standard deviation by $\sqrt{1.5} \approx 1.22$) at every block. Across eight blocks, $0.69 \times 1.5^4 \approx 3.5$, right in the middle of our measurements. This is the exact variance-growth analysis of Section 5.3 (with factor 1.5 here instead of 2 under He initialization), underscoring why deep residual branches benefit from scaled-down output projections.

Two rules keep hooks leak-free: always `.detach()` any tensor stored inside a hook (otherwise the entire autograd graph stays alive in memory across steps), and always remove the hook handle (`h.remove()`) once inspection is done.

5.8.3.2 A NaN Finder

When a training loss suddenly turns NaN at step 40,000, your first diagnostic question is *which layer emitted the first non-finite value?* Attaching a finiteness hook to every leaf layer pinpoints the culprit in a single forward pass, because hooks execute in forward execution order. To test our detector, we deliberately inject a NaN into one weight of the third residual block (index 3):

```
def make_finite_check(name):
    def check(module, inputs, output):
        if not torch.isfinite(output).all():
            raise RuntimeError(f'first non-finite output in {name}')
    return check

handles = [m.register_forward_hook(make_finite_check(name))
           for name, m in net.named_modules()
```

```
            if len(list(m.children())) == 0]  # leaf modules only
saved = net[3].body[0].weight[0, 0].item()
with torch.no_grad():
    net[3].body[0].weight[0, 0] = float('nan')  # break one layer
try:
    net(torch.randn(2, 20))
except RuntimeError as e:
    print(e)
for h in handles:
    h.remove()
with torch.no_grad():
    net[3].body[0].weight[0, 0] = saved  # restore the weight
```

```
first non-finite output in 3.body.0
```

The hook immediately identifies the corrupted layer in block 3 without adding a single print statement inside the layer classes. Of course, catching a planted NaN is the clean demo case; in a real training crash, the finder points you to the first *symptom*, while the underlying cause (such as runaway activation growth culminating in overflow) may begin several layers earlier.

5.8.3.3 Gradient Inspection

We can monitor the backward pass with the same precision. Since forward activations grew across our eight residual blocks, do gradients grow in reverse? Using the mean squared output on 256 standard normal inputs as our loss, we log the gradient norm at each residual block's output alongside the weight and bias gradient norms of the first linear layer in block 3.

`module.register_full_backward_hook(f)` invokes `f(module, grad_input, grad_output)` during `backward()` (firing in reverse topological order, from last block to first):

```
X = torch.randn(256, 20)
grad_norms, handles = [], []
def record(module, grad_input, grad_output):
    grad_norms.append(grad_output[0].norm().item())  # store a number
for block in net[1:-1]:
    handles.append(block.register_full_backward_hook(record))
net.zero_grad()
(net(X) ** 2).mean().backward()
for h in handles:
    h.remove()
# The hooks fire in backward order, last block first
print('block output gradients:',
      ' '.join(f'{g:.3g}' for g in reversed(grad_norms)))
layer = net[3].body[0]  # block 3, first layer
print(f'weight {layer.weight.grad.norm().item():.3g}, '
      f'bias {layer.bias.grad.norm().item():.3g}')
```

```
block output gradients: 1.04 0.83 0.659 0.489 0.396 0.317 0.257 0.194
weight 12.9, bias 2.52
```

Reading backward from the eighth block to the first, the output gradient norm grows by

roughly $1.2\times$ to $1.4\times$ per block (five- to sixfold across the stack), mirroring the forward $1.22\times$ growth for the exact same reason: on the backward pass, each residual block adds the branch gradient back onto the skip-connection gradient. Isolated parameter gradient norms mean little on their own, but comparing them across depth or tracking them across training steps reveals vanishing or exploding dynamics immediately (see the exercises).

Remember what these debugging tools do and do not solve: they cannot force two different machines to match bit for bit, nor can a single run settle whether an experimental gain is real (for that, compare distributions across multiple seeds). Yet combined with resumable checkpoints (Section 5.6), you now have every piece needed to replay a failing job right up to the crash and attach a diagnostic hook to isolate the exact layer at fault.

The 8-billion-parameter language model that opened this chapter has not shrunk (it still weighs 32 GB in fp32), but nothing about how frameworks handle it is a black box anymore. Constructing architectures from configs, auditing parameters and buffers, initializing deep stacks, building custom layers and gradients, training in mixed precision, saving crash-safe checkpoints, managing GPU memory, and inspecting live runs all operate over the exact same tree of modules. Equipped with this engineering toolkit, we are ready to move on to architectures built around the structure of real-world data, starting with convolutional neural networks in Chapter 6.

5.8.4 Summary

Every random choice in training (initialization, shuffling, dropout, and data augmentation) traces back to pseudorandom number generators controlled by seeds, with private generator objects or explicit keys preventing unrelated code from perturbing a stream. Because floating-point addition is non-associative and parallel GPU atomic operations sum in arbitrary thread order, bit-for-bit repeatability also requires deterministic kernels and holds only on one machine and software stack, making bitwise determinism a debugging tool while scientific conclusions must hold across multiple seeds. Finally, forward and backward hooks (and their framework equivalents) let us inspect activation scales, gradient norms, and first-occurrence NaNs inside unmodified models.

5.8.5 Exercises

1. code **Residual variance budget.** Let $\mathbf{x}_k$, $\mathbf{b}_k$ and $\mathbf{h}_k$ denote the input, body output and output of block k, with each variance taken over all entries. On one batch, record for every `ResidualBlock` the variance of $\mathbf{x}_k$ with `register_forward_pre_hook` (the pre-hook receives the argument tuple, so unpack it), of $\mathbf{h}_k$ with a forward hook on the block, and of $\mathbf{b}_k$ with a forward hook on its `body`. The standard deviations in the table of Section 5.8.3.1 grow because each block adds its body's output to the stream. Measure how closely

$$\mathrm{Var}[\mathbf{h}_k] = \mathrm{Var}[\mathbf{x}_k] + \mathrm{Var}[\mathbf{b}_k] \tag{5.12}$$

   holds for every block, and name the term this equation omits.

2. code **Seeds and data loading.** ● Extend `train_once` to load its data through a `DataLoader` with `shuffle=True` and `num_workers=4`, keeping the single `torch.manual_seed(seed)` call at its top. The dataset adds noise to each example inside `__getitem__`, drawn from an `np.random.default_rng()` object created in the dataset's constructor. Run the code as a

standalone script, because worker processes started with `spawn`, the default on macOS and Windows, cannot import a dataset class defined in a notebook.

1. Predict whether two runs with the same seed agree on the shuffle order and on the noise, and whether the four workers draw the same noise. Justify each prediction from where that randomness is drawn, then verify all three.
2. Add the `seed_worker` of this section and a seeded `generator` to the loader, and verify that two runs now agree and that the workers draw different noise. Does an example receive the same noise in the second epoch as in the first? Explain the answer from when the loader draws its base seed and calls the initializer.

Adapted from the PyTorch documentation Multi-process data loading.

3. code **Divergence monitor.** ● Turn the finiteness check of Section 5.8.3.2 into a per-step training monitor. Rebuild `net` and train it with SGD at learning rate 1 on the mean squared error between its output and standard normal targets, using one fixed batch of 256 standard normal inputs. Report the first step at which a layer's output is non-finite, and the first such layer by depth. Check the loss at every step as well, and report whether it becomes non-finite before any layer output does.

4. code **Gradient norms.** ● Rebuild `net`, multiply the weight of every linear layer inside its residual blocks by 3, and use the mean squared output on a fixed batch as the loss.

   1. Record the norm of each block's output gradient as in this section. After the backward pass, compute each block's parameter-gradient norm, and plot both families against depth.
   2. Express the gradient of a linear layer's weight through the layer's input and its output gradient. Use this expression to explain why the two depth profiles differ, and state which family to monitor for an exploding backward signal.

Chapter 6

Convolutional Neural Networks

Every image classifier we have built so far has ignored the spatial geometry of images, flattening a two-dimensional pixel grid into a one-dimensional vector. The MLP in Chapter 4 treats a 28×28 Fashion-MNIST photo as an unordered list of 784 numbers: if we applied the same fixed random permutation to the pixels of every image and retrained, the MLP would reach the exact same accuracy—even though the scrambled images would look like pure static to human eyes. Humans recognize a sneaker by its outline and textures, which are patterns formed by *neighboring* pixels, and an outline carries the same meaning wherever in the frame it appears.

Neuroscience offered an early clue. In classic experiments on cats and monkeys, Hubel and Wiesel discovered that neurons in the early visual cortex respond to oriented edges and bars within localized patches of the visual field—each neuron's *receptive field* (Hubel and Wiesel 1959, 1962, 1968). Later, Field (1987) modeled these responses on natural images using a small bank of local filters swept across every image location (Figure 6.1).

Inspired by these findings, Fukushima's Neocognitron (Fukushima 1980) organized local detectors into hierarchical layers repeated across space, and LeCun and collaborators showed how to train such architectures end to end via backpropagation to recognize handwritten ZIP codes (LeCun et al. 1989, 1995). Deeper layers of trained vision networks even exhibit representational parallels with higher stages of the primate visual pathway (Kuzovkin et al. 2018), though biology served as architectural inspiration rather than a blueprint for silicon computation.

At the heart of these architectures lies the *convolutional layer*, which slides a small shared kernel of weights across every neighborhood of an image. Each output unit depends only on nearby pixels, and shifting the input simply shifts the output, building locality and translation equivariance directly into the network while slashing parameter counts by orders of magnitude. These *convolutional neural networks* (CNNs) drove the watershed breakthrough on ImageNet (Deng et al. 2009; Krizhevsky et al. 2012) and remain a cornerstone of computer vision alongside vision transformers (Chapter 11).

This chapter derives and builds CNNs from first principles in three steps. We begin by showing that enforcing locality and translation equivariance on a dense layer leaves convolution as the unique linear choice (Section 6.1), then implement two-dimensional cross-correlation, learn an edge-detecting kernel by gradient descent, recast convolution as matrix multiplication (*im2col*), and trace how stacked layers widen receptive fields (Section 6.2).

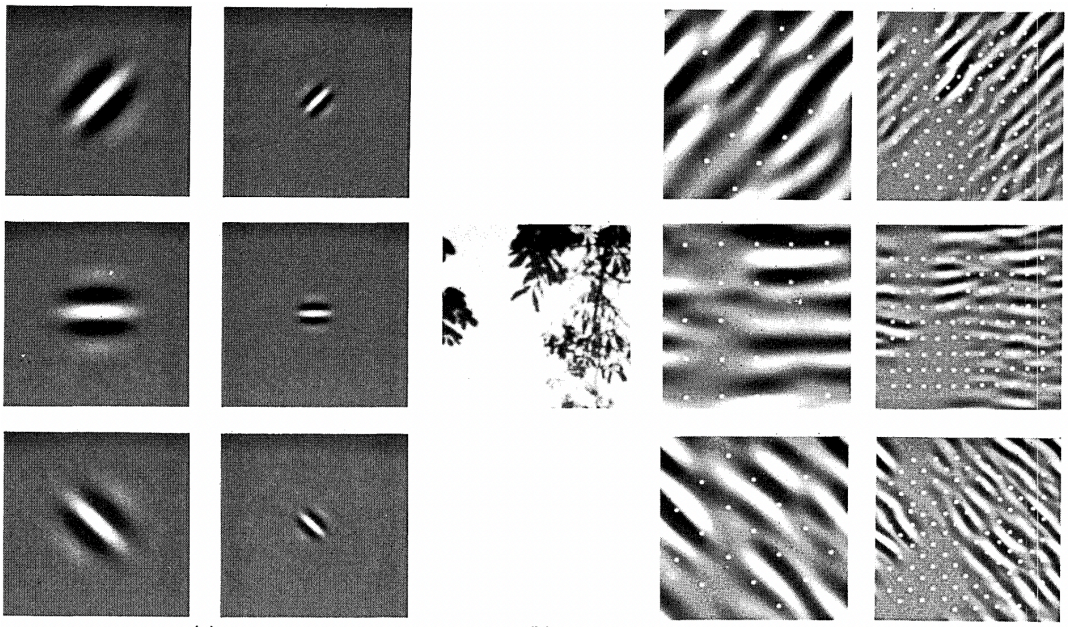


Figure 6.1: Six receptive-field filters and their responses to a natural image, adapted from Field (1987) . The left panel shows the filters, the middle panel the input image, and the right panel the filtered responses sampled at intervals proportional to each filter's size. Different filters respond to different local edge patterns.

Next, we control spatial resolution, field of view, and feature capacity through padding, stride, and dilation (Section 6.3), multi-channel filter banks and 1×1 bottlenecks (Section 6.4), and local and global pooling (Section 6.5). Finally, we assemble these pieces into LeNet (Section 6.6), compare it against our MLP on Fashion-MNIST, inspect its learned first-layer filters, and test both models on pixel-shuffled images to isolate the contribution of spatial inductive bias. Modern deep convolutional architectures follow in Chapter 7.

Resources and Further Reading

The resources below accompany the sections of this chapter; each names the section it goes with. All are freely accessible online except where noted.

Books

- Deep Learning (Goodfellow, Bengio & Courville): free HTML; Chapter 9 develops sparse interactions, parameter sharing, and equivariance in detail (Section 6.1, Section 6.5).
- Understanding Deep Learning (Simon J. D. Prince): free PDF; Chapter 10 covers convolutions, stride, dilation, and receptive fields with clear geometric figures (Section 6.2, Section 6.3).
- Neural Networks and Deep Learning (Michael Nielsen): free online; Chapter 6 introduces convolutional layers, shared weights, and pooling from first principles (Section 6.2, Section 6.5).

Courses and video lectures

- Stanford CS231n: Deep Learning for Computer Vision: free notes; the "Convolutional Networks" module (layer arithmetic, parameter counting, layer patterns) is a widely used online companion to this chapter (Section 6.3, Section 6.4).
- Michigan EECS 498-007: Deep Learning for Computer Vision (Justin Johnson): free lecture videos working through convolution and pooling arithmetic at this chapter's pace (Section 6.2, Section 6.5).

Tutorials, notes, and interactive

- A guide to convolution arithmetic for deep learning (Dumoulin & Visin, 2016): free, with companion animations; visual reference for padding, stride, dilation, and transposed convolutions (Section 6.3).
- CNN Explainer (Wang et al., Polo Club): interactive browser convnet for inspecting activations, kernels, and receptive fields (Section 6.4, Section 6.6).
- Computing Receptive Fields of Convolutional Neural Networks (Araujo, Norris & Sim, 2019, *Distill*): derives the closed-form receptive-field arithmetic introduced in Section 6.2, including strided and multi-path cases.
- Conv Nets: A Modular Perspective (Chris Olah): short classic building intuition for convolution as structured weight sharing (Section 6.1).
- Image Kernels Explained Visually (Victor Powell): interactive tool for sliding hand-designed kernels over an image, complementing Section 6.2.
- Feature Visualization (Olah, Mordvintsev & Schubert, 2017, *Distill*): visual exploration of what convnet units respond to, accompanying Section 6.6.

Foundational papers

- Gradient-Based Learning Applied to Document Recognition (LeCun, Bottou, Bengio & Haffner, 1998, *Proc. IEEE*): free PDF; the LeNet-5 paper behind Section 6.6.
- Backpropagation Applied to Handwritten Zip Code Recognition (LeCun et al., 1989, *Neural Computation*): the first convnet trained end to end with backpropagation (paywalled) (Section 6.6).
- Neocognitron (Fukushima, 1980, *Biological Cybernetics*): the pre-backpropagation ancestor of alternating convolution and pooling stages (paywalled) (Section 6.5).
- Receptive fields, binocular interaction and functional architecture in the cat's visual cortex (Hubel & Wiesel, 1962, *J. Physiology*): free (PMC); the biological root of local receptive fields and hierarchical feature detection (Section 6.1).

6.1 From Fully Connected Layers to Convolutions

The multilayer perceptrons of Chapter 4 flatten every image into an unordered vector of numbers: appropriate for tabular data, but wasteful for images because flattening discards which pixels sit next to one another. On a one-megapixel (1000×1000) grayscale photograph, even a modest 1,000-unit hidden layer demands $10^6 \times 10^3 = 10^9$ weights (4 GB in fp32), and a full-resolution 1000×1000 hidden map demands 10^{12} weights. Yet natural images are nothing like arbitrary million-dimensional vectors: neighboring pixels are strongly correlated (*locality*), and a visual feature such as a whisker carries the same meaning wherever it appears in the frame (*translation equivariance*).

What happens if we impose those two geometric principles directly on a dense layer mapping a 2D pixel grid to a 2D hidden grid? Translation equivariance alone slashes 10^{12} weights down to roughly 4×10^6, and restricting each hidden unit to a small local neighborhood cuts them further to a few hundred. What survives both constraints is a *convolutional*

layer.

6.1.1 Equivariance and Invariance

The children's game "Where's Waldo" (Figure 6.2) captures the exact challenge of visual perception. Each busy scene teems with distractions, with Waldo hiding somewhere unexpected, yet *what Waldo looks like* does not depend on *where he stands*. We can therefore sweep a single Waldo detector across the image, scoring every patch by how strongly it matches Waldo using the exact same rule at every position. Modern object detection (Girshick et al. 2014) and semantic segmentation (Long et al. 2015) pipelines build directly on this idea.

Figure 6.2: Can you find Waldo (image courtesy of William Murphy (Infomatique))?

This intuition points to two guiding principles for the early layers of a vision network:

1. They should respond identically to the same local patch regardless of where it appears, so shifting the input shifts the output by the same amount.
2. They should attend only to small local neighborhoods without worrying about distant parts of the image—the *locality* principle.

Subsequent layers can then compose these local responses into progressively larger, more abstract patterns, culminating in a whole-image prediction.

To formalize the first principle, let T_v denote the shift operator that translates an image by an offset $v = (v_1, v_2)$, moving pixel (i, j) to $(i + v_1, j + v_2)$ so that $[T_v\mathbf{X}]_{i,j} = [\mathbf{X}]_{i-v_1,j-v_2}$. An image-to-image mapping f is *translation equivariant* if shifting its input shifts its output by the exact same vector, whereas an image-to-label mapping is *translation invariant* if shifting its input leaves its output completely unchanged:

$$\begin{aligned} f(T_v\mathbf{X}) &= T_v f(\mathbf{X}) \quad \text{(equivariance)}, \\ f(T_v\mathbf{X}) &= f(\mathbf{X}) \quad\;\; \text{(invariance)}. \end{aligned} \tag{6.1}$$

Figure 6.3 illustrates translation equivariance as a commuting diagram. Here f matches a small template against every window of the image—the exact operation we are about to derive. Shifting first and matching second produces the identical map as matching first and shifting second.

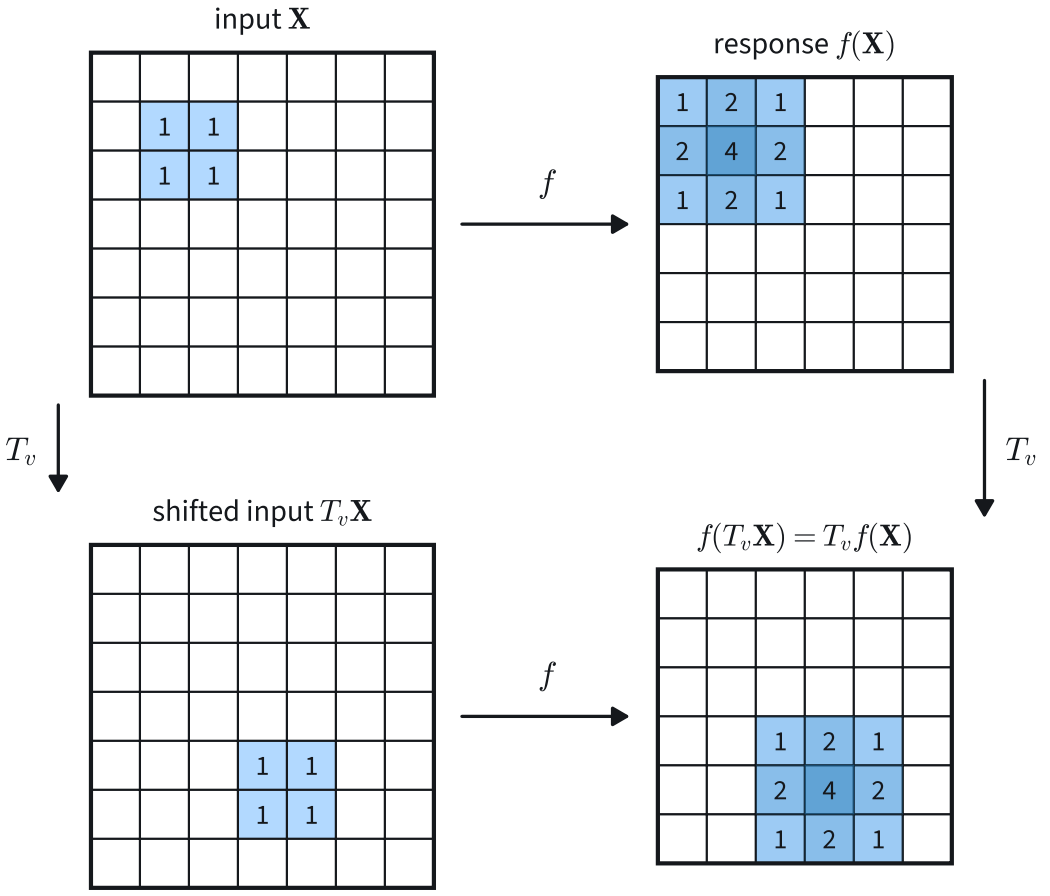


Figure 6.3: Translation equivariance. The input holds a 2×2 patch of ones, and f compares every 2×2 window with that patch, so the response is 4 where a window covers the whole patch. Shifting the input by $v = (3, 2)$ and then applying f (bottom row) gives the same map as applying f and then shifting the response (right column).

Notice the clean division of labor: *equivariance* preserves *where* a pattern appears, whereas *invariance* discards location. A Waldo detector should be equivariant so its score map tracks Waldo across the scene; answering the yes-or-no question "is Waldo in this picture?" should be invariant, achieved at the end by aggregating the equivariant map across space (for instance, taking its maximum).

6.1.2 Constraints on the Fully Connected Layer

Both constraints become crystal clear in one dimension, where we can draw the entire weight matrix. Consider an eight-sample signal $x_0, \dots, x_7$ mapped by a fully connected layer to eight hidden units, $h_i = u_i + \sum_k W_{i,k} x_k$. Its 8×8 weight matrix has 64 independent entries, shown in panel (a) of Figure 6.4.

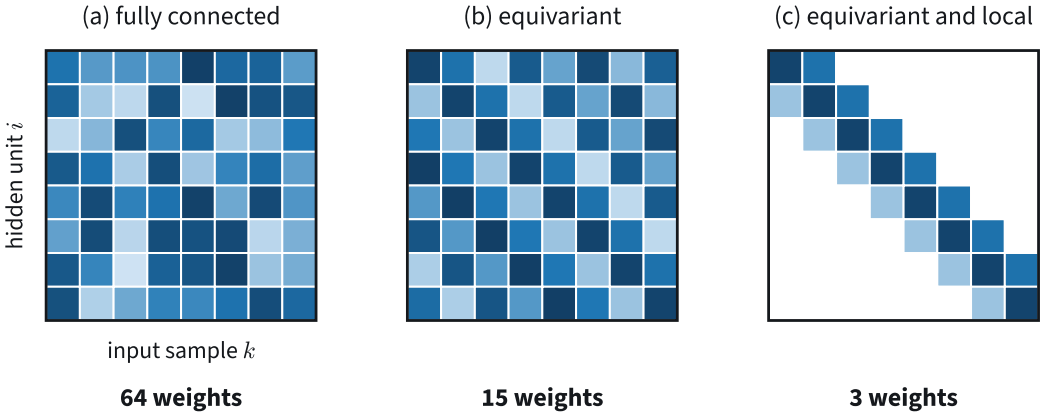


Figure 6.4: Weights of a linear layer that maps a one-dimensional signal of 8 samples (columns) to 8 hidden units (rows). Equal colors are equal weights; white is zero. (a) Fully connected: every weight is free. (b) Translation equivariant: a weight depends only on the offset between input and output, so the matrix is constant along each diagonal. (c) Equivariant and local: only the offsets -1, 0, and 1 have nonzero weights.

What does translation equivariance impose on $\mathbf{W}$? Ignoring biases for a moment (our proof below shows all biases must coincide), feed in a unit impulse with a single 1 at sample k: the layer outputs column k of $\mathbf{W}$. Shift that impulse to $k+1$, and the layer outputs column $k+1$. Equivariance insists that column $k+1$ must equal column k shifted down by one row, i.e., $W_{i+1,k+1} = W_{i,k}$. Consequently, $\mathbf{W}$ must be constant along every diagonal (a *Toeplitz* matrix), as in panel (b). Each diagonal corresponds to a single relative offset $a = k - i$, collapsing 64 free weights down to just 15 (offsets -7 through 7).

Locality prunes almost all of those remaining 15 weights. Restricting each hidden unit to look only at neighbors within distance 1 zeroes out every offset outside $\{-1, 0, 1\}$, leaving just 3 shared weights repeated across every row in panel (c). The layer now sweeps one tiny 3-weight template across the entire signal. That is the entire derivation in miniature; moving to 2D images simply adds a second spatial index.

For images, both the input $\mathbf{X}$ and the hidden representation $\mathbf{H}$ are two-dimensional grids, with input pixel $[\mathbf{X}]_{k,l}$ at (k, l) and hidden unit $[\mathbf{H}]_{i,j}$ at (i, j). Connecting every hidden unit to every pixel turns the weights into a fourth-order tensor W alongside a bias matrix $\mathbf{U}$:

$$[\mathbf{H}]_{i,j} = [\mathbf{U}]_{i,j} + \sum_k \sum_l [\mathsf{W}]_{i,j,k,l}[\mathbf{X}]_{k,l}.$$

Because both constraints depend on where a pixel sits *relative* to hidden unit (i, j), we re-index by offsets $a = k - i$ and $b = l - j$, defining $[\mathsf{V}]_{i,j,a,b} = [\mathsf{W}]_{i,j,i+a,j+b}$:

$$[\mathbf{H}]_{i,j} = [\mathbf{U}]_{i,j} + \sum_a \sum_b [\mathsf{V}]_{i,j,a,b}[\mathbf{X}]_{i+a,j+b}. \tag{6.2}$$

Here a and b span both positive and negative offsets. This re-indexing is purely cosmetic: we are simply reading W along its diagonals without reducing parameters yet. On a 1000×1000 image mapped to a 1000×1000 hidden grid, this single layer still demands $10^6 \times 10^6 = 10^{12}$ weights.

6.1.2.1 Translation Equivariance

Just as in one dimension, a layer of the form Equation 6.2 is translation equivariant if and only if its weights and bias are independent of location (i, j). Checking that shared weights yield equivariance takes one line; the converse is what makes convolution unavoidable, since equivariance leaves no other linear option.

To handle boundaries cleanly (where a shift pushes pixels off the edge of a finite frame), we embed the image on an infinite grid padded with zeros outside its support so no information ever leaves the grid. On finite images, equivariance holds strictly away from borders, a detail we address in Section 6.3.

Proposition. Let the input (with finitely many nonzero pixels) and the hidden representation live on the infinite grid. The layer Equation 6.2 is translation equivariant for every shift v if and only if there exist a scalar u and a matrix $\mathbf{K}$ such that $[\mathbf{U}]_{i,j} = u$ and $[\mathsf{V}]_{i,j,a,b} = [\mathbf{K}]_{a,b}$ for all i, j, a, b.

Proof. If neither weights nor bias depend on (i, j), then shifting the input by v directly shifts the output by v:

$$\begin{aligned}[f(T_v\mathbf{X})]_{i,j} &= u + \sum_a \sum_b [\mathbf{K}]_{a,b}[\mathbf{X}]_{i-v_1+a,j-v_2+b} \\ &= [f(\mathbf{X})]_{i-v_1,j-v_2} = [T_v f(\mathbf{X})]_{i,j}.\end{aligned}$$

Conversely, assume the layer is equivariant. The all-zero image is invariant under every shift and maps to $\mathbf{U}$, forcing $\mathbf{U} = T_v\mathbf{U}$ for all v, so the bias must be a constant u. Subtracting u from both sides shows that the linear part alone is also equivariant. Feed in an image with a single 1 at (k, l) and zeros elsewhere: its response at (i, j) is $[\mathsf{W}]_{i,j,k,l}$. Shifting that impulse by v moves it to $(k+v_1, l+v_2)$, and equivariance requires $[\mathsf{W}]_{i,j,k+v_1,l+v_2} = [\mathsf{W}]_{i-v_1,j-v_2,k,l}$. Setting $(k, l) = (a, b)$ and $v = (i, j)$ yields $[\mathsf{V}]_{i,j,a,b} = [\mathsf{W}]_{i,j,i+a,j+b} = [\mathsf{W}]_{0,0,a,b}$, which is independent of (i, j). ■

In short, probing the layer with a single bright pixel forces every location to share the exact same weights, collapsing the layer to

$$[\mathbf{H}]_{i,j} = u + \sum_a \sum_b [\mathbf{K}]_{a,b}[\mathbf{X}]_{i+a,j+b}. \tag{6.3}$$

Every hidden unit now applies the same pattern $\mathbf{K}$ centered at its own coordinates. On a 1000×1000 zero-padded image, only offsets $a, b \in [-999, 999]$ ever touch a nonzero pixel, slashing 10^{12} weights down to $1999^2 \approx 4 \times 10^6$.

6.1.2.2 Locality

Four million weights for a single filter is still far too many, and most of those weights link a hidden unit to pixels clear across the image. Enforcing *locality* (requiring $[\mathbf{H}]_{i,j}$ to depend only on pixels within distance Δ along each axis) sets $[\mathbf{K}]_{a,b} = 0$ whenever $|a| > \Delta$ or $|b| > \Delta$, giving

$$[\mathbf{H}]_{i,j} = u + \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} [\mathbf{K}]_{a,b}[\mathbf{X}]_{i+a,j+b}. \tag{6.4}$$

Each hidden unit now computes a weighted sum over a $(2\Delta+1) \times (2\Delta+1)$ local window centered at (i, j), using just $(2\Delta+1)^2$ shared weights and one bias: 9 weights for $\Delta = 1$ (a 3×3 window), or 121 for $\Delta = 5$. Because Δ is typically well below 10, a few dozen or hundred weights replace 4×10^6 while preserving the full spatial resolution of the hidden map.

Equation Equation 6.4 defines a *convolutional layer*, the building block of *convolutional neural networks* (CNNs). The shared matrix $\mathbf{K}$ is called the *convolution kernel* or *filter*. Early pioneers of spatial and temporal weight sharing included time-delay neural networks for speech recognition (Waibel et al. 1989) and the shift-invariant networks of Zhang et al. (1988) .

Crucially, the parameter count shrank from 10^{12} to $(2\Delta+1)^2$ by restricting the hypothesis class to local, translation-equivariant mappings. This built-in *inductive bias* (Section 4.5) dramatically improves sample efficiency whenever features are local and position-independent (such as edges and textures), though a single layer cannot distinguish identical patches by absolute position alone (e.g., an eye near the top of a passport photo versus a nostril lower down). Stacking multiple layers with nonlinearities allows deeper units to integrate context across progressively wider regions.

6.1.3 Convolution and Cross-Correlation

Strictly speaking, the name "convolution" in deep learning is a slight misnomer. In mathematics, the discrete *convolution* of two 1D functions is $(f * g)(i) = \sum_a f(a)g(i-a)$ (with an integral on $\mathbb{R}^d$), measuring the overlap between f and a *flipped*, shifted copy of g. In two dimensions,

$$(f * g)(i, j) = \sum_a \sum_b f(a, b)g(i-a, j-b). \tag{6.5}$$

Compare this with Equation 6.4, which indexes $\mathbf{X}$ at $(i+a, j+b)$ rather than $(i-a, j-b)$. Substituting $-a$ for a and $-b$ for b reveals that Equation 6.4 is the mathematical convolution of $\mathbf{X}$ with the flipped kernel $[\mathbf{K}']_{a,b} = [\mathbf{K}]_{-a,-b}$:

$$\sum_a \sum_b [\mathbf{K}]_{a,b}[\mathbf{X}]_{i+a,j+b} = \sum_a \sum_b [\mathbf{K}']_{a,b}[\mathbf{X}]_{i-a,j-b} = (\mathbf{K}' * \mathbf{X})(i,j). \tag{6.6}$$

Flipping reverses the kernel along both axes, so that its top-left weight becomes the bottom-right one (Figure 6.5).

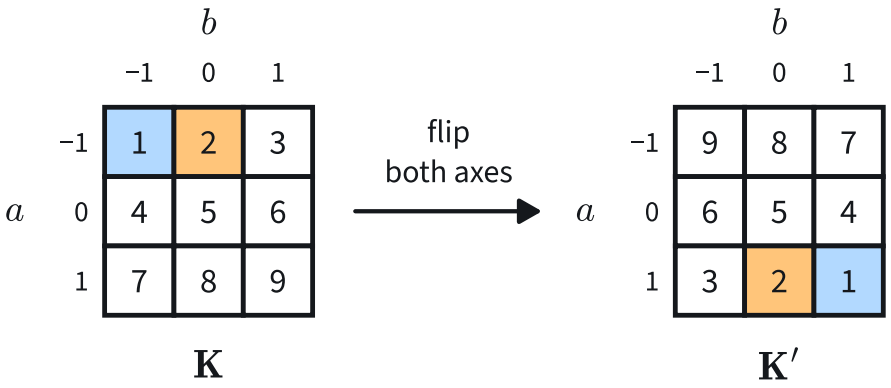


Figure 6.5: A 3×3 kernel $\mathbf{K}$ and its flipped copy $\mathbf{K}'$, with rows and columns indexed by the offsets a and b. The flip reverses both axes: the two shaded weights move from offsets $(-1,-1)$ and $(-1,0)$ to $(1,1)$ and $(1,0)$. Cross-correlation with $\mathbf{K}$ is convolution with $\mathbf{K}'$.

The unflipped operation is properly called *cross-correlation*. Because CNN kernels are learned from data, the distinction has no effect on expressive power: a layer that cross-correlates with $\mathbf{K}$ learns the exact same family of functions as one that convolves with $\mathbf{K}'$. Deep learning frameworks compute cross-correlation and call it convolution, and we follow that standard convention throughout this book.

6.1.4 Channels

So far we treated both inputs and hidden maps as single grayscale grids. Real color photographs have three intensity values per pixel (red, green, and blue), so an image is a third-order tensor X of shape $3 \times$ height $\times$ width with entries $[\mathsf{X}]_{c,i,j}$. The leading *channel* axis attaches a feature vector to every spatial pixel, and a kernel sums over both the local spatial window and all input channels.

Just as importantly, a single kernel produces only a single response map (like the Waldo score map in Figure 6.6), capturing just one visual pattern. Rich visual representations require detecting many patterns in parallel (oriented edges, corners, color contrasts, textures), so the hidden representation H also becomes a third-order tensor stacked along an *output channel* axis. Each output channel o has its own learned kernel slice and bias u_o, turning K into a fourth-order tensor:

$$[\mathsf{H}]_{o,i,j} = u_o + \sum_c \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} [\mathsf{K}]_{o,c,a,b}[\mathsf{X}]_{c,i+a,j+b}. \tag{6.7}$$

Here c indexes input channels and o indexes output channels; we keep this output-before-input channel convention throughout the chapter. Because each output channel maps where a particular learned feature occurs across the grid, hidden channels are also called *feature maps*, and the entire stack feeds into the next layer. With c_{i} input channels and c_{o} output channels, the layer requires $(2\Delta+1)^2 c_{\mathrm{i}} c_{\mathrm{o}}$ weights and c_{o} biases. A 3×3 layer mapping an RGB image to 64 feature maps therefore has just $9 \cdot 3 \cdot 64 + 64 = 1{,}792$ parameters, completely independent of the image's resolution.

Figure 6.6: A Waldo detector: the response of a kernel that matches Waldo peaks at his location (image courtesy of William Murphy (Infomatique)).

Nothing in this derivation is tied to two dimensions: the exact same logic yields 1D convolutions sliding across audio samples or text tokens, and 3D convolutions sliding across video frames or volumetric medical scans.

Equation Equation 6.7 is the foundation for the rest of this chapter. Because we derived it on an idealized infinite grid, three practical issues remain: handling finite image borders, expanding how much of the image deeper units can see, and aggregating spatial feature maps into a single image-level decision (like whether Waldo appears *anywhere* in the scene). First, however, we bring Equation 6.4 to life in code.

6.1.5 Summary

Flattening an image into an unstructured vector makes dense layers intractably large (10^{12} weights to map a megapixel image to an equal-sized hidden grid). Two natural assumptions about images collapse that count to a few hundred: *translation equivariance* forces every spatial position to share one kernel (as proved on the infinite grid), and *locality* restricts that kernel to a small window. The resulting layer computes cross-correlation (universally called convolution in deep learning), and stacking multiple input and output channels lets a single layer detect many visual patterns simultaneously while keeping parameter counts independent of image size. Those gains stem directly from inductive bias: a convolutional layer trades away arbitrary position-dependent mappings for strong generalization on spatially structured data.

6.1.6 Exercises

1. **Weight sharing on a ring.** Repeat the argument of Figure 6.4 for a signal of four samples on a periodic grid, where the sample after the last one is the first one again, so that a shift loses nothing.
   1. Feed the layer a single 1 at each sample in turn and write down the 4×4 weight matrix of a translation-equivariant layer. How many free weights does it have? Compare with the $2 \cdot 4 - 1 = 7$ offsets of four samples surrounded by zeros.
   2. Impose locality with offsets -1, 0, and 1. Which entries of the matrix are nonzero, and which of them have no counterpart in panel (c) of the figure?
2. **Commutativity of convolution.**
   1. Prove that the convolution of Equation 6.5 is commutative, i.e., $f * g = g * f$.
   2. Define the cross-correlation $(f \star g)(i,j) = \sum_a \sum_b f(i+a, j+b)\, g(a,b)$, the operation of Equation 6.4 with input f and kernel g. Express $(g \star f)(i,j)$ through values of $f \star g$, and decide from this expression whether cross-correlation is commutative.
3. **Convolutions for audio.** Audio data is often represented as a one-dimensional sequence of samples.
   1. Derive the one-dimensional analog of Equation 6.4.
   2. Name one audio task for which a layer that is local and translation equivariant in time is a good fit, and one for which it is not. Justify each choice.
   3. A spectrogram represents a recording as a two-dimensional array whose axes are time and frequency, with frequency on a linear scale in hertz. For each axis, state whether translation equivariance is a reasonable assumption, and justify your answer from what a shift along that axis does to the sound. Then answer the same question for a frequency axis on a logarithmic scale.
4. **Equivariance versus invariance.** A convolutional classifier computes feature maps $h(\mathsf{X})$ with stacked layers of Equation 6.7 and nonlinearities in between, and then a prediction $g(h(\mathsf{X}))$ with a head g.
   1. Assume a periodic grid, which wraps around so that the column after the last one is the first one again; on it, h is translation equivariant by the argument of the proposition. Let g average each channel of $h(\mathsf{X})$ over all locations. Using Equation 6.1, show that the prediction is translation invariant.
   2. An image shows a single digit in one of the nine cells of a 3×3 grid, and the task is to name the cell. Explain why the classifier of the first part cannot solve this task. Describe a head g that solves it from the same feature maps $h(\mathsf{X})$.
5. **Rotation equivariance.** ● The proposition of this section derived weight sharing from equivariance to shifts. Ask the same of rotations. On the infinite grid of the proposition, let R rotate an image by 90° about the origin, $[R\mathbf{X}]_{i,j} = [\mathbf{X}]_{-j,i}$.
   1. Show that the convolutional layer Equation 6.4 satisfies $f(R\mathbf{X}) = Rf(\mathbf{X})$ for every image $\mathbf{X}$ if and only if $[\mathbf{K}]_{a,b} = [\mathbf{K}]_{-b,a}$ for all offsets (a,b).
   2. For $\Delta = 1$, how many free weights does such a kernel have? Show that its response to an image does not change when the image is rotated by 90° about the pixel being evaluated, so that it cannot tell a vertical edge from a horizontal one.
   3. Instead of constraining one kernel, use four output channels whose kernels are copies of one unconstrained kernel $\mathbf{K}$, rotated by 0°, 90°, 180°, and 270°. Show that rotating the input rotates every output map and permutes the four channels cyclically. What does this layer keep that the constrained kernel of the first part loses?

6.2 Convolutions for Images

In Section 6.1, we derived the convolutional layer on paper from two geometric principles: translation equivariance and locality. Now we see what that formula actually computes on an image, and whether gradient descent can discover useful kernels automatically from data.

Consider locating vertical edges where a bright region meets a dark one. Subtracting each pixel's right-hand neighbor ($[\mathbf{X}]_{i,j} - [\mathbf{X}]_{i,j+1}$) yields zero inside uniform regions, $+1$ where white transitions to black, and -1 where black returns to white. Sweeping the two-weight detector $(1, -1)$ across every valid image location produces an *edge map*. In this section, we implement two-dimensional cross-correlation, test our hand-crafted edge detector, let gradient descent *learn* $(1, -1)$ directly from data, show how a convolution becomes a single matrix multiplication (*im2col*), and derive how stacking layers widens a unit's receptive field.

6.2.1 Cross-Correlation

6.2.1.1 The Operation

Leaving channels aside for the moment, let the input $\mathbf{X}$ be an $n_\mathrm{h} \times n_\mathrm{w}$ matrix and the kernel $\mathbf{K}$ be a $k_\mathrm{h} \times k_\mathrm{w}$ matrix. The kernel defines a sliding *window* of shape $k_\mathrm{h} \times k_\mathrm{w}$. Placing the window's upper-left corner at (i, j), we multiply the covered input patch elementwise with $\mathbf{K}$ and sum the products to produce the output entry at (i, j):

$$[\mathbf{Y}]_{i,j} = \sum_{a=0}^{k_\mathrm{h}-1} \sum_{b=0}^{k_\mathrm{w}-1} [\mathbf{K}]_{a,b} \, [\mathbf{X}]_{i+a,j+b}. \tag{6.8}$$

This is the *two-dimensional cross-correlation* of $\mathbf{X}$ with $\mathbf{K}$, written $\mathbf{Y} = \mathbf{X} \star \mathbf{K}$ with zero-based indices matching code. Whereas Section 6.1 centered offsets symmetrically from $-\Delta$ to Δ, anchoring at the upper-left corner differs only by a fixed shift of Δ while naturally handling even-sized kernels (such as a 2×2 kernel or our 1×2 edge detector) that have no center pixel.

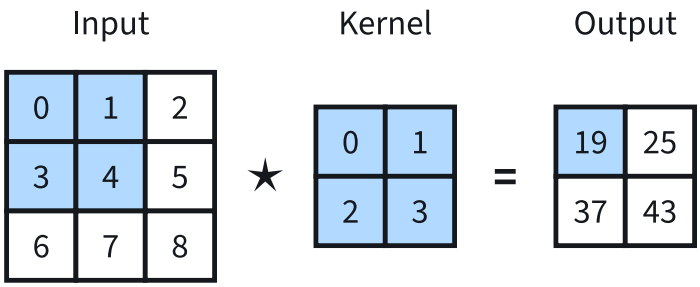


Figure 6.7: Two-dimensional cross-correlation. The shaded portions are the first output element and the input and kernel elements used to compute it: $0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19$.

In Figure 6.7, we slide a 2×2 kernel across a 3×3 input one pixel at a time, left to right and top to bottom. The window fits cleanly at four positions, yielding the 2×2 output:

$$0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19,$$
$$1 \times 0 + 2 \times 1 + 4 \times 2 + 5 \times 3 = 25,$$
$$3 \times 0 + 4 \times 1 + 6 \times 2 + 7 \times 3 = 37,$$
$$4 \times 0 + 5 \times 1 + 7 \times 2 + 8 \times 3 = 43.$$

Notice that the output shrinks slightly along each axis because the kernel must stay wholly inside the input. Fitting a width-k window into length n leaves $n - k + 1$ valid positions, so the output shape is

$$(n_{\mathrm{h}} - k_{\mathrm{h}} + 1) \times (n_{\mathrm{w}} - k_{\mathrm{w}} + 1). \tag{6.9}$$

Thus a 3×3 input with a 2×2 kernel yields 2×2, a 6×8 image with a 1×2 kernel yields 6×7, and a 28×28 Fashion-MNIST image with a 5×5 kernel yields 24×24. Whenever we want to preserve spatial dimensions, we surround the input with a border of zeros (*padding*), which we examine alongside strides and dilation in Section 6.3.

6.2.1.2 Implementation

We implement Equation 6.8 directly in `corr2d` using a pair of loops over every output coordinate.

```
def corr2d(X, K):
    """Compute 2D cross-correlation."""
    h, w = K.shape
    Y = d2l.zeros((X.shape[0] - h + 1, X.shape[1] - w + 1))
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            Y[i, j] = d2l.reduce_sum((X[i: i + h, j: j + w] * K))
    return Y
```

To verify `corr2d` against our hand calculation in Figure 6.7, we construct the 3×3 input tensor `X` and 2×2 kernel `K` and evaluate `corr2d(X, K)`.

```
X = d2l.tensor([[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]])
K = d2l.tensor([[0.0, 1.0], [2.0, 3.0]])
corr2d(X, K)
```

```
tensor([[19., 25.],
        [37., 43.]])
```

The output reproduces Figure 6.7 entry for entry (and as shown in Section 6.1, strict mathematical convolution with a flipped kernel $\mathbf{K}'$ learns the exact same family of layers, so we follow standard deep learning terminology and call cross-correlation "convolution").

6.2.2 Convolutional Layers

6.2.2.1 The Layer

A two-dimensional convolutional layer cross-correlates its input with a learned kernel and adds a scalar bias. Just like our MLP weights in Chapter 4, we initialize the kernel from

$\mathcal{N}(0, 0.01^2)$ and the bias to zero. Below, `Conv2D` registers both parameters and invokes `corr2d` in its forward pass; a layer with a kernel of height h and width w is called an $h \times w$ convolutional layer.

```
class Conv2D(nn.Module):
    def __init__(self, kernel_size):
        super().__init__()
        self.weight = nn.Parameter(0.01 * torch.randn(kernel_size))
        self.bias = nn.Parameter(torch.zeros(1))

    def forward(self, x):
        return corr2d(x, self.weight) + self.bias
```

Unlike a dense layer, a convolutional layer is not locked to a fixed input resolution because its parameters depend only on the kernel size. The exact same 2×2 layer (5 parameters total) processes both our 3×3 tensor `X` and a 6×8 image, returning outputs of shapes 2×2 and 5×7:

```
layer = Conv2D((2, 2))
layer(X).shape, layer(d2l.ones((6, 8))).shape
```

```
(torch.Size([2, 2]), torch.Size([5, 7]))
```

6.2.2.2 Edge Detection

We now test the vertical edge detector from the introduction on a 6×8 synthetic image whose middle four columns are black (`0`) and outer columns are white (`1`).

```
X = d2l.ones((6, 8))
X[:, 2:6] = 0
X
```

```
tensor([[1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.]])
```

We define the 1×2 kernel

$$\mathbf{K} = \begin{bmatrix} 1 & -1 \end{bmatrix}. \tag{6.10}$$

At every position (i, j), this computes the horizontal *finite difference* $[\mathbf{X}]_{i,j} - [\mathbf{X}]_{i,j+1}$, a discrete approximation (up to sign) of the horizontal derivative of brightness, which vanishes across flat regions and spikes wherever intensity jumps.

```
K = d2l.tensor([[1.0, -1.0]])
```

Cross-correlating the 6×8 stripe image `X` with the 1×2 horizontal-difference kernel `K` via `corr2d(X, K)` produces the 6×7 edge map `Y`:

```
Y = corr2d(X, K)
Y
```

```
tensor([[ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.]])
```

Sure enough, `Y` fires `+1` along the white-to-black boundary, `-1` along the black-to-white boundary, and `0` everywhere else. Applying the exact same kernel to the transposed image (turning the vertical stripe into a horizontal one) yields all zeros:

```
corr2d(d2l.transpose(X), K)
```

```
tensor([[0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.]])
```

Clearly `[1, -1]` responds strictly to vertical edges. Detecting horizontal edges, diagonals, corners, or textures requires different kernels altogether.

6.2.2.3 Learning a Kernel

Hand-crafting `[1, -1]` is easy when we know the exact feature we want, yet no engineer can specify thousands of intermediate filters across a deep network by hand. Can gradient descent recover the kernel `[1, -1]` from the input–output pair `(X, Y)` alone? We test this using each framework's built-in 2D convolutional layer (with bias omitted and weights initialized from $\mathcal{N}(0, 0.01^2)$), minimizing squared error summed across all 42 output entries over 10 gradient steps at learning rate 0.03. Because built-in layers expect 4D tensors including batch and channel axes (Section 6.4), we reshape `X` and `Y` accordingly.

```
conv2d = nn.Conv2d(1, 1, kernel_size=(1, 2), bias=False)
nn.init.normal_(conv2d.weight, std=0.01)
# Layers take (example, channel, height, width): one example, one channel
X = X.reshape((1, 1, 6, 8))
Y = Y.reshape((1, 1, 6, 7))
lr = 3e-2  # Learning rate

for i in range(10):
    l = ((conv2d(X) - Y) ** 2).sum()
    conv2d.zero_grad()
```

```
    l.backward()
    with torch.no_grad():
        conv2d.weight -= lr * conv2d.weight.grad
    if i % 3 == 0:
        print(f'step {i}, loss {l:.3f}')
```

```
step 0, loss 12.216
step 3, loss 0.840
step 6, loss 0.058
step 9, loss 0.004
```

Initially the loss sits near 12 (the near-zero random kernel misses all 12 ± 1 edge pixels by roughly 1), and within just nine steps it drops below 0.01. Inspecting the learned kernel confirms what it found:

```
d2l.reshape(conv2d.weight.data, (1, 2))
```

```
tensor([[ 0.9891, -0.9876]])
```

Both entries land within roughly 0.02 of `[1, -1]`. Nothing in this loop knew anything about edges: given a loss signal, gradient descent discovered the filter on its own. Here we supervised the single layer directly with `Y`; in a deep CNN, backpropagation learns every layer's kernels end to end from the final task loss alone.

6.2.3 Convolution as Matrix Multiplication

While the nested loops in `corr2d` make the math transparent, visiting one output pixel at a time in Python is far too slow for real workloads. A classic remedy, and still one of the algorithms that libraries choose from, is to recast cross-correlation as a single large matrix multiplication, the exact operation GPUs execute at peak speed. Notice that every output entry in Figure 6.7 is simply the dot product of the flattened $k_\mathrm{h}k_\mathrm{w}$-element kernel with the corresponding flattened input window. Extracting all $(n_\mathrm{h} - k_\mathrm{h} + 1)(n_\mathrm{w} - k_\mathrm{w} + 1)$ windows and stacking them as rows yields a patch matrix; multiplying it by the flattened kernel computes the entire cross-correlation in one shot. This standard layout transformation is called *im2col* ("image to columns", or rows here). Figure 6.8 carries it out on the example of Figure 6.7.

The trade-off is extra workspace memory, since each interior pixel is duplicated into up to $k_\mathrm{h}k_\mathrm{w}$ patches. Below we verify on Figure 6.7 that this single matrix multiply matches `corr2d` exactly:

```
X = d2l.tensor([[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]])
K = d2l.tensor([[0.0, 1.0], [2.0, 3.0]])
h, w = K.shape
h_o, w_o = X.shape[0] - h + 1, X.shape[1] - w + 1
patches = d2l.stack([d2l.reshape(X[i:i + h, j:j + w], (-1,))
                     for i in range(h_o) for j in range(w_o)])
Y_mat = d2l.reshape(d2l.matmul(patches, d2l.reshape(K, (-1, 1))), (h_o, w_o))
Y_mat, d2l.reduce_sum(d2l.abs(Y_mat - corr2d(X, K)))
```

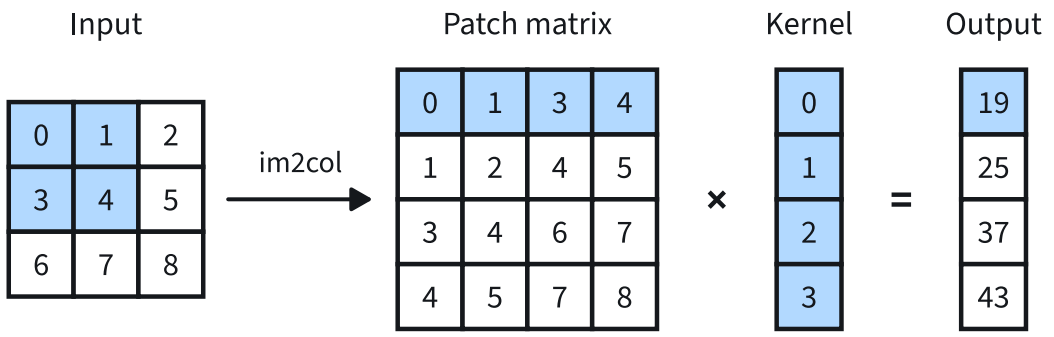


Figure 6.8: The cross-correlation of Figure 6.7 as one matrix product. Each of the four 2×2 windows of the input becomes a row of the patch matrix, and multiplying by the flattened kernel gives the flattened output. The shaded cells follow the first window to its row and to the output 19. The center pixel 4 lies in all four windows, so it appears in every row.

```
(tensor([[19., 25.],
         [37., 43.]]),
 tensor(0.))
```

The discrepancy is identically zero: one matrix product reproduces `corr2d` perfectly.

Equivalently, we can unroll the *kernel* instead of the input into a sparse banded matrix $\mathbf{W}$ multiplying the flattened image—the exact 2D analog of panel (c) in Figure 6.4, where every row repeats the same handful of kernel weights. Seeing a convolution as a structured dense layer brings us full circle to the derivation of Section 6.1 (and the exercises build $\mathbf{W}$ explicitly).

6.2.4 Feature Maps and Receptive Fields

A single convolutional layer sees only a patch the size of its kernel, whereas recognizing whole objects demands integrating context across much wider areas. Recall from Section 6.1.4 that a convolutional layer's output is called a *feature map* (like our vertical edge map `Y`). For any element x in a feature map, its *receptive field* is the region of the input image (or of an earlier layer) whose values can influence x during forward propagation—borrowing the neurophysiology term from Figure 6.1.

6.2.4.1 Stacked Layers

In Figure 6.7, the shaded output `19` has a 2×2 receptive field covering the top-left four pixels of $\mathbf{X}$. Now stack a *second* 2×2 convolutional layer on top of the 2×2 output $\mathbf{Y}$ to produce a single scalar z: z depends on all four elements of $\mathbf{Y}$, which together depend on all nine elements of $\mathbf{X}$. Stacking layers expands the receptive field smoothly with depth.

How quickly does it grow? Each $k \times k$ convolutional layer reaches $k-1$ pixels farther than a single element of its input, so stacking L layers with kernel sizes $k_1, \dots, k_L$ yields a receptive field of side length

$$r = 1 + \sum_{i=1}^{L} (k_i - 1). \tag{6.11}$$

For a stack of L consecutive 3×3 layers, each layer adds 2 pixels, covering $(2L+1)\times(2L+1)$ inputs: 5×5 after two layers and 7×7 after three (Figure 6.9). This linear growth is actually a blessing rather than a reason to reach for huge kernels: three stacked 3×3 layers achieve the same 7×7 receptive field as a single 7×7 layer using far fewer parameters ($3 \times 9 = 27$ weights vs. 49 per channel pair) while interleaving three nonlinearities instead of one.

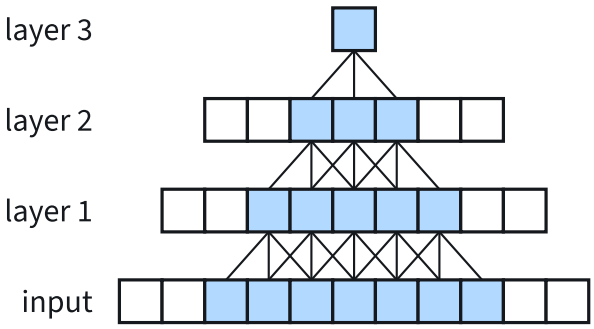


Figure 6.9: Receptive field of one unit after three one-dimensional layers with kernel size 3. Shaded cells are the units that the top unit depends on, and the lines connect each of them to its inputs. Each layer widens the receptive field by 2, to 7 inputs.

6.2.4.2 Effective Receptive Fields

Keep in mind that r measures the *theoretical* receptive field (every input pixel with *any* path to an output unit) without saying how strongly each pixel actually contributes. Measuring input gradients directly, Luo et al. (2016) showed that the *effective* receptive field decays sharply away from the center in a roughly Gaussian bell curve, in both untrained and trained networks. Deep units can theoretically see far across the image, but their sensitivity concentrates heavily near the center.

Counting paths explains the bell shape. An input at the edge of the receptive field reaches the output unit along a single chain of connections, whereas the center input can take many routes through the layers in between. Figure 6.10 counts them for the tower of Figure 6.9: the center input has seven paths to the top unit and each edge input has one. If all weights have a similar magnitude, the influence of an input grows with its number of paths.

Two practical gaps remain in our basic layer: every unpadded convolution shrinks spatial dimensions while under-sampling border pixels, and a single kernel detects only one pattern. We address spatial sizing and borders next in Section 6.3, and multi-channel feature maps in Section 6.4.

6.2.5 Summary

A convolutional layer applies two-dimensional cross-correlation with a learned kernel plus a scalar bias, sliding the kernel across every valid window of the input. A simple 1×2

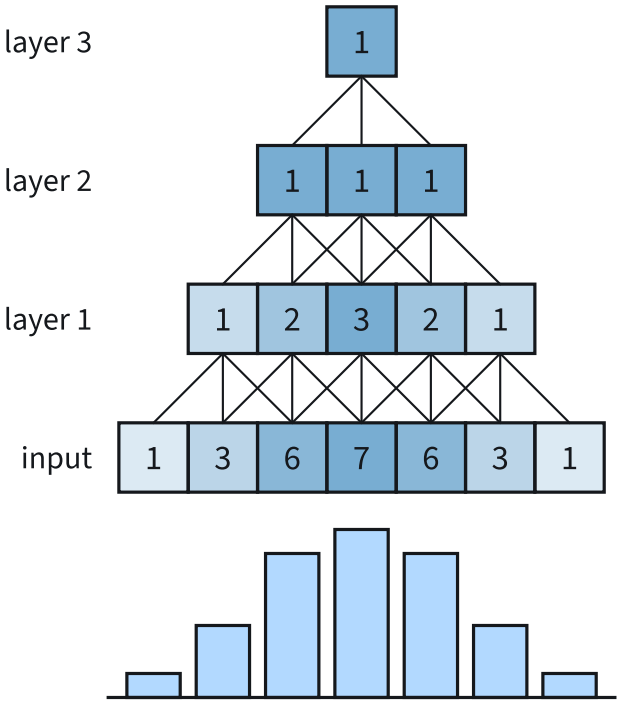


Figure 6.10: Number of paths from each unit to the top unit of Figure 6.9 (three one-dimensional layers with kernel size 3). Each count is the sum of the counts of the units that the cell feeds. Darker cells have more paths than the other cells of their layer, and the bars repeat the counts of the input row: 1, 3, 6, 7, 6, 3, 1.

finite-difference kernel `[1, -1]` detects vertical edges anywhere in an image, and gradient descent recovers those exact weights from input–output data alone, just as deep CNNs learn entire hierarchies of filters end to end. Mathematical convolution flips the kernel, making no difference once kernels are learned. A convolution can be evaluated as a single matrix multiplication (*im2col*), and stacking L convolutional layers widens each unit's receptive field linearly by $k_i - 1$ per layer.

6.2.6 Exercises

1. code **Hand-designed kernels.** Design some kernels manually.
    1. Given a directional vector $\mathbf{v} = (v_1, v_2)$, derive an edge-detection kernel that detects edges orthogonal to $\mathbf{v}$, i.e., edges in the direction $(v_2, -v_1)$.
    2. Derive a finite difference operator for the second derivative. Which structures in images respond most strongly to it?
    3. What is the minimum size of a kernel to obtain a derivative of order d?
    4. How would you design a blur kernel? Why might you want to use such a kernel?
    5. Load a photograph, convert it to one channel, and downsample it to about 100×100 pixels (`corr2d` loops in Python). Apply the edge detector, the blur kernel, and the sharpening kernel "twice the identity minus the blur", and display each result next to the original. What does the edge map of a photograph show that the edge map of the black band could not?

    Adapted from Stanford CS231n, Assignment 2, ConvolutionalNetworks notebook.

2. code **Convolution as a banded matrix.** The patch matrix of Section 6.2.3 unfolds the input. The opposite construction unrolls the kernel. Let $\mathbf{X}$ be a 4×4 input, $\mathbf{K}$ a 3×3 kernel, and $\mathrm{vec}(\cdot)$ the row-by-row flattening.
    1. Construct the 4×16 matrix $\mathbf{W}$ with $\mathbf{W}\,\mathrm{vec}(\mathbf{X}) = \mathrm{vec}(\mathbf{Y})$, where $\mathbf{Y}$ is the cross-correlation of $\mathbf{X}$ with $\mathbf{K}$. How many nonzero entries does each row have, and how many distinct values appear in $\mathbf{W}$?
    2. Build $\mathbf{W}$ in code from `K` and confirm that the product reproduces `corr2d(X, K)`.
    3. For an $n \times n$ input and a $k \times k$ kernel, count the entries of $\mathbf{W}$ stored as a dense array, the nonzero entries of $\mathbf{W}$, and the entries of the patch matrix. How does each count grow with n?
    4. Keep the sparsity pattern of $\mathbf{W}$ but give every row its own k^2 values. The result is a locally connected layer: the fourth-order tensor V of Section 6.1 in $[\mathbf{H}]_{i,j} = [\mathbf{U}]_{i,j} + \sum_{a,b} [\mathsf{V}]_{i,j,a,b} [\mathbf{X}]_{i+a,j+b}$, restricted to a $k \times k$ window of offsets (a, b) before translation equivariance ties the weights. For an $n \times n$ input, count the parameters of this layer, of a fully connected layer with the same input and output shapes, and of the convolution.

    Adapted from CMU 11-785, Homework 2 Part 1.

3. code **Effective receptive field.** ● The theoretical receptive field r of this section bounds which inputs *can* influence an output; the effective receptive field measures how much they actually do (Luo et al. 2016). Build a stack of L single-channel 3×3 convolutional layers with no padding, no bias, and no nonlinearity on a 32×32 input, using the built-in layer of Section 6.2.2.3. Draw every kernel entry independently from a standard normal distribution.
    1. For $L = 3$ and a single draw of the kernels, backpropagate a unit gradient from

an output unit near the center of the last layer to the input and plot the magnitude of the input gradient. Compare its support with the 7×7 theoretical receptive field, and say where the magnitude is largest.
2. Average the squared input gradient over 500 draws of the kernels, for $L = 3$ and for $L = 10$. Sum each averaged map over its rows and normalize the resulting column profile to sum to one. How does the ratio of the profile's standard deviation to the theoretical side length $2L + 1$ change with depth?
3. Figure 6.10 counts the paths for $L = 3$ in one dimension. Show that for general L the expected squared input gradient, summed over rows, is proportional to the convolution of L copies of the vector $(1, 1, 1)$. Derive how the standard deviation of the normalized profile grows with L.

4. code **Backward pass of a convolution.** ● Let $\mathbf{Y}$ be the single-channel cross-correlation of an input $\mathbf{X}$ with a kernel $\mathbf{K}$, as computed by `corr2d`, and let ℓ be a scalar loss that depends on $\mathbf{X}$ and $\mathbf{K}$ only through $\mathbf{Y}$.
    1. Derive $\partial \ell / \partial \mathbf{K}$ and $\partial \ell / \partial \mathbf{X}$ in terms of $\partial \ell / \partial \mathbf{Y}$, in the style of the backpropagation equations of Section 4.3.
    2. Implement your expression for $\partial \ell / \partial \mathbf{K}$ without automatic differentiation for the loss $\ell = \sum_{i,j} G_{ij} Y_{ij}$, where $\mathbf{G}$ is a fixed random matrix of the shape of $\mathbf{Y}$. Compare it with a centered finite-difference gradient with step size 10^{-2} on a small random `X` and `K`, and report the largest relative error.
    3. Check your expression for $\partial \ell / \partial \mathbf{K}$ against the kernel gradient that automatic differentiation computes in the experiment of Section 6.2.2.3.

Adapted from Michael Nielsen, Neural Networks and Deep Learning, Chapter 6, and Stanford CS231n, Assignment 2, ConvolutionalNetworks notebook.

6.3 Padding, Stride, and Dilation

An unpadded $k \times k$ convolution shrinks its input by $k-1$ pixels along each axis (Section 6.2). Across ten 5×5 layers, a 240×240 image shrinks to 200×200, discarding 30% of its area along the borders. Yet keeping every layer at full resolution with stride 1 has the opposite problem: each 3×3 layer widens a unit's receptive field by only two pixels, requiring over a hundred layers to span a 224×224 image. And in dense prediction tasks like semantic segmentation, we want *both* a wide field of view and full spatial resolution without the quadratic cost of huge kernels.

Three geometric controls navigate these trade-offs: *padding* surrounds the input with a border so kernels can center on edge pixels and preserve spatial dimensions; *stride* steps the window by more than one pixel to downsample feature maps and accelerate receptive-field growth; and *dilation* spaces kernel weights apart so a small kernel spans a wider footprint at full resolution.

6.3.1 Padding

6.3.1.1 Boundary Pixels

Even before successive layers trim away the perimeter of an image, pixels near the border are heavily underutilized. Figure 6.11 counts how many sliding windows cover each pixel for an unpadded convolution. With a $k \times k$ kernel, any interior pixel at least $k - 1$ steps

from every border participates in k^2 windows, whereas an edge pixel appears in at most k windows and a corner pixel in only one. For a 5×5 kernel, an interior pixel contributes to 25 output locations while a corner pixel contributes to just one, meaning that an object near the corner barely registers in the next layer. Without intervention, each layer draws disproportionately from the center of its input.

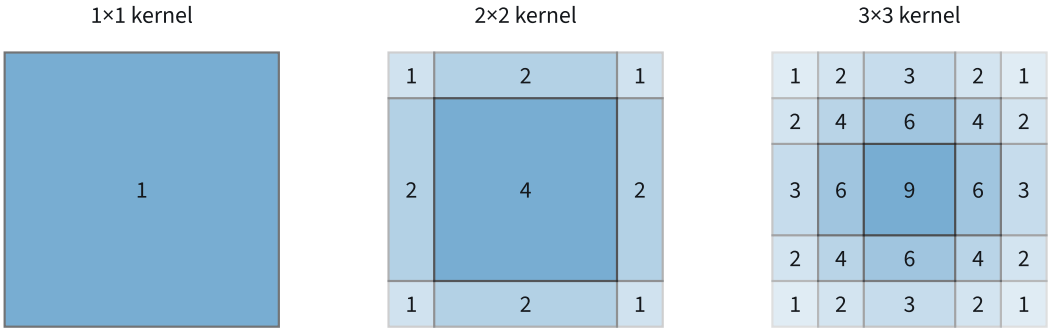


Figure 6.11: Number of windows that contain each pixel when a 1×1, 2×2, or 3×3 kernel slides over a large input without padding. Interior pixels are covered k^2 times, corner pixels once. Each stretched cell stands for all pixels at that distance from the border, however many there are.

6.3.1.2 Zero Padding

The simplest remedy is to surround the input with a border of extra filler pixels (typically zeros) before sliding the kernel across it, a technique known as *padding*. In Figure 6.12, adding one row of zeros to the top and bottom and one column to the left and right expands a 3×3 input into a 5×5 grid, allowing a 2×2 kernel to produce a 4×4 output rather than a 2×2 one. The shaded upper-left output element combines three padded zeros with the corner of the original input: $0 \times 0 + 0 \times 1 + 0 \times 2 + 0 \times 3 = 0$.

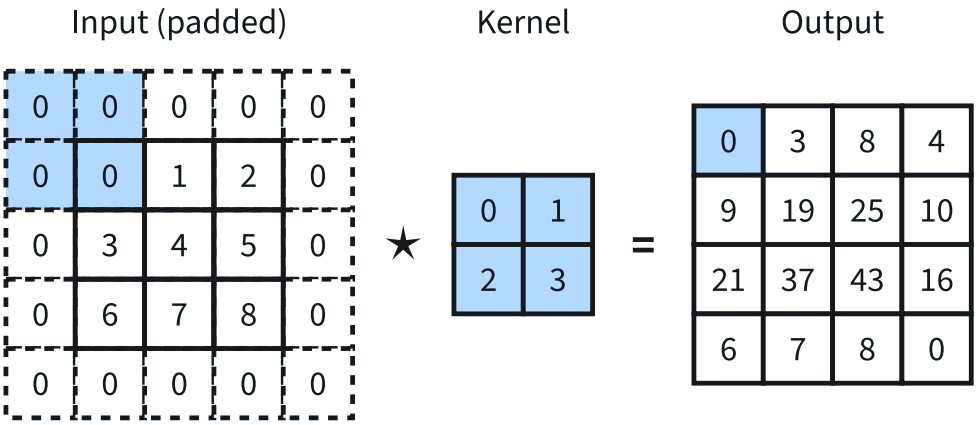


Figure 6.12: Two-dimensional cross-correlation with one pixel of zero padding on every side.

Following the convention used in deep learning frameworks, we count padding per side: p_h rows added above and p_h below the input, and p_w columns added to the left and to the right. The padded input then has shape $(n_\text{h} + 2p_\text{h}) \times (n_\text{w} + 2p_\text{w})$, yielding an output of shape

$$(n_\text{h} + 2p_\text{h} - k_\text{h} + 1) \times (n_\text{w} + 2p_\text{w} - k_\text{w} + 1). \tag{6.12}$$

Each row of padding on either side adds one row to the output, and each column adds one column. Because the kernel removes $k_\text{h} - 1$ rows and $k_\text{w} - 1$ columns, setting $p_\text{h} = (k_\text{h} - 1)/2$ and $p_\text{w} = (k_\text{w} - 1)/2$ preserves the exact input height and width. This is by far the most common choice in practice: a stack of shape-preserving layers maintains the same spatial resolution at every depth, making network architectures much easier to design and reason about.

Notice that dividing $k - 1$ by two explains why CNNs almost always use odd kernel sizes, such as 1, 3, 5, or 7, paired with paddings of 0, 1, 2, or 3. An odd kernel has a well-defined center pixel, so symmetric padding places the window computing output `Y[i, j]` directly centered on input pixel `X[i, j]`, keeping the input and output grids aligned. For an even kernel, $k_\text{h} - 1$ is odd, so preserving the spatial dimensions requires asymmetric padding (one extra row on one side compared to the other) and shifts the output grid by half a pixel relative to the input.

To verify these output-shape formulas in code, we first define a small helper. Because built-in convolutional layers expect four-dimensional inputs carrying batch and channel axes, `comp_conv2d` inserts two singleton dimensions around a 2D input, applies the layer, and strips those extra dimensions from the result.

```
def comp_conv2d(conv2d, X):
    """Apply a convolutional layer to a single-channel 2D input."""
    Y = conv2d(X.reshape((1, 1) + X.shape))  # input (1, 1, h, w)
    return Y.reshape(Y.shape[2:])
```

For a 3×3 kernel, shape-preserving padding requires $p = (3 - 1)/2 = 1$ (`padding=1` in code). Applying this layer to an 8×8 input should leave the spatial shape unchanged at 8×8.

```
X = torch.rand(size=(8, 8))
conv2d = nn.LazyConv2d(1, kernel_size=3, padding=1)
comp_conv2d(conv2d, X).shape
```

```
torch.Size([8, 8])
```

6.3.1.3 Padding Values

Zero padding implicitly assumes that the image is surrounded by black pixels outside its frame. More formally, padding specifies a *boundary condition*: a rule for supplying values beyond the image border where no measurement exists. To see how zero padding affects the output, let us pad a uniform 5×5 image of all ones with one pixel of zeros on each side and cross-correlate it with a 3×3 kernel of all ones using `corr2d` from Section 6.2. Each output entry then counts how many true image pixels fall inside its window.

```
img = d2l.ones((5, 5))
d2l.corr2d(F.pad(img, (1, 1, 1, 1)), d2l.ones((3, 3)))
```

```
tensor([[4., 6., 6., 6., 4.],
        [6., 9., 9., 9., 6.],
        [6., 9., 9., 9., 6.],
        [6., 9., 9., 9., 6.],
        [4., 6., 6., 6., 4.]])
```

The interior outputs equal 9, edge outputs drop to 6, and corner outputs drop to 4: even though the original image is completely uniform, the layer sees a dark frame around it. That artificial frame propagates inward through subsequent layers, which can learn to use it as an implicit positional cue; at the same time, boundary artifacts can create blind spots near the edges where the network responds weakly (Alsallakh et al. 2020). Two common alternatives avoid introducing zeros: *reflect padding* mirrors pixels across the border, while *replicate padding* repeats the border values outward. Both map a constant image to a constant output, making them popular in image-generation and restoration networks where border darkening would appear directly in the generated pixels. In classification networks, however, zero padding remains the standard choice because it is simple and can be applied implicitly by fast kernels without allocating a larger padded tensor.

6.3.2 Stride

6.3.2.1 Strided Convolution

Keeping a 224×224 image at full resolution throughout a network is both expensive and slow to aggregate global context: every channel contains 50,176 spatial positions per layer, and since each 3×3 convolution expands a unit's view by only two pixels, more than 100 layers would be needed before a single unit could see the entire image. Stepping the convolution window across the input in larger increments solves both problems at once, reducing the spatial resolution and shrinking the depth needed for global context.

Until now, our sliding window has advanced one pixel at a time across each row and down each column. We call the step size along the height and width the *stride*. With vertical stride s_{h} and horizontal stride s_{w}, the layer evaluates the kernel only at every s_{h}-th row and every s_{w}-th column, cutting the number of output elements by roughly a factor of $s_{\mathrm{h}} s_{\mathrm{w}}$. Figure 6.13 illustrates a vertical stride of 3 and a horizontal stride of 2 on the padded input from Figure 6.12 (an asymmetric choice that makes the two axes easy to distinguish). Each shaded output matches the color of the window that produced it: the blue entry equals $0 \times 0 + 0 \times 1 + 1 \times 2 + 2 \times 3 = 8$ from a window shifted two columns to the right of the first, and the orange entry equals $0 \times 0 + 6 \times 1 + 0 \times 2 + 0 \times 3 = 6$ from a window shifted three rows down. Stepping two more columns to the right would push the window past the right edge of the padded input, leaving two outputs per row.

To determine the output height in general, note that the window starts at row 0 of the padded input and must fit entirely within the $n_{\mathrm{h}} + 2p_{\mathrm{h}}$ padded rows. Its valid starting rows are $0, s_{\mathrm{h}}, 2s_{\mathrm{h}}, \ldots$ up to $n_{\mathrm{h}} + 2p_{\mathrm{h}} - k_{\mathrm{h}}$. Applying the same reasoning to the width gives an output shape of

$$\left(\left\lfloor \frac{n_{\mathrm{h}} + 2p_{\mathrm{h}} - k_{\mathrm{h}}}{s_{\mathrm{h}}} \right\rfloor + 1\right) \times \left(\left\lfloor \frac{n_{\mathrm{w}} + 2p_{\mathrm{w}} - k_{\mathrm{w}}}{s_{\mathrm{w}}} \right\rfloor + 1\right). \tag{6.13}$$

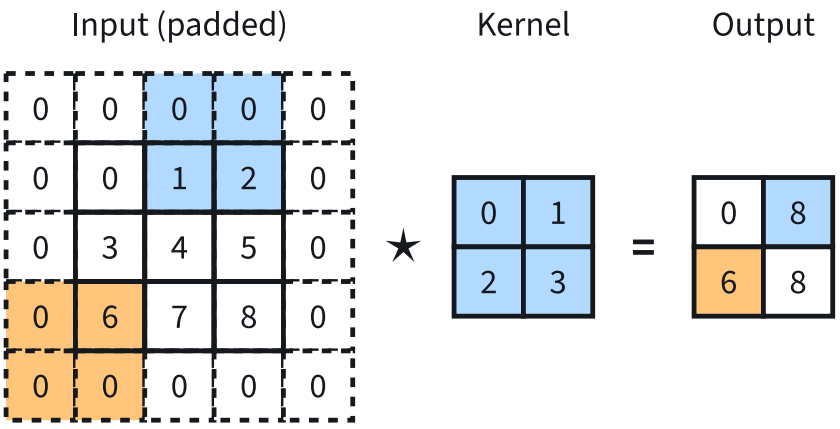


Figure 6.13: Cross-correlation with strides of 3 and 2 for height and width, respectively. Each shaded output has the color of the window that produces it.

Here the floor term counts how many full steps the window can take after its initial placement, and the $+1$ accounts for that starting position. Checking Figure 6.13 yields $\lfloor 3/3 \rfloor + 1 = 2$ rows and $\lfloor 3/2 \rfloor + 1 = 2$ columns, and setting $s_\mathrm{h} = s_\mathrm{w} = 1$ recovers our earlier padding formula. When we pair stride with shape-preserving padding ($2p_\mathrm{h} = k_\mathrm{h} - 1$), the output height simplifies to $\lfloor (n_\mathrm{h} - 1)/s_\mathrm{h} \rfloor + 1 = \lceil n_\mathrm{h}/s_\mathrm{h} \rceil$: a stride of s divides the spatial resolution by s, rounding up.

A very common downsampling layer, used throughout the ResNet family (Section 7.4), is a 3×3 convolution with padding 1 and stride 2. Applied to our 8×8 input, it halves both spatial dimensions to 4×4.

```
conv2d = nn.LazyConv2d(1, kernel_size=3, padding=1, stride=2)
comp_conv2d(conv2d, X).shape
```

```
torch.Size([4, 4])
```

Height and width need not share the same kernel size, padding, or stride; the output-shape formula applies independently to each axis.

An important special case occurs when the stride equals the kernel size and padding is zero: the windows then partition the input into non-overlapping tiles. For instance, a 16×16 convolution with stride 16 splits a 224×224 image into a 14×14 grid of patches and projects each patch to a vector of channel outputs. This *patchify* stem is how vision transformers convert images into sequences of tokens, and several modern CNNs open with the same operation.

When height and width share the same settings, we drop the subscripts and simply write p and s. Unless specified otherwise, a convolutional layer defaults to stride 1 and zero padding ($p = 0$).

6.3.2.2 Stride and the Receptive Field

In Section 6.2, we saw that at stride 1 each $k \times k$ layer expands the receptive field by $k - 1$ pixels per axis, producing only linear growth with depth. Stride accelerates this growth dramatically. After a layer with stride 2, adjacent entries in its output correspond to locations two pixels apart in the original image; consequently, a subsequent 3×3 kernel widens the receptive field by $(3 - 1) \cdot 2 = 4$ input pixels rather than 2. More generally, a single step in the input to layer i spans $\prod_{j<i} s_j$ pixels of the original image, so a unit at the top of an L-layer stack with kernel sizes k_i and strides s_i has a receptive field of width

$$r = 1 + \sum_{i=1}^{L} (k_i - 1) \prod_{j=1}^{i-1} s_j, \tag{6.14}$$

where the empty product for $i = 1$ is defined as 1. Each layer contributes the span of its kernel measured in original image pixels; setting all strides to 1 reduces every term to $k_i - 1$, recovering the linear count from Section 6.2. Notice that padding does not enter the formula: padding determines which output locations exist, not how wide a patch each unit observes. Comparing two three-layer stacks of 3×3 convolutions, strides $(1, 1, 1)$ give $r = 1 + 2 + 2 + 2 = 7$, whereas strides $(2, 2, 1)$ more than double the reach to $r = 1 + 2 + 4 + 8 = 15$.

Because each stride multiplies the effective reach of every layer above it, the receptive field grows geometrically whenever downsampling continues. After five stride-2 stages (typical in image classifiers operating on 224×224 inputs), a single step in the feature map spans 32 input pixels, enabling top-level units to view the entire image in only a handful of layers.

6.3.3 Dilation

6.3.3.1 Dilated Convolution

Strided convolutions buy a wide receptive field at the expense of spatial resolution. For some tasks, however, we want both a wide context and an output at every input pixel. Simply enlarging the kernel would give a wider view, but its parameter count and compute cost grow quadratically with kernel size. Whereas padding and stride control *where* the kernel is placed, dilation modifies the *spacing of the kernel itself*: a *dilated convolution* (Yu and Koltun 2016) inserts gaps between adjacent weights so that they are evaluated d pixels apart, where d is the *dilation* rate. A $k \times k$ kernel with dilation d still contains only k^2 learnable weights, yet spans an *effective kernel size* of

$$k' = k + (k - 1)(d - 1) = d(k - 1) + 1 \tag{6.15}$$

pixels along each axis, skipping $d - 1$ pixels between consecutive taps. Setting $d = 1$ recovers the ordinary dense convolution. Figure 6.14 contrasts a 3×3 kernel at $d = 1$ and $d = 2$: at dilation 2, the same nine weights cover a 5×5 footprint by sampling every other pixel, while still requiring only nine multiplications per output location.

As far as output dimensions are concerned, a dilated kernel behaves identically to a dense kernel of size k'. Replacing k with $k' = d(k - 1) + 1$ in our strided formula yields a single unified expression for the output size along an axis with arbitrary padding, stride, and dilation:

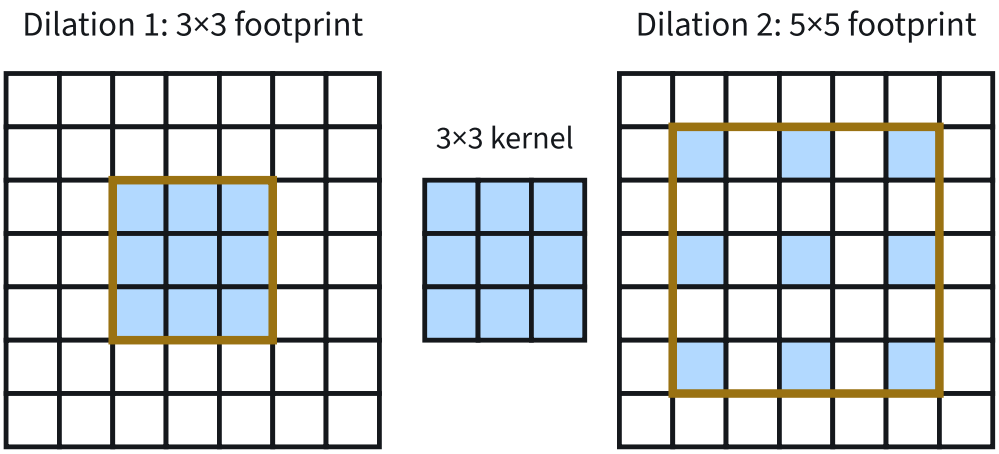


Figure 6.14: A 3×3 kernel at dilation 1 and dilation 2. At dilation 2 its nine weights span a 5×5 footprint.

$$\left\lfloor \frac{n + 2p - d(k-1) - 1}{s} \right\rfloor + 1, \tag{6.16}$$

which applies independently to height and width with their respective parameters n, k, p, s, and d. Setting $d = 1$ reproduces the strided formula, and setting $s = 1$ and $p = 0$ brings us all the way back to $n - k + 1$. On our 8×8 input, a 3×3 kernel with dilation 2 should therefore produce the same 4×4 output shape as a dense 5×5 kernel.

```
conv2d = nn.LazyConv2d(1, kernel_size=3, dilation=2)
conv5 = nn.LazyConv2d(1, kernel_size=5)
comp_conv2d(conv2d, X).shape, comp_conv2d(conv5, X).shape
```

```
(torch.Size([4, 4]), torch.Size([4, 4]))
```

6.3.3.2 Dilation and the Receptive Field

At stride 1, a dilated layer widens the receptive field by $k' - 1 = d(k-1)$ pixels, so Equation 6.14 still holds once each k_i is replaced by its effective size k'_i. If we double the dilation rate at each layer, $d_i = 1, 2, 4, \ldots, 2^{L-1}$, a stack of L layers with 3×3 kernels attains a receptive field of width $1 + 2(1 + 2 + \cdots + 2^{L-1}) = 2^{L+1} - 1$. Just like a strided stack, its receptive field grows exponentially with depth, yet every layer retains the full spatial resolution of the input.

Figure 6.15 illustrates these two strategies side by side. Panel (a) shows our strided stack with strides $(2, 2, 1)$: the three layers add 2, 4, and 8 pixels to the top unit's view, reaching 15 input pixels in total. Panel (b) achieves the exact same 15-pixel receptive field using stride 1 and dilations $(1, 2, 4)$, wiring the top unit to the input through the identical pattern of connections. The sole difference is spatial density: the strided stack retains only every fourth spatial position, whereas the dilated stack evaluates all of them. Indeed, without padding and with shared weights, the strided stack's output is simply every fourth entry

of the dilated stack's output, even when elementwise nonlinearities are applied after every layer. That makes dilated convolutions especially valuable for *dense prediction* tasks such as semantic segmentation, where we must assign a class label to every pixel while still drawing on broad surrounding context (Yu and Koltun 2016). The tradeoff is computation and memory, because every layer must process full-resolution feature maps; an alternative is to downsample with strides and later recover spatial resolution using learned upsampling layers such as the transposed convolution of Section 32.10.

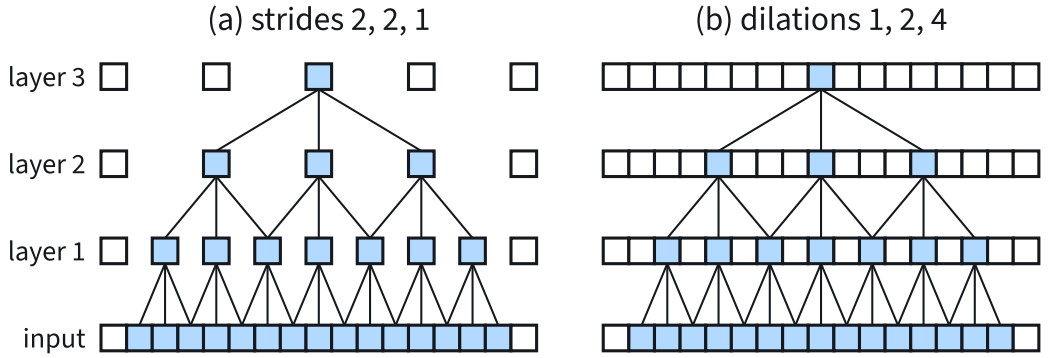


Figure 6.15: Receptive field of the top unit of three stacked layers with kernel size 3, in one dimension. Cells are the positions each layer computes, shaded cells those the top unit depends on, and lines the kernel's taps. (a) Strides 2, 2, 1: layer 1 keeps every second position, layers 2 and 3 every fourth. (b) Stride 1 with dilations 1, 2, 4: every layer keeps all positions. Both top units see the same 15 input pixels.

So far, every example in this section has operated on a single-channel input with a single kernel. Real-world images contain multiple color channels, and useful hidden layers detect dozens or hundreds of distinct patterns at once. We lift both restrictions in Section 6.4.

6.3.4 Summary

Without padding, a convolution shrinks its input by $k-1$ pixels along each axis and samples border pixels in far fewer windows than interior ones. Adding $(k-1)/2$ pixels of padding to each side of an odd kernel preserves the input shape and keeps each output centered on its corresponding input pixel, though zero padding introduces a boundary condition that deep layers can detect. Using a stride $s > 1$ evaluates only every s-th window position, reducing spatial resolution and computation by roughly a factor of s per axis while causing the receptive field of higher layers to grow geometrically. Finally, dilation d spaces a kernel's weights across an effective footprint of $d(k-1)+1$, expanding the receptive field exponentially at full resolution without adding parameters. All three mechanisms are captured by a single output-shape formula, Equation 6.16.

6.3.5 Exercises

1. code **Mirror padding.** Mirror padding, also called reflect padding, extends a tensor by reflecting its entries about the border without repeating the border entry: padding the vector $(1, 2, 3)$ by one entry on each side gives $(2, 1, 2, 3, 2)$.

   1. Implement mirror padding by one entry on every side of a two-dimensional tensor, and check the result by hand on a 3×3 tensor with distinct entries.
   2. Repeat the computation of Section 6.3.1.3 with mirror padding instead of zero padding, and explain the result.
   3. Take the 5×5 ramp $x_{ij} = j$ and the 3×3 kernel whose entries all equal $\frac{1}{9}$. Show that in the interior the output equals the input. Compute the outputs in the first and last columns under zero padding and under mirror padding, and find a way to pad that reproduces the ramp exactly.
2. **Dilation and gridding.** A network stacks four 3×3 convolutions with stride 1. The doubling dilation schedule $(1, 2, 4, 8)$ gives one output element a receptive field of side 31. Compare it with the constant schedule $(4, 4, 4, 4)$.
   1. Use Equation 6.14, with each kernel replaced by its effective size, to compute the receptive field of the constant schedule.
   2. For each schedule, determine which pixels inside the receptive field the output actually depends on, and count them. A schedule whose output depends only on a sparse lattice within its receptive field is said to exhibit *gridding*.
   3. Give one reason gridding is harmful for dense prediction tasks such as segmentation.
   4. ● State a condition on the dilations $d_1, \dots, d_L$ of a stack of 3×3 convolutions with stride 1 that guarantees that the output depends on every pixel of its receptive field, and show that the condition suffices.
3. **Strides versus dilations.** Consider the two one-dimensional stacks of Figure 6.15, without padding, with kernels $\mathbf{w}_1, \mathbf{w}_2, \mathbf{w}_3$ of size 3 shared by both stacks and an elementwise nonlinearity after every layer.
   1. Write the output of each stack at position i as a function of the input, and show that the output of the strided stack at position i equals the output of the dilated stack at position $4i$.
   2. For an input of length n, count the multiplications each stack performs, ignoring the border.
   3. Generalize the first part: which dilations reproduce a stack with strides $s_1, \dots, s_L$?
4. **Shapes through a stem.** A 224×224 image passes through four layers: a 7×7 convolution with padding 3 and stride 2, a 3×3 convolution with padding 1 and stride 2, and two 3×3 convolutions with padding 1 and stride 1.
   1. Give the height of the output of every layer.
   2. Use Equation 6.14 to compute the receptive field of a unit of the last layer. What fraction of the image's width does it see?
   3. Replace the first two layers by a single 4×4 convolution with stride 4 and no padding, the patchify layer of this section. Give the output height and the receptive field again. Do neighboring windows of the first layer still overlap?

6.4 Multiple Input and Multiple Output Channels

So far our convolutions have mapped a single 2D grid to a single 2D grid. Real images and hidden representations have multiple *channels* on both sides: an RGB photograph is a $3 \times h \times w$ tensor (requiring cross-channel comparisons to detect a red-on-green boundary of equal luminance), and a hidden layer extracts dozens or hundreds of distinct feature maps in parallel.

As derived in Section 6.1, a layer with c_{i} input channels and c_{o} output channels maintains

one $c_\mathrm{i} \times k_\mathrm{h} \times k_\mathrm{w}$ kernel per output channel, cross-correlating across all input channels and summing the results. In this section, we implement multi-channel convolutions, account for their parameter, FLOP, and activation-memory costs (governed by the product $c_\mathrm{i} c_\mathrm{o}$), and introduce 1×1 bottleneck convolutions.

6.4.1 Multiple Input Channels

Let us begin with the input side. When the input contains c_i channels, the kernel must supply one $k_\mathrm{h} \times k_\mathrm{w}$ filter for each input channel, giving a third-order kernel tensor of shape $c_\mathrm{i} \times k_\mathrm{h} \times k_\mathrm{w}$. We cross-correlate every input channel $\mathbf{X}_c$ with its corresponding kernel slice $\mathbf{K}_c$ using two-dimensional cross-correlation (Section 6.2, computed by `corr2d`) and sum the c_i intermediate grids:

$$\mathbf{Y} = \sum_{c=1}^{c_\mathrm{i}} \mathbf{X}_c \star \mathbf{K}_c. \tag{6.17}$$

Each entry of the resulting two-dimensional output is therefore a weighted sum over the same spatial window across all input channels, combining $c_\mathrm{i} k_\mathrm{h} k_\mathrm{w}$ scalar products. Figure 6.16 illustrates this calculation for two input channels, where the shaded upper-left output entry equals $(1 \times 1 + 2 \times 2 + 4 \times 3 + 5 \times 4) + (0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3) = 56$.

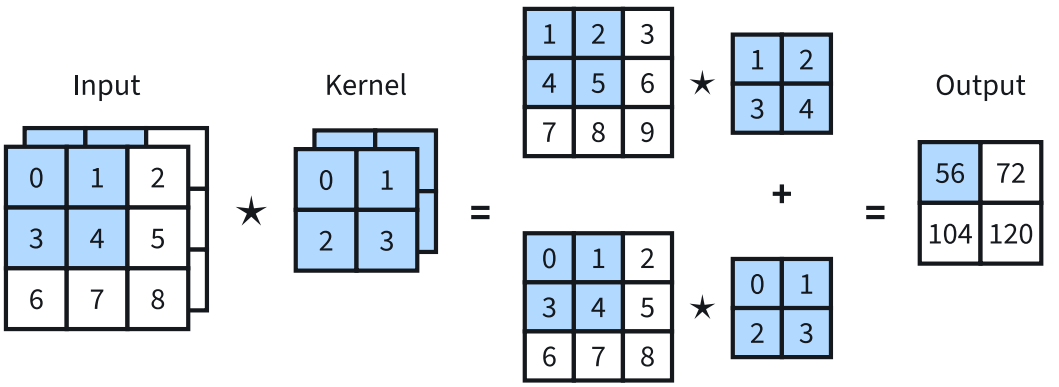


Figure 6.16: Cross-correlation with two input channels: one cross-correlation per channel, then a sum.

Summing across channels is precisely what allows a kernel to compare information carried by different channels, such as color. Consider a 1×1 kernel with weight $+1$ on the red channel and -1 on the green channel: at every pixel it computes red minus green, producing positive values in red regions, negative values in green regions, and zero in neutral gray regions regardless of brightness. Wherever its sign flips between adjacent pixels, a red region meets a green one, even if both regions have identical luminance, in which case a grayscale brightness filter would see no edge at all.

Implementing multi-input cross-correlation translates the formula directly into Python: we call `corr2d` once per channel and sum the results.

```
def corr2d_multi_in(X, K):
    """Cross-correlate each channel of X with its channel of K and add."""
    return sum(d2l.corr2d(x, k) for x, k in zip(X, K))
```

Let us verify the implementation on the input and kernel tensors from Figure 6.16; the upper-left output entry should be 56.

```
X = d2l.tensor([[[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]],
                [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]]])
K = d2l.tensor([[[0.0, 1.0], [2.0, 3.0]], [[1.0, 2.0], [3.0, 4.0]]])

corr2d_multi_in(X, K)
```

```
tensor([[ 56.,  72.],
        [104., 120.]])
```

6.4.2 Multiple Output Channels

No matter how many input channels it reads, a single multi-input kernel produces just one two-dimensional *feature map* (Section 6.1.4). To extract c_o distinct feature maps simultaneously, a convolutional layer maintains c_o such kernels, each of shape $c_\mathrm{i} \times k_\mathrm{h} \times k_\mathrm{w}$. Stacking them along a new axis yields a fourth-order kernel tensor of shape $c_\mathrm{o} \times c_\mathrm{i} \times k_\mathrm{h} \times k_\mathrm{w}$, where output channel o is given by

$$\mathbf{Y}_o = \sum_{c=1}^{c_\mathrm{i}} \mathbf{X}_c \star \mathbf{K}_{o,c}, \quad o = 1, \ldots, c_\mathrm{o}. \tag{6.18}$$

Notice that every output channel connects to *every* input channel through its own $k_\mathrm{h} \times k_\mathrm{w}$ slice $\mathbf{K}_{o,c}$; this dense all-to-all channel wiring is the primary driver of the layer's computational cost. Adding a scalar bias u_o per output channel recovers the multi-channel convolutional layer derived in Section 6.1. In typical CNN architectures, the number of channels grows as spatial resolution shrinks via strides or pooling, trading fine-grained spatial layout for richer, jointly learned feature descriptions at each location.

PyTorch stores kernels channels-first as $(c_\mathrm{o}, c_\mathrm{i}, k_\mathrm{h}, k_\mathrm{w})$.

To implement multi-channel output cross-correlation, we iterate over the output kernels, apply `corr2d_multi_in` to each, and stack the resulting feature maps along axis 0.

```
def corr2d_multi_in_out(X, K):
    """Apply corr2d_multi_in once per output channel of K and stack."""
    return d2l.stack([corr2d_multi_in(X, k) for k in K], 0)
```

Let us test this function by stacking `K`, `K + 1`, and `K + 2` into a kernel tensor with three output channels.

```
K = d2l.stack((K, K + 1, K + 2), 0)
K.shape
```

```
torch.Size([3, 2, 2, 2])
```

Applying `corr2d_multi_in_out` to our two-channel input `X` and the stacked $3 \times 2 \times 2 \times 2$ kernel `K` produces a three-channel output tensor of shape $(3, 2, 2)$.

```
corr2d_multi_in_out(X, K)
```

```
tensor([[[ 56.,  72.],
         [104., 120.]],

        [[ 76., 100.],
         [148., 172.]],

        [[ 96., 128.],
         [192., 224.]]])
```

The first output channel matches our earlier result because its kernel is the original `K`. We can verify the second channel by mental arithmetic: adding 1 to every kernel weight increases each output entry by the sum of all elements in its receptive window across both input channels. For the upper-left window that sum is 20, so the second channel begins at $56 + 20 = 76$.

6.4.2.1 Parameters and Cost

Adding channels gives a network expressive power, but it also scales up memory and computation rapidly. Including one scalar bias per output channel, a convolutional layer contains

$$c_{\mathrm{o}} c_{\mathrm{i}} k_{\mathrm{h}} k_{\mathrm{w}} + c_{\mathrm{o}} \tag{6.19}$$

learnable parameters. Near the input of a network, this number is modest: the 3×3 layer mapping 3 RGB channels to 64 channels in Section 6.1 has only 1,792 parameters. Deeper inside the network, however, channel counts grow large: a 3×3 layer mapping 256 channels to 256 channels requires 589,824 weights plus 256 biases.

Turning to computational cost, each output element requires one multiply-add for every weight in its output channel ($c_{\mathrm{i}} k_{\mathrm{h}} k_{\mathrm{w}}$ in all). Across an output feature map of spatial size $h_{\mathrm{o}} \times w_{\mathrm{o}}$ and c_{o} output channels, the layer performs

$$h_{\mathrm{o}} w_{\mathrm{o}} \cdot c_{\mathrm{o}} c_{\mathrm{i}} k_{\mathrm{h}} k_{\mathrm{w}} \tag{6.20}$$

multiply-adds. Applied to a 56×56 feature map, our 256-to-256 layer executes $56^2 \cdot 589{,}824 \approx 1.8 \times 10^9$ multiply-adds for a single image. Both parameter count and arithmetic cost scale with the product $c_{\mathrm{i}} c_{\mathrm{o}}$, meaning that doubling the width of every layer quadruples both. Because the same weights are reused at every spatial location, arithmetic cost also scales directly with the output area $h_{\mathrm{o}} w_{\mathrm{o}}$.

A third resource cost is just as important during training yet easily overlooked: *activation memory*. To compute gradients in the backward pass, the framework must retain each layer's output activations, storing $h_{\mathrm{o}} w_{\mathrm{o}} c_{\mathrm{o}}$ numbers per training example (Section 5.7.3.2). Unlike parameter memory, activation memory is largest in the *earliest* layers, where spatial resolution is highest even though parameter counts are tiny. For example, a 3-to-64 layer

producing a 224×224 output must cache $224^2 \cdot 64 \approx 3.2 \times 10^6$ activations per example, nearly two thousand times more numbers than its 1,792 parameters.

6.4.3 The 1×1 Convolution

At first glance, a 1×1 convolution ($k_\text{h} = k_\text{w} = 1$) might look pointless: because its spatial window covers only a single pixel, it cannot detect edges, corners, or any other pattern spanning neighboring locations. When multiple channels are present, however, it performs a very useful operation, just as our red-minus-green color filter did earlier. At every spatial coordinate (i, j), a 1×1 convolution multiplies the c_i-dimensional channel vector by a shared $c_\text{o} \times c_\text{i}$ weight matrix to produce a c_o-dimensional output vector. In other words, it acts as a fully connected layer applied independently at every pixel with weights tied across all spatial positions. Figure 6.17 depicts a 1×1 convolution mapping 3 input channels to 2 output channels while leaving height and width intact.

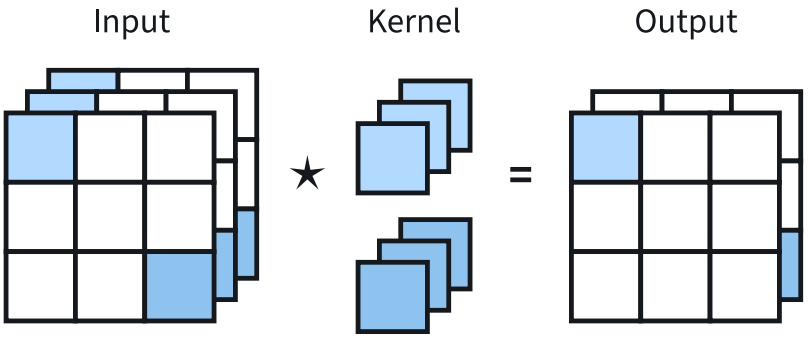


Figure 6.17: A 1×1 convolution with three input channels and two output channels. Each output element is a linear combination of the input elements at the same position.

Because the exact same linear map is applied at every pixel, we can compute the entire 1×1 convolution with a single matrix multiplication: we reshape the input into a $c_\text{i} \times hw$ matrix, left-multiply by the $c_\text{o} \times c_\text{i}$ kernel matrix, and reshape the output back to (c_o, h, w).

```
def corr2d_multi_in_out_1x1(X, K):
    """A 1x1 convolution as one matrix product."""
    c_i, h, w = X.shape
    c_o = K.shape[0]
    Y = d2l.matmul(d2l.reshape(K, (c_o, c_i)), d2l.reshape(X, (c_i, h * w)))
    return d2l.reshape(Y, (c_o, h, w))
```

Let us verify on random inputs that this matrix-multiplication formulation agrees with `corr2d_multi_in_out`.

```
X = d2l.normal(0, 1, (3, 3, 3))
K = d2l.normal(0, 1, (2, 3, 1, 1))
Y1 = corr2d_multi_in_out_1x1(X, K)
Y2 = corr2d_multi_in_out(X, K)
float(d2l.reduce_sum(d2l.abs(Y1 - Y2)))
```

```
0.0
```

As expected, the two results match up to floating-point rounding error.

Why are 1×1 convolutions so ubiquitous in modern architectures? Their main strength is adjusting channel dimensionality cheaply so that expensive spatial convolutions can operate on narrower representations. Return to our 256-to-256 3×3 layer. Instead of applying 3×3 kernels directly to all 256 channels, we can use a 1×1 convolution to project 256 channels down to 64, apply the 3×3 convolution across those 64 channels, and use a second 1×1 convolution to expand back to 256. Together, the three layers use only $16{,}384 + 36{,}864 + 16{,}384 = 69{,}632$ weights instead of $589{,}824$, a reduction of roughly $8.5\times$ in both parameters and multiply-adds. Such a *bottleneck* restricts intermediate spatial mixing to a 64-channel subspace, a tradeoff that architectures such as ResNet (Section 7.4) happily make in exchange for much greater depth. Placing an activation function after a 1×1 convolution turns it into a tiny nonlinear MLP evaluated at every pixel (Lin et al. 2013); without a nonlinearity, consecutive linear convolutions would simply collapse into a single linear map.

Two other architectural mechanisms reduce the $c_{\mathrm{i}}c_{\mathrm{o}}$ factor by restricting which input channels feed into each output channel: *grouped convolutions*, popularized by ResNeXt (Section 7.4.5.1), and *depthwise-separable convolutions*, which underpin lightweight mobile models (Section 7.5.1.1). We will introduce each in the next chapter as we build the architectures that rely on them.

Notice that channels alone leave one basic property unchanged: convolutional outputs still shift pixel-for-pixel whenever the input shifts. In Section 6.5, we introduce pooling layers to summarize local neighborhoods and build tolerance to small spatial shifts.

6.4.4 Summary

To handle multi-channel inputs, a convolutional layer cross-correlates each input channel with its own spatial kernel and sums the results, allowing a single filter to integrate cues across channels such as color. To produce c_{o} output feature maps, the layer stacks c_{o} such multi-input kernels into a tensor of shape $c_{\mathrm{o}} \times c_{\mathrm{i}} \times k_{\mathrm{h}} \times k_{\mathrm{w}}$, incurring $c_{\mathrm{o}}c_{\mathrm{i}}k_{\mathrm{h}}k_{\mathrm{w}} + c_{\mathrm{o}}$ parameters and $h_{\mathrm{o}}w_{\mathrm{o}}c_{\mathrm{o}}c_{\mathrm{i}}k_{\mathrm{h}}k_{\mathrm{w}}$ multiply-adds (both proportional to $c_{\mathrm{i}}c_{\mathrm{o}}$), while training must also store $h_{\mathrm{o}}w_{\mathrm{o}}c_{\mathrm{o}}$ activations per example. A 1×1 convolution acts as a shared fully connected layer at every pixel, letting architectures mix channels and construct parameter- and compute-efficient bottlenecks.

6.4.5 Exercises

1. **Composition of convolutions.** Two convolutional layers with stride 1, no padding, and no nonlinearity in between have kernels of shape $c_{\mathrm{m}} \times c_{\mathrm{i}} \times k_1 \times k_1$ and $c_{\mathrm{o}} \times c_{\mathrm{m}} \times k_2 \times k_2$.
    1. Prove that their composition is a single convolution, and give the shape of its kernel.
    2. Compare the number of weights of the pair with that of the single convolution. For which intermediate width c_{m} is the pair cheaper?
    3. As Section 6.2 showed, three stacked single-channel 3×3 kernels have a 7×7 receptive field. Decide whether every single-channel 7×7 kernel is the composition of three 3×3 kernels, and justify the answer.

2. **Three costs, three layers.** A network for 224×224 RGB images begins with four 3×3 convolutions with padding 1: from 3 to 64 channels at stride 1, from 64 to 128 at stride 2, from 128 to 128 at stride 1, and from 128 to 256 at stride 2.
    1. For each layer give the output shape, the number of weights, the number of multiply-adds per image, and the number of output values that training must keep for the backward pass.
    2. Which layer has the most weights, which performs the most multiply-adds, and which keeps the most output values? Explain why these are three different layers.
    3. The images are replaced by 448×448 ones. Which of the three costs change, and by what factor?
3. code **Multi-channel im2col.** Extend the patch matrix of Section 6.2.3 to several input and output channels. Rebuild the two-channel `X` of Figure 6.16 and the three-output-channel `K` of Section 6.4.2, since later cells of this section overwrite both. Build the patch matrix, whose rows are input patches of shape $c_{\mathrm{i}} \times k_{\mathrm{h}} \times k_{\mathrm{w}}$ flattened in that order, and arrange `K` into a $c_{\mathrm{i}}k_{\mathrm{h}}k_{\mathrm{w}} \times c_{\mathrm{o}}$ kernel matrix whose rows follow the same order. Confirm that their product, rearranged into the shape of the output, reproduces `corr2d_multi_in_out(X, K)`.

    Adapted from Simon Prince, Understanding Deep Learning, Problems 10.6 and 10.7.

6.5 Pooling

Convolutional layers answer local questions (whether an edge sits at pixel (i, j)), whereas a classifier must answer a global question about the whole image. Bridging from local feature maps to a global prediction requires both wide spatial reach (Section 6.3.2.2) and robustness to tiny spatial shifts so a one-pixel camera jitter does not alter the prediction.

A *pooling layer* serves both goals without adding learnable parameters: it slides a window across each channel independently and summarizes every patch by its *maximum* or *average*. Because the maximum over a small block stays unchanged as long as the peak response remains inside that block, local pooling provides *tolerance* to small shifts, while *global pooling* across the entire feature map collapses spatial coordinates into a single channel-wise vector.

6.5.1 Maximum Pooling and Average Pooling

Like a convolutional layer, a pooling operator slides a fixed-shape window (the *pooling window*) across its input and computes a single scalar output at each position. Unlike a convolution, however, a pooling layer has no learnable kernel weights: it applies a fixed deterministic reduction to the elements inside the window. *Maximum pooling* (*max-pooling* for short) returns the largest entry in the window, whereas *average pooling* returns their arithmetic mean.

In Figure 6.18, a 2×2 window slides across a 3×3 input with stride 1, moving from left to right and top to bottom. Because the window fits in two positions along each axis, the output has shape 2×2, with entries given by the maximum of each window:

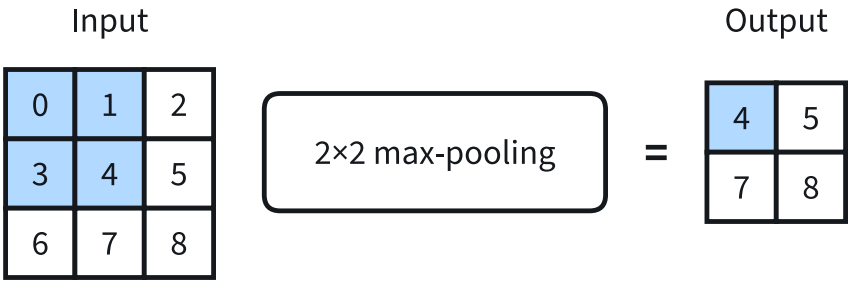


Figure 6.18: Max-pooling with a pooling window shape of 2×2. The shaded portions are the first output element as well as the input tensor elements used for the output computation: $\max(0, 1, 3, 4) = 4$.

$$\max(0, 1, 3, 4) = 4,$$
$$\max(1, 2, 4, 5) = 5,$$
$$\max(3, 4, 6, 7) = 7,$$
$$\max(4, 5, 7, 8) = 8.$$

Notice that five of the nine input entries (0, 1, 2, 3, and 6) are never the maximum of any window. Perturbing them slightly leaves the output completely unchanged, because max-pooling depends only on the winning element in each window. The same sliding-window construction generalizes immediately to any $k_{\mathrm{h}} \times k_{\mathrm{w}}$ pooling window.

Below we implement forward propagation for two-dimensional pooling in `pool2d`. Its nested loop mirrors `corr2d` from Section 6.2, replacing the weighted inner product with either `max` or `mean`. We also include a `stride` argument so we can examine downsampling next.

```
def pool2d(X, pool_size, mode='max', stride=1):
    k_h, k_w = pool_size
    Y = d2l.zeros(((X.shape[0] - k_h) // stride + 1,
                   (X.shape[1] - k_w) // stride + 1))
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            window = X[i * stride: i * stride + k_h,
                       j * stride: j * stride + k_w]
            Y[i, j] = window.max() if mode == 'max' else window.mean()
    return Y
```

Testing `pool2d` on the 3×3 tensor from Figure 6.18 confirms that max-pooling yields 4, 5, 7, and 8.

```
X = d2l.tensor([[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]])
pool2d(X, (2, 2))
```

```
tensor([[4., 5.],
        [7., 8.]])
```

Switching to average pooling on the same input returns the four window means: 2, 3, 5, and 6.

```
pool2d(X, (2, 2), 'avg')
```

```
tensor([[2., 3.],
        [5., 6.]])
```

6.5.2 Tolerance to Small Shifts

Just how much spatial shift can pooling absorb? Let us test the simplest possible setting. A 2×2 pooling layer with stride 2 partitions its input into non-overlapping 2×2 blocks and summarizes each block by a single value. Imagine a 4×4 feature map containing a single nonzero response of 1 (like an edge detector firing at an isolated edge), and slide that response one pixel at a time across row 1. Below we compare the max-pooled and average-pooled outputs across three consecutive column positions.

```
def single_response(i, j, n=4):
    """An n x n feature map that is 1 at (i, j) and 0 elsewhere."""
    return d2l.tensor([[float(r == i and c == j) for c in range(n)]
                       for r in range(n)])

for j in range(3):
    Z = single_response(1, j)
    print(f'response at (1, {j}): max',
          d2l.numpy(pool2d(Z, (2, 2), stride=2)).tolist(),
          'avg', d2l.numpy(pool2d(Z, (2, 2), 'avg', stride=2)).tolist())
```

```
response at (1, 0): max [[1.0, 0.0], [0.0, 0.0]] avg [[0.25, 0.0], [0.0, 0.0]]
response at (1, 1): max [[1.0, 0.0], [0.0, 0.0]] avg [[0.25, 0.0], [0.0, 0.0]]
response at (1, 2): max [[0.0, 1.0], [0.0, 0.0]] avg [[0.0, 0.25], [0.0, 0.0]]
```

Shifting the response from column 0 to column 1 leaves both pooled outputs completely unchanged because both locations fall inside the same 2×2 block. Shifting it one pixel further to column 2 crosses a block boundary, causing the pooled response to jump to the adjacent output cell. Thus, a one-pixel horizontal shift is absorbed whenever the response remains within its block (half of all starting columns) and shifts the output by one position otherwise. Local pooling therefore makes a representation *tolerant* to certain small shifts, not strictly invariant to them. Figure 6.19 illustrates the max-pooled outputs across all four column positions.

Notice also how max-pooling and average pooling summarize the block differently. Max-pooling preserves the full strength of the activation (1) anywhere inside the block, effectively asking whether the feature appears somewhere in that neighborhood. Average pooling outputs 0.25 because the feature occupies only one of the four cells, measuring the feature's spatial coverage and diluting a strong isolated response with its zero neighbors.

Subsampling with stride 2 also introduces a subtle signal-processing hazard whenever a feature crosses a sampling boundary. In the extreme case, taking every second sample of a high-frequency alternating signal $1, 0, 1, 0, \ldots$ produces either all ones or all zeros depending on a single-pixel shift. This distortion, occurring whenever a signal varies faster than the sampling rate can resolve, is called *aliasing*, and it affects every stride-2 downsampler,

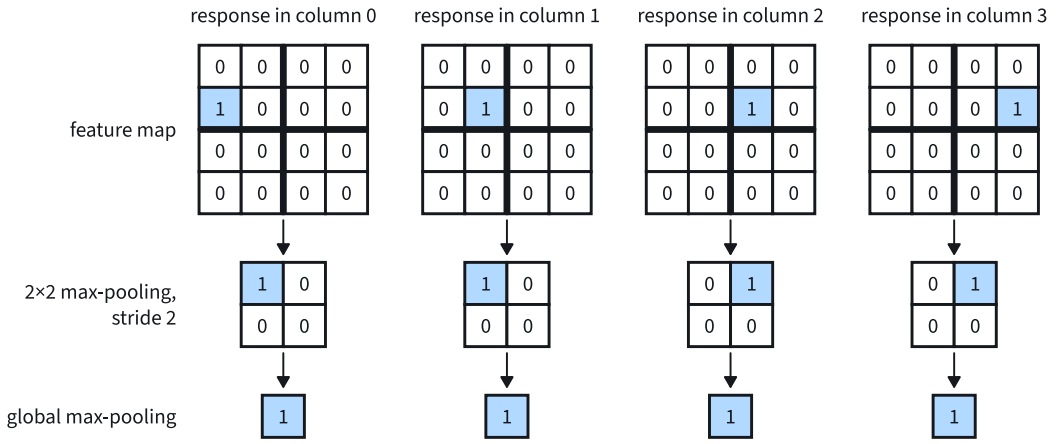


Figure 6.19: A single response moved through the four columns of a 4×4 feature map (top), its 2×2 max-pooled map with stride 2 (middle), and its global maximum (bottom). Heavy lines mark the pooling blocks. The pooled response stays put while the response stays inside its block and jumps when it crosses into the next one; the global maximum never moves.

whether pooled or convolutional. Applying a small smoothing filter before subsampling mitigates aliasing (Zhang 2019), a technique explored in the exercises below.

Historically, shift tolerance was the primary motivation behind pooling. Fukushima's Neocognitron (Fukushima 1980) paired each layer of feature-detecting cells with pooling-like cells that fired whenever a feature appeared anywhere in a local neighborhood. An early version of max-pooling appeared in speech recognition (Yamaguchi et al. 1990), and max-pooling was subsequently proposed in computational neuroscience as a model of hierarchical evidence aggregation in the visual cortex (Riesenhuber and Poggio 1999).

6.5.3 Global Pooling

What about the full translation invariance we desired in Section 6.1 when asking whether Waldo appears *anywhere* in an image? As we just saw, a small fixed window cannot guarantee it. Expanding the pooling window to cover the *entire* spatial extent of each feature map can: the maximum or mean over all spatial positions yields the exact same scalar regardless of where the response sits (bottom row of Figure 6.19). This operation is called *global pooling*, and it collapses each channel to a single number.

Inside a full CNN, however, global pooling sits atop zero-padded convolutional layers, which are strictly shift-equivariant only away from the image borders (Section 6.1). As we saw in Section 6.3.1.3, an interior pixel participates in all nine windows of a 3×3 kernel whereas a border pixel participates in only six (and a corner pixel in four), so while a global maximum over an isolated response stays constant across columns, a global average drops when a feature moves onto the padded border. Stride-1 convolutional stacks ending in global pooling are therefore strictly invariant to shifts that keep salient content away from the boundaries, and approximately invariant otherwise. Strided layers weaken this guarantee: because of the aliasing we met above, the invariance is then exact only for shifts by a multiple of the product of the strides.

In the modern architectures of Chapter 7, the two roles of pooling are frequently separated.

Intermediate spatial downsampling is often handled by stride-2 convolutions, which learn how to weight each window rather than committing to a fixed maximum or average, while the network terminates in *global average pooling* (Lin et al. 2013) before the final linear classifier. LeNet, the classic architecture we build in the next section, predates both conventions.

6.5.4 Padding and Stride

Because pooling slides a window over spatial grids in exactly the same way as convolution, deep learning frameworks parameterize pooling layers with the same window shape, padding, and stride arguments, and the output shape obeys the identical formula from Section 6.3: padding p pixels on each side of an axis of length n with window size k and stride s yields $\lfloor (n + 2p - k)/s \rfloor + 1$ outputs along that axis. The one subtle difference is what *value* fills the padded border. To demonstrate these behaviors with built-in pooling layers, we first construct a four-dimensional tensor holding a single 4×4 feature map with entries 0 through 15.

```
# (example, channel, height, width)
X = d2l.reshape(d2l.arange(16, dtype=d2l.float32), (1, 1, 4, 4))
X
```

```
tensor([[[[ 0.,  1.,  2.,  3.],
          [ 4.,  5.,  6.,  7.],
          [ 8.,  9., 10., 11.],
          [12., 13., 14., 15.]]]])
```

Because pooling is usually used to summarize non-overlapping regions, most frameworks default to setting the stride equal to the pooling window size so the windows tile the input without overlap. On our 4×4 input, a 3×3 window with stride 3 fits only once, returning the maximum of the upper-left 3×3 block (10). Since pooling has no learnable weights, the layer requires no parameter initialization.

```
pool = nn.MaxPool2d(3)  # The stride defaults to the window shape
pool(X)
```

```
tensor([[[[10.]]]])
```

We can override these defaults whenever we want overlapping windows or explicit padding. Setting padding to 1 and stride to 2 produces an output of size $\lfloor (4 + 2 - 3)/2 \rfloor + 1 = 2$ along both height and width.

```
pool = nn.MaxPool2d(3, padding=1, stride=2)
pool(X)
```

```
tensor([[[[ 5.,  7.],
          [13., 15.]]]])
```

What value should fill the padded border of a pooling layer? In a convolution, zero is the natural neutral element because multiplying by zero contributes nothing to a weighted sum. For max-pooling, however, the neutral element is $-\infty$: padded cells can never beat a real activation, ensuring every output equals the maximum over the genuine input entries inside its window. In our example above, the windows cover real rows and columns $\{0, 1\}$ and

$\{1, 2, 3\}$, yielding 5, 7, 13, and 15. Padding with $-\infty$ also preserves shift equivariance in *value*: adding a constant to every input entry shifts every pooled output by the exact same constant. We can verify this on `X - 16`, whose entries are all strictly negative.

```
pool(X - 16)
```

```
tensor([[[[-11.,  -9.],
          [ -3.,  -1.]]]])
```

Every output decreases by exactly 16, as expected. Had the layer padded with zeros instead, the three windows touching the border would have returned 0, corrupting the result simply because 0 happens to be larger than all negative inputs.

Average pooling has no analogous neutral constant. Including padded zeros in the denominator biases border averages toward 0; the cleaner alternative is to divide only by the count of real input entries falling inside each window.

Average pooling here counts the padded zeros by default (`count_include_pad=True`); `count_include_pad=False` averages only the real entries.

6.5.5 Multiple Channels

Whereas a convolutional layer mixes information across all input channels by summing over them (Section 6.4), a pooling layer operates independently on each channel. Consequently, the output always has the exact same number of channels as the input. To verify this behavior, we concatenate `X` and `X + 1` along the channel axis to create a two-channel tensor.

```
X = d2l.concat((X, X + 1), 1)
X
```

```
tensor([[[[ 0.,  1.,  2.,  3.],
          [ 4.,  5.,  6.,  7.],
          [ 8.,  9., 10., 11.],
          [12., 13., 14., 15.]],

         [[ 1.,  2.,  3.,  4.],
          [ 5.,  6.,  7.,  8.],
          [ 9., 10., 11., 12.],
          [13., 14., 15., 16.]]]])
```

Applying our earlier pooling operation (padding 1, stride 2) yields two output channels: the first reproduces 5, 7, 13, and 15, while the second returns the same values incremented by 1.

```
pool(X)
```

```
tensor([[[[ 5.,  7.],
          [13., 15.]],

         [[ 6.,  8.],
          [14., 16.]]]])
```

Because pooling has no learnable weights, it discovers no patterns on its own; it can only summarize the features detected by preceding convolutional layers. With convolutions, padding, stride, dilation, multi-channel feature maps, and pooling all in hand, we now have every building block needed to assemble and train a complete convolutional neural network and compare it directly against our earlier MLPs.

6.5.6 Summary

Pooling layers summarize each spatial window of an input feature map by its maximum or average. They introduce zero learnable parameters, process each channel independently, and follow the exact same padding and stride output-shape rules as convolutions; max-pooling pads borders with $-\infty$ so that padded positions never affect the maximum. A 2×2 pooling layer with stride 2 halves spatial resolution along each axis and absorbs small shifts that stay within a block, while still jumping when a response crosses a block boundary. Max-pooling preserves the peak activation in a neighborhood, whereas average pooling dilutes isolated peaks across the window. Finally, global pooling collapses each entire feature map to a single scalar. On top of stride-1 layers this yields full translation invariance for interior patterns (and approximate invariance near padded borders); with strided layers the invariance is exact only for shifts by a multiple of the product of the strides.

6.5.7 Exercises

1. **Pooling alternatives.**
   1. Do we need a separate minimum-pooling layer, or can it be expressed through an operation already in this section?
   2. *Softmax pooling* with temperature $\tau > 0$ replaces the window entries $x_1, \dots, x_m$ by their softmax-weighted average

$$\sum_{i=1}^{m} x_i \frac{\exp(x_i/\tau)}{\sum_{j=1}^{m} \exp(x_j/\tau)}. \tag{6.21}$$

      Find its limits as $\tau \to 0$ and as $\tau \to \infty$. Give one cost that softmax pooling adds relative to max-pooling.

2. **Aliasing and blur-pool.** Naive stride-2 downsampling keeps every second entry of its input. Measure frequency in cycles per input sample, so that the one-dimensional signals $(1, 0, 1, 0, 1, 0)$ and $(0, 1, 0, 1, 0, 1)$ both have frequency $\frac{1}{2}$.
   1. Apply naive stride-2 downsampling to both signals, which differ by a shift of one position, and compare the two outputs. Why is this behavior called *aliasing*?
   2. Which input frequencies can a stride-2 subsampler represent faithfully?
   3. Blur-pool (Zhang 2019) counteracts aliasing by applying a small averaging filter before subsampling. Work out what blur-pool with a two-tap box filter $(\frac{1}{2}, \frac{1}{2})$ computes on the two signals above. Which pooling operation from this section does it coincide with?

3. code **Backward pass of max-pooling.** Derive the gradient of two-dimensional max-pooling with respect to its input, for a $k_\mathrm{h} \times k_\mathrm{w}$ window and stride equal to the window size, and implement it without automatic differentiation. Verify against the gradient that automatic differentiation computes through the built-in max-pooling operation of this section on a small random tensor, to a relative error below 10^{-4}.

Adapted from Stanford CS231n, Assignment 2, ConvolutionalNetworks, naive max-pooling backward pass.

4. **Max-pooling via ReLU.** ● A convolution is linear, so it cannot compute a maximum by itself.
    1. Express $\max(a, b)$ using only affine operations and $\mathrm{ReLU}(x) = \max(0, x)$.
    2. Use this to implement max-pooling as a stack of convolutions and ReLU layers.
    3. Count the channels per layer and the ReLU layers that your construction uses for a 2×2 window and for a 3×3 window. How does the number of ReLU layers grow with the number of entries in the window?

6.6 LeNet

We now have all the building blocks of a convolutional neural network in hand: convolutional layers (Section 6.2), padding and stride (Section 6.3), multiple channels (Section 6.4), and pooling (Section 6.5). It is time to assemble them into a complete image classifier and put our earlier arguments to the test: does respecting two-dimensional spatial structure actually outperform a standard MLP?

Our baseline is the MLP from Section 4.2, which reaches roughly 87% validation accuracy on Fashion-MNIST after 30 epochs. Whereas the MLP flattens each image into 784 unordered numbers (Section 6.1), a CNN preserves the pixel grid and applies shared local kernels across every spatial position. To compare the two architectures fairly, we train our CNN with the same cross-entropy loss, SGD optimizer, and 30-epoch budget. Because a single accuracy number never tells the whole story, we will also inspect the filters learned by the first convolutional layer and run a controlled experiment on pixel-shuffled images to verify where the CNN's advantage originates.

The architecture we implement is *LeNet*, named after Yann LeCun, who developed it with colleagues at AT&T Bell Labs to recognize handwritten digits (LeCun, Léon Bottou, et al. 1998). It marked the culmination of a decade of research that began with one of the earliest end-to-end convolutional networks trained by backpropagation (LeCun et al. 1989). By 1998, LeNet-5 achieved error rates below 1% on handwritten digits, rivaling the support vector machines that dominated machine learning at the time, and variants of it were deployed commercially to read handwritten amounts on bank checks and ATM deposits. LeNet provided early, compelling proof that an end-to-end learned network could replace carefully hand-engineered vision pipelines.

6.6.1 The LeNet Architecture

At a high level, LeNet divides cleanly into two stages, illustrated in Figure 6.20: a *convolutional encoder* built from two convolutional blocks, followed by a *dense classification head* consisting of three fully connected layers.

Each convolutional block pairs a 5×5 convolutional layer and a sigmoid activation with a 2×2 average-pooling layer of stride 2. The first convolutional layer outputs 6 channels and the second outputs 16.

The encoder output has channels-first shape `(batch_size, channels, height, width)`.

To connect the convolutional encoder to the dense block, we flatten each example's spatial feature maps into a single vector, producing a two-dimensional matrix with one row per

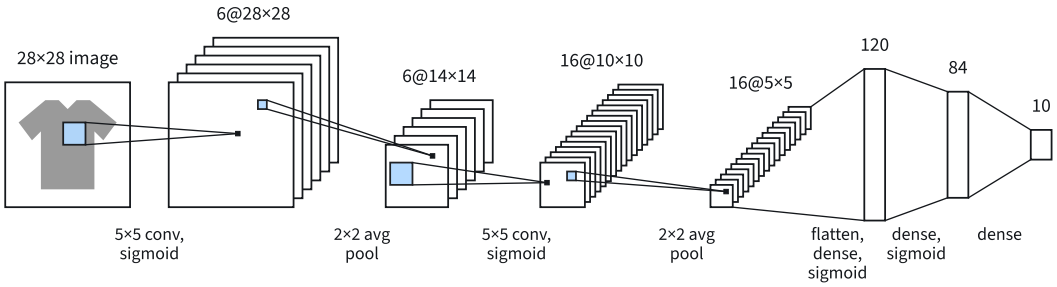


Figure 6.20: Data flow in LeNet on one 28×28 Fashion-MNIST image, with the feature maps drawn to scale. Each shaded window produces the dark unit it points to in the next layer. The three fully connected layers have 120, 84, and 10 outputs.

minibatch example. Three fully connected layers then project this vector through 120 and 84 hidden units to 10 output logits, one per class, just as in softmax regression.

Note that this is the standard pedagogical version of LeNet-5 rather than a bit-for-bit historical replica. It preserves the 1998 network's alternating convolutions and pooling as well as its layer widths (6, 16, 120, and 84), while simplifying legacy details that are rarely used today (such as a scaled tanh activation, learnable pooling scales and biases, sparse channel wiring between the two convolutions, and radial-basis-function output units). One historical detail does remain visible in our code: the original LeNet-5 took 32×32 inputs. MNIST digits are 28×28, and the 1998 paper surrounded them with two background pixels on every side so that strokes near the edge of a digit fall inside the receptive fields of the top feature detectors. Two pixels of padding in our first convolutional layer do the same job, and the first 5×5 convolution again produces a 28×28 output.

Before defining the network, we write `init_cnn` to initialize all convolutional and linear weights with Xavier uniform initialization (Section 4.4.2.1).

```
def init_cnn(module):
    """Initialize weights for CNNs."""
    if type(module) == nn.Linear or type(module) == nn.Conv2d:
        nn.init.xavier_uniform_(module.weight)
```

With modern deep learning frameworks, assembling `LeNet` requires only a single `Sequential` block `self.net` that chains the two convolutional–pooling stages with the three fully connected layers, setting all biases to zero.

```
class LeNet(d2l.Classifier):
    """The layout of LeNet-5, with sigmoid activations and average pooling."""
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(
            nn.LazyConv2d(6, kernel_size=5, padding=2), nn.Sigmoid(),
            nn.AvgPool2d(kernel_size=2, stride=2),
            nn.LazyConv2d(16, kernel_size=5), nn.Sigmoid(),
```

```
            nn.AvgPool2d(kernel_size=2, stride=2),
            nn.Flatten(),
            nn.LazyLinear(120), nn.Sigmoid(),
            nn.LazyLinear(84), nn.Sigmoid(),
            nn.LazyLinear(num_classes))

    def weight_init(self, module):
        # The Trainer calls this after the first minibatch fixed the shapes
        init_cnn(module)  # Xavier-uniform weights
        if type(module) in (nn.Linear, nn.Conv2d) and module.bias is not None:
            nn.init.zeros_(module.bias)
```

6.6.2 Layer Shapes

Because a single mismatched padding value or a forgotten flatten layer shifts every downstream tensor shape, it is good practice to verify intermediate shapes before starting training. Figure 6.21 diagrams LeNet in compressed notation alongside the expected tensor shape at each boundary. To check that our implementation matches this diagram, we pass a single-channel 28×28 dummy tensor through the model and print the output shape after each layer. We package this diagnostic as a `layer_summary` method on `d2l.Classifier` so that we can reuse it for deeper architectures in the next chapter.

```
@d2l.add_to_class(d2l.Classifier)
def layer_summary(self, X_shape):
    X = d2l.randn(*X_shape)
    for layer in self.net:
        X = layer(X)
        name = getattr(layer, '__name__', type(layer).__name__)
        print(f'{name:<22} {tuple(X.shape)}')

model = LeNet()
model.layer_summary((1, 1, 28, 28))
```

```
Conv2d                 (1, 6, 28, 28)
Sigmoid                (1, 6, 28, 28)
AvgPool2d              (1, 6, 14, 14)
Conv2d                 (1, 16, 10, 10)
Sigmoid                (1, 16, 10, 10)
AvgPool2d              (1, 16, 5, 5)
Flatten                (1, 400)
Linear                 (1, 120)
Sigmoid                (1, 120)
Linear                 (1, 84)
Sigmoid                (1, 84)
Linear                 (1, 10)
```

The printed shapes match Figure 6.21 exactly and follow our output-shape rules from Section 6.3. An unpadded 5×5 kernel reduces height and width by 4: our first convolutional layer adds 2 pixels of padding per side to preserve the 28×28 resolution, whereas the second uses no padding and shrinks 14×14 to 10×10. Each 2×2 average-pooling layer halves both spatial dimensions ($28 \to 14$ and $10 \to 5$). Meanwhile, channel depth expands

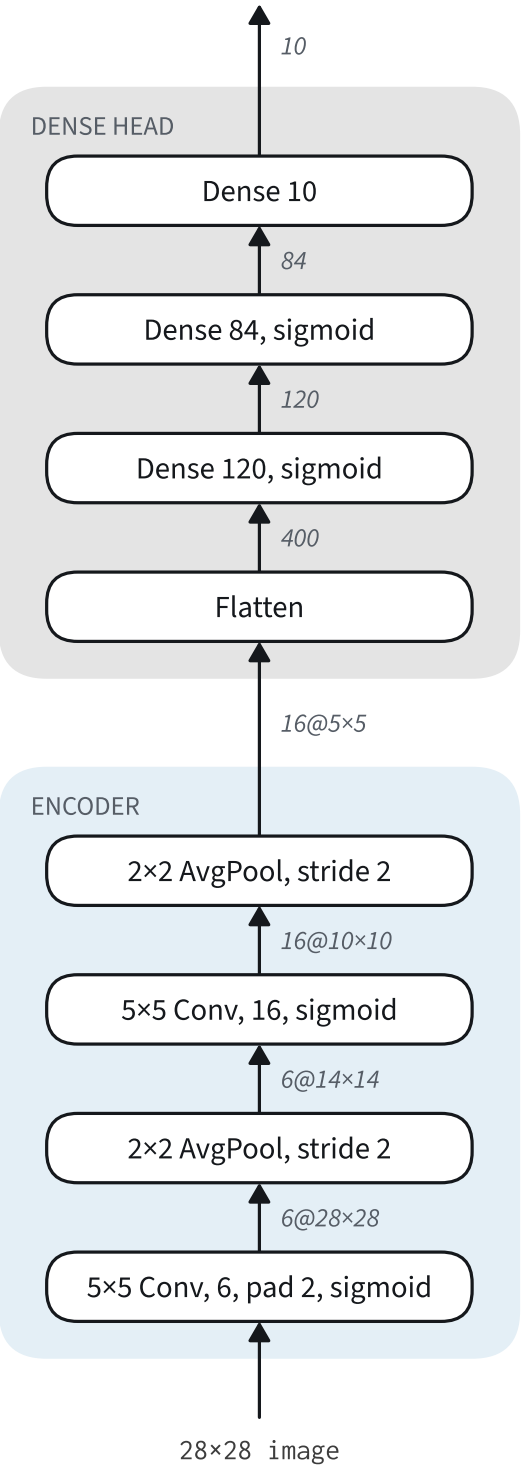


Figure 6.21: Compressed notation for LeNet. Beside each arrow is the shape of the tensor it carries, as channels@height×width.

from 1 to 6 to 16 as the encoder trades spatial resolution for richer feature representations. Flattening the final $16 \times 5 \times 5$ tensor yields a 400-dimensional vector, which the dense block progressively compresses down to 10 class logits.

6.6.3 Parameters and Multiply-Adds

With every layer's output shape verified, we can tally both parameter memory and arithmetic cost. Recall from Section 6.4.2 that a convolutional layer with c_{i} input channels, c_{o} output channels, and a $k \times k$ kernel stores $c_{\text{i}} c_{\text{o}} k^2$ weights plus c_{o} biases, and reuses every weight once per output spatial location. By contrast, a fully connected layer uses each weight only once per example. Table 6.1 breaks down these figures across LeNet.

Table 6.1: Parameters and multiply-adds per example of the layers of LeNet. Pooling and activation layers have no parameters.

layer	output	parameters	multiply-adds
5×5 convolution, 1 to 6 channels	6@28×28	$150 + 6 = 156$	$150 \times 784 = 117{,}600$
5×5 convolution, 6 to 16 channels	16@10×10	$2{,}400 + 16 = 2{,}416$	$2{,}400 \times 100 = 240{,}000$
fully connected, 400 to 120	120	$48{,}000 + 120 = 48{,}120$	$48{,}000$
fully connected, 120 to 84	84	$10{,}080 + 84 = 10{,}164$	$10{,}080$
fully connected, 84 to 10	10	$840 + 10 = 850$	840
total		$61{,}706$	$416{,}520$

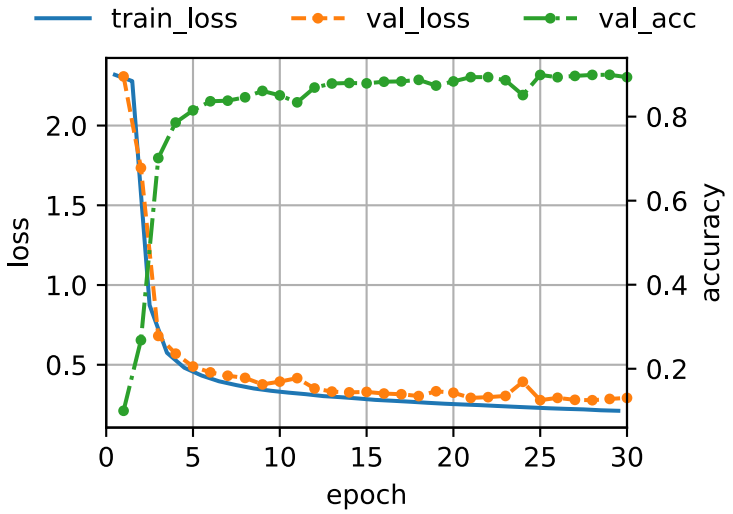
The breakdown highlights a striking division between memory and computation that recurs throughout CNN design. The two convolutional layers account for just 4% of LeNet's parameters yet perform 86% of its multiply-adds, because every kernel weight is evaluated across an entire spatial grid. Conversely, the first fully connected layer alone holds 78% of the parameters while contributing only 12% of the arithmetic. Compared with our earlier Fashion-MNIST MLP (203,530 parameters and roughly 203,000 multiply-adds per image), LeNet uses fewer than a third as many parameters while performing roughly twice as much computation.

6.6.4 Training

Just as with the MLP, we train LeNet by minimizing cross-entropy loss with minibatch SGD for 30 epochs, using a minibatch size of 128 (half the MLP's 256). One hyperparameter, however, must change substantially: the learning rate. Because the derivative of the sigmoid activation never exceeds $1/4$ (Section 4.1.4), gradients flowing back to the first convolutional layer are attenuated by four successive sigmoids. At the MLP's learning rate of 0.1, training crawls: validation accuracy lingers near chance for the first two to four epochs and climbs only to about 73% after 10 epochs. We therefore raise the learning rate to 2.0 (an empirically tuned factor of twenty, reflecting the smaller gradient scale).

As in Section 5.7.4, `d2l.Trainer` automatically dispatches training to a GPU when one is available.

```
trainer = d2l.Trainer(max_epochs=30, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128)
model = LeNet(lr=2.0)
trainer.fit(model, data)
```



After 30 epochs, LeNet achieves roughly 88–90% validation accuracy, improving on the MLP's ~87% despite having less than a third as many parameters.

Look closely at the beginning of the training curve, too. Depending purely on the random initialization, accuracy sometimes climbs immediately, while in other runs it hovers at chance (10%) with the loss pinned near $\ln 10 \approx 2.3$ for a few epochs before suddenly taking off. During that initial plateau, the network predicts nearly identical uniform probabilities for every image: after passing through four sigmoid layers, input-dependent variations in the logits are initially too tiny to separate the classes. We quantify this vanishing-signal effect in an exercise below, and we will see the same plateau resurface in our pixel-shuffling experiment.

6.6.4.1 The Learned Features

For softmax regression, we could visualize each class's weights directly as a 28×28 template image (Section 3.3), and similarly each hidden unit of an MLP weighted fixed pixel locations (Section 4.2). A convolutional kernel, by contrast, is not tied to any particular pixel: it is just a 5×5 patch of weights that is hard to interpret in isolation. To understand what the first convolutional layer has discovered, we first pull one sneaker (class 7) and one shirt (class 6) from a validation minibatch into a two-image tensor `X`.

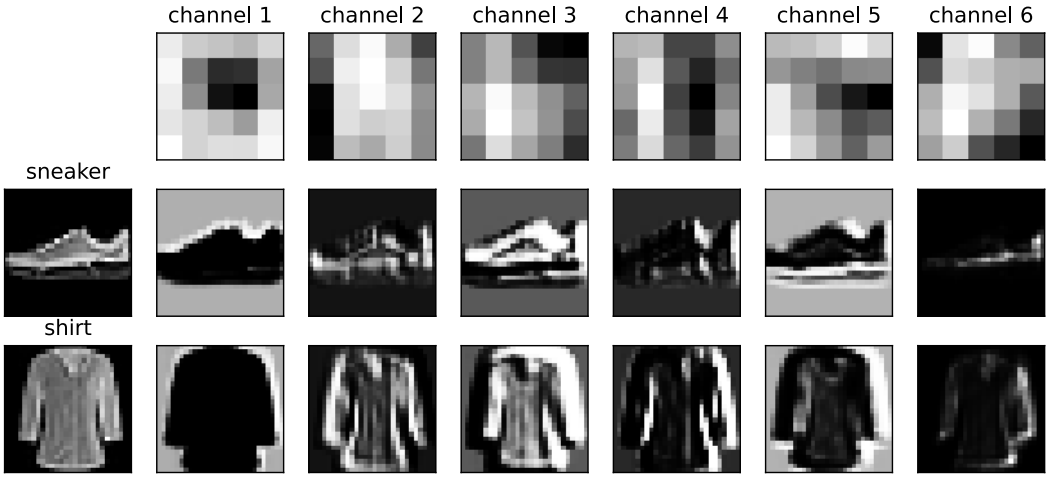
```
X, y = trainer.prepare_batch(next(iter(data.val_dataloader())))
labels = d2l.numpy(y)
X = d2l.stack([X[int(np.argmax(labels == c))] for c in (7, 6)], 0)
```

Next, we pass `X` through the first convolutional layer and its sigmoid activation to extract the six 28×28 feature `maps` alongside the learned 5×5 `kernels`.

```
maps = model.net[1](model.net[0](X))  # The first convolution and its sigmoid
images, maps = d2l.numpy(X[:, 0]), d2l.numpy(maps)
kernels = d2l.numpy(model.net[0].weight[:, 0])  # Six 5x5 kernels
```

We can now display the six `kernels` on the top row above the input `images` and their six post-sigmoid feature `maps`, scaling each panel independently from its minimum (black) to its maximum (white).

```
names = data.text_labels([7, 6])
axes = d2l.show_images(
    [kernels[0], *kernels, images[0], *maps[0], images[1], *maps[1]], 3, 7,
    titles=['', *[f'channel {c}' for c in range(1, 7)],
            names[0], *[''] * 6, names[1], *[''] * 6], cmap='gray')
axes[0].set_visible(False)  # No kernel above the input images
```



Across training runs, two distinct families of feature maps consistently emerge. Some channels act as local brightness or foreground detectors, reproducing the filled silhouette of the garment (either bright on dark or inverted). The remaining channels light up selectively along the garment's contours, especially its left or right boundaries. Examining their 5×5 kernels reveals how they do it: bright columns paired beside dark columns, forming learned, multi-row relatives of the hand-crafted $(1, -1)$ vertical edge kernel from Section 6.2.2.2. Earlier, we had to supply an explicit target edge map for gradient descent to learn an edge filter; here, given nothing but raw garment pixels and class labels, LeNet discovers on its own that edges and silhouettes are useful building blocks for visual classification. (The exact assignment of roles across the six channels varies from run to run.)

6.6.4.2 Shuffled Pixels

How much weight should we put on a 1–3 percentage point gain over the MLP? Taken alone, that margin could be noisy: the MLP's validation accuracy fluctuates by roughly a point from epoch to epoch (Section 4.2), and LeNet also differs in depth, activation function, and learning rate. A cleaner controlled experiment strips away the single piece of information that only convolutions exploit: spatial adjacency. Returning to the thought experiment from

the chapter introduction, we draw one fixed random permutation of the 784 pixel positions and apply that exact same reordering to every training and validation image.

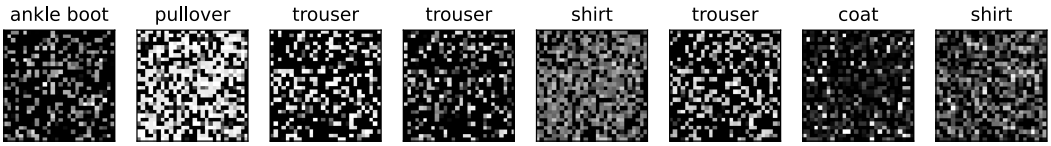
```
perm = np.random.permutation(28 * 28)  # One fixed order for every image

def shuffle_pixels(images):
    """Rearrange the 784 pixels of every image into the order perm."""
    return images.reshape(len(images), -1)[:, perm].reshape(images.shape)
```

We now apply `shuffle_pixels` with this shared permutation `perm` to every image in the training and validation splits of `data_shuffled`.

```
data_shuffled = d2l.FashionMNIST(batch_size=128)
for split in (data_shuffled.train, data_shuffled.val):
    split.data = shuffle_pixels(split.data)  # The stored 8-bit images
```

Visualizing a validation minibatch from `data_shuffled` confirms that permuting the 784 pixel coordinates destroys all recognizable garment contours, leaving patterns that look like pure static to human eyes.

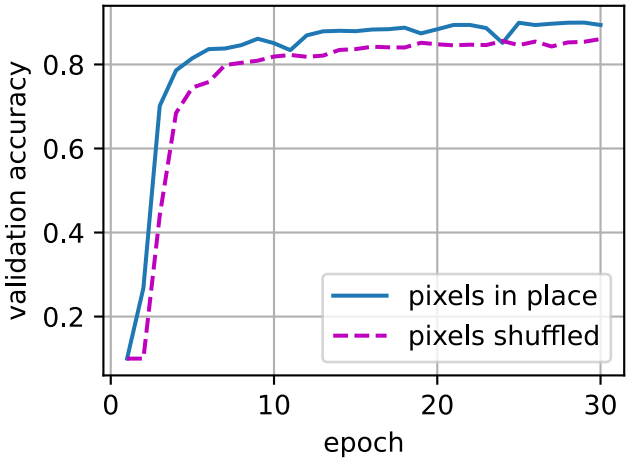
```
data_shuffled.visualize(next(iter(data_shuffled.val_dataloader())))
```



An MLP is completely blind to this permutation, and we do not even need to retrain it to prove why: permuting the input coordinates merely reorders the columns of its first weight matrix $\mathbf{W}^{(1)}$. Since every weight is initialized identically and independently, training on shuffled pixels is statistically identical to training on the original images, yielding the same ~87% accuracy. For LeNet, however, shuffling destroys spatial locality: every 5×5 kernel now reads 25 pixels scattered randomly across the original garment. Let us retrain LeNet with the exact same settings on the shuffled dataset and plot both validation curves side by side.

```
curves = {'pixels in place': trainer.history['val_acc']}
trainer.board.display = False
trainer.fit(LeNet(lr=2.0), data_shuffled)
curves['pixels shuffled'] = trainer.history['val_acc']
d2l.plot(list(range(1, trainer.max_epochs + 1)), list(curves.values()),
         'epoch', 'validation accuracy', legend=list(curves))
for name, acc in curves.items():  # A single epoch can land a point low
    print(f'{name}: {sum(acc[-5:]) / 5:.3f} averaged over the last 5 epochs')
```

```
pixels in place: 0.897 averaged over the last 5 epochs
pixels shuffled: 0.853 averaged over the last 5 epochs
```



On shuffled pixels, LeNet drops several percentage points to around 85–86% in a typical run, at or slightly below the MLP. Once spatial neighborhood structure is destroyed, LeNet's advantage vanishes, confirming that its edge over the MLP comes directly from its convolutional inductive bias.

Notice that the initial chance-level plateau can reappear here as well, and just as unpredictably. While it rarely exceeds a few epochs on ordered images, on shuffled images an unlucky initialization can occasionally leave the model stuck near chance for much of the 30-epoch run, depressing final accuracy even further. That instability stems from stacking four sigmoid activations rather than from the pixel ordering itself; even in runs that escape the plateau immediately, LeNet on shuffled pixels never surpasses the MLP.

Finally, let us connect LeNet back to the symmetry principles of Section 6.1. Idealized image classifiers would have strictly translation-equivariant feature layers and a strictly translation-invariant output, yet LeNet achieves neither exactly: padded convolutions break equivariance near borders, pooling absorbs only shifts that stay within a pooling block (Section 6.5), and flattening the final 5×5 maps assigns distinct dense weights to every spatial cell. LeNet's shift tolerance is therefore only partial: sufficient for Fashion-MNIST, where every garment is neatly centered, though inadequate for unconstrained scenes.

Taken together, our two experiments demonstrate that a CNN can match and outperform an MLP using less than a third as many parameters, precisely because weight sharing lets each kernel learn a local visual pattern once and detect it anywhere on the grid. On tiny 28×28 images, that efficiency is a modest bonus; on the megapixel photographs discussed in Section 6.1, where a single dense layer demands a billion weights, the same principle determines whether a model can be built at all. Making deep convolutional networks *easy to train* at scale is the next challenge, and the sluggish, erratic convergence of LeNet's four sigmoid layers already hints at why modern architectures had to evolve far beyond 1998.

6.6.5 Summary

LeNet pairs a convolutional encoder (which progressively trades spatial resolution for deeper channel representations) with a three-layer fully connected classifier head. Its convolutional layers hold just 4% of the parameters yet perform 86% of the multiply-adds, whereas the first dense layer alone accounts for 78% of the parameters. Trained under

the same 30-epoch SGD setup as the MLP of Section 4.2, LeNet reaches ~88–90% validation accuracy on Fashion-MNIST (versus ~87% for the MLP) with fewer than a third of the parameters, automatically learning silhouette and vertical-edge filters from class labels alone. Shuffling pixel positions erases that gain, isolating spatial locality as its source. At the same time, propagating gradients through four sigmoid layers forces a much larger learning rate (2.0 rather than 0.1) and causes unpredictable early training plateaus.

While the overall encoder-plus-head blueprint remains foundational, almost every individual component of LeNet has since been modernized: ReLUs replaced sigmoids, max-pooling and strided convolutions replaced average pooling, global average pooling followed by a single linear layer replaced heavy dense heads, and normalization layers stabilized deep training. In the next chapter, beginning with AlexNet (Section 7.1), we trace these architectural advances one by one as networks scale up in depth, width, and resolution.

6.6.6 Exercises

1. code **Modern components.** Retrain `LeNet` with the changes below, using the training settings of this section, and compare the validation accuracy of each variant after 30 epochs with that of `LeNet`.
    1. Replace average pooling with max-pooling.
    2. Replace the sigmoid activations with ReLU. If the loss diverges or the accuracy stays at chance, lower the learning rate until training succeeds, and report the learning rate you used.
    3. Make both changes together, and call the result the modernized LeNet. Which of the two changes accounts for most of the difference in accuracy?
    4. ● With the weights the model has at the start of training, measure for `LeNet` and for the modernized LeNet the norm of the gradient of the loss with respect to the first convolutional kernel, and the standard deviation of the logits across a mini-batch. The derivation of Xavier initialization in Section 4.4.2.1 assumes inputs with zero mean and no nonlinearity. Use the mean of the sigmoid's output and its slope $\sigma'(0) = 1/4$ to explain your measurements, the learning rate that `LeNet` needs, and the epochs at chance at the start of its training.
2. code **Original MNIST.** Write an `MNIST` data class by adapting `d21.FashionMNIST`, and train both `LeNet` and the MLP of Section 4.2 on it, with the settings that each uses for Fashion-MNIST. LeCun, Léon Bottou, et al. (1998) report a test error rate below 1% for LeNet-5 on MNIST.
    1. Compare the validation error rate of `LeNet` with this figure. The two are measured on the same images: as for Fashion-MNIST, our validation set is the dataset's official test set.
    2. For each dataset, compute the difference and the ratio of the validation error rates of the two networks. Section 3.2 states that MNIST makes differences among model classes less visible. Does each of the two measures support this statement?
3. code **Deeper feature maps.** The text drew the feature maps of the first convolutional layer.
    1. Display the 16 feature maps of the second convolutional layer of the trained `LeNet`, after its sigmoid, for validation images of two classes that are easily confused,

for example pullovers and coats. Can you still say what a channel responds to? Which channels differ most between the two classes?

2. Train the modernized LeNet of problem 1 and measure the maximum activation of each of its two convolutional layers, taken after the ReLU, on Fashion-MNIST validation images, on out-of-distribution photos such as a cat or a car, and on uniform random noise. Convert each photo to a 28×28 grayscale image, and keep every input in $[0, 1]$, the pixel range of `d2l.FashionMNIST`. Do the maxima for photos and noise fall inside or clearly outside the range observed on the validation images?

4. code **Shifted garments.** The text argued that `LeNet` is only partly tolerant to shifts.

   1. Shift every validation image by one, two, and four pixels to the right, filling the vacated columns with zeros, and measure the validation accuracy of the trained `LeNet` on each shifted set. Which shifts does the network tolerate? Relate what you find to the two pooling layers and to the dense head.
   2. Explain why training `LeNet` on translated copies of its training images can improve the accuracy on shifted validation images even though its convolutional layers share weights across all locations.

   Adapted from Michael Nielsen, Neural Networks and Deep Learning, Chapter 6.

5. code **The dense head.** ● Table 6.1 shows that the first fully connected layer holds 78% of LeNet's parameters.

   1. Replace the entire dense head, from the flatten through the output layer, by global average pooling followed by a single linear layer from 16 to 10 units. How many parameters does the network now have? Before training, predict whether the validation accuracy changes by more than one percentage point relative to `LeNet`; then retrain and compare.
   2. Instead connect the flattened 400-dimensional convolutional output directly to a ten-way linear output layer. How many parameters does this network have? Retrain, and compare the accuracy of the two reduced heads with each other and with your prediction. What does the comparison say about where LeNet's capacity is used?

   Adapted from Michael Nielsen, Neural Networks and Deep Learning, Chapter 6.

Chapter 7

Modern Convolutional Neural Networks

With the mechanics of convolutional layers and LeNet in hand (Chapter 6), we now trace a decade of convolutional architecture design, from AlexNet's 2012 ImageNet breakthrough to the early-2020s dialogue between convolutional networks and Vision Transformers. Rather than treating each landmark model as an isolated artifact, we ask what bottleneck prompted each design step and carry a cumulative toolkit forward across the chapter.

The opening sections move from hand-wired layers to modular building blocks. AlexNet (Section 7.1) demonstrates how end-to-end feature learning overtook hand-engineered vision pipelines once large labeled datasets and GPUs converged, introducing ReLU activations, dropout, data augmentation, and a warmup-and-cosine learning-rate schedule reused throughout the chapter, while exposing a stark imbalance between convolutional compute and dense-head parameter count. VGG, Network in Network (NiN), and GoogLeNet (Section 7.2) address that imbalance by organizing networks into repeatable 3×3 blocks, mixing channels per pixel with 1×1 convolutions, replacing massive dense heads with global average pooling, and combining parallel multi-scale branches inside a stem–body–head layout.

The middle sections scale depth and hardware efficiency. Normalization layers (Section 7.3) standardize intermediate activations across the minibatch or within individual examples, rendering hidden layers insensitive to weight scale and widening usable learning rates by orders of magnitude. Residual networks (Section 7.4) resolve the optimization degradation of deep plain stacks through additive shortcuts $\mathbf{x} + g(\mathbf{x})$ that keep the identity mapping at $g = 0$, before extending the block via grouped convolutions (ResNeXt) and feature concatenation (DenseNet). Efficient convnets (Section 7.5) elevate latency, memory, and energy alongside accuracy through depthwise-separable factorization, inverted bottlenecks, compound scaling, squeeze-and-excitation gates, and structural re-parameterization, while showing why operation counts often diverge from GPU wall-clock time.

The closing sections disentangle training methodology from architecture. Modern training recipes (Section 7.6) show that contemporary optimization and regularization—AdamW, cosine warmup, label smoothing, Mixup, CutMix, stochastic depth, and weight averaging—account for several percentage points of ImageNet accuracy on an unchanged ResNet. Holding such a recipe fixed, ConvNeXt (Section 7.7) modernizes a ResNet step by step to match

hierarchical Vision Transformers without attention, and convolutional design spaces (Section 7.8) compare empirical error distributions across sampled populations to distill the simple structural rules of the RegNet family.

7.1 AlexNet and Learned Image Representations

In Section 6.6, LeNet trained every stage of its pipeline, from convolutional filters to the final classifier, end to end from raw pixels. Yet for two decades after LeCun's 1989 results, mainstream computer vision split feature extraction from classification: hand-engineered descriptors such as SIFT (Lowe 2004), SURF (Bay et al. 2006), HOG (Dalal and Triggs 2005), and bags of visual words (Sivic and Zisserman 2003) mapped pixels into fixed feature vectors that fed a linear or kernel classifier. Because the training loss updated only the classifier and never the features, progress relied on human feature engineering.

The watershed came at the 2012 ImageNet challenge (1,000 object categories evaluated by *top-5 error*, the fraction of validation images whose true label falls outside the model's top five predictions). *AlexNet* (Krizhevsky et al. 2012) trained an eight-layer convolutional network end to end on raw pixels. Its competition entry, an ensemble of seven such networks, cut top-5 error from 26.2% (for the best hand-engineered pipeline) to 15.3% in a single leap, and one network alone already reached about 18%. Architecturally, AlexNet is LeNet scaled up, so its sudden triumph points straight to three catalysts that converged in 2012: a million-image labeled dataset (ImageNet), GPUs fast enough to train tens of millions of weights in days rather than months, and key modeling practices (ReLU activations, dropout, and data augmentation) that made deep networks trainable and resistant to memorization.

7.1.1 Learned Features, Data, and Hardware

What did backpropagation discover on ImageNet that hand-crafted descriptors missed? Figure 7.1 shows the 96 learned 11×11 kernels of AlexNet's first convolutional layer: oriented edge and frequency detectors alongside color-opponent blobs, found purely by minimizing classification loss and composed by deeper layers into textures, parts, and object categories.

7.1.1.1 Data and Hardware

Models with 60 million parameters quickly memorize small benchmarks such as PASCAL VOC (~10^4 images). The 2012 ImageNet training split (Deng et al. 2009; Russakovsky et al. 2015) supplied 1.2 million labeled photographs across 1,000 categories, and AlexNet multiplied effective sample diversity via data augmentation—extracting random 224×224 crops and horizontal reflections from 256×256 images (up to $32 \cdot 32 \cdot 2 = 2{,}048$ views per photograph).

Hardware supplied the second half of the breakthrough. Convolving a filter across every spatial location and channel executes millions of independent multiply-adds—the exact workload graphics processing units (GPUs) were built to run across hundreds of parallel cores connected to high-bandwidth memory. Each of the two NVIDIA GTX 580 GPUs used for AlexNet delivered ~1.6 TFLOPS and 192 GB/s of memory bandwidth, versus ~0.2 TFLOPS and 20 GB/s for a contemporary quad-core CPU. One forward pass of AlexNet on a 224×224 RGB image costs ~10^9 multiply-adds; counting a backward pass at twice the forward pass and 2 FLOPs per multiply-add, 90 epochs over 1.2 million images require

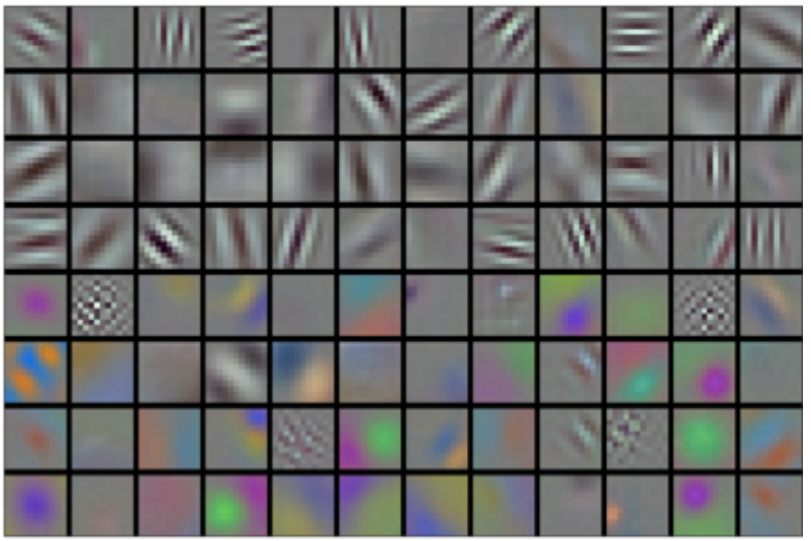


Figure 7.1: The 96 kernels of the first convolutional layer of AlexNet, learned on ImageNet. Reproduction courtesy of Krizhevsky et al. (2012).

$$10^9 \cdot 3 \cdot 2 \cdot 1.2 \times 10^6 \cdot 90 \approx 6.5 \times 10^{17} \text{ FLOPs} \tag{7.1}$$

—five to six days on two GTX 580s using Krizhevsky's custom CUDA kernels (cuda-convnet), compared with well over a month on a CPU.

7.1.2 The AlexNet Architecture

Figure 7.2 compares LeNet (Section 6.6) side by side with the AlexNet variant we train on Fashion-MNIST below. Both follow the same high-level design, a stack of convolutional stages feeding into a dense classification head, but AlexNet scales up every dimension and modernizes three critical components: the activation function, pooling, and regularization.

7.1.2.1 Depth, Width, ReLU, and Dropout

AlexNet expands LeNet from five parameterized layers to eight—five convolutions and three dense layers (Figure 7.2). Because 224×224 inputs are eight times wider than 28×28 thumbnails, layer 1 uses an 11×11 kernel with stride 4, layer 2 uses a 5×5 kernel, and layers 3–5 use 3×3 kernels, with overlapping 3×3 stride-2 max-pooling after the first, second, and fifth convolutions. Channel widths grow from 96 to 384 (vs. 6 and 16 in LeNet), and the two hidden dense layers widen to 4096 units each (vs. 120 and 84). Our implementation omits two historical 2012 artifacts: splitting layers across two 3 GB GPUs and local response normalization (superseded by batch normalization in Section 7.3).

Three modeling choices make this larger network effective. The first two are AlexNet's own; the third is our update of it:

1. **ReLU activations:** Replacing saturating sigmoids (whose derivative $\leq 1/4$ attenuates deep gradients by 4^{-K}) with $\text{ReLU}(x) = \max(x, 0)$ keeps derivatives at 1 on active

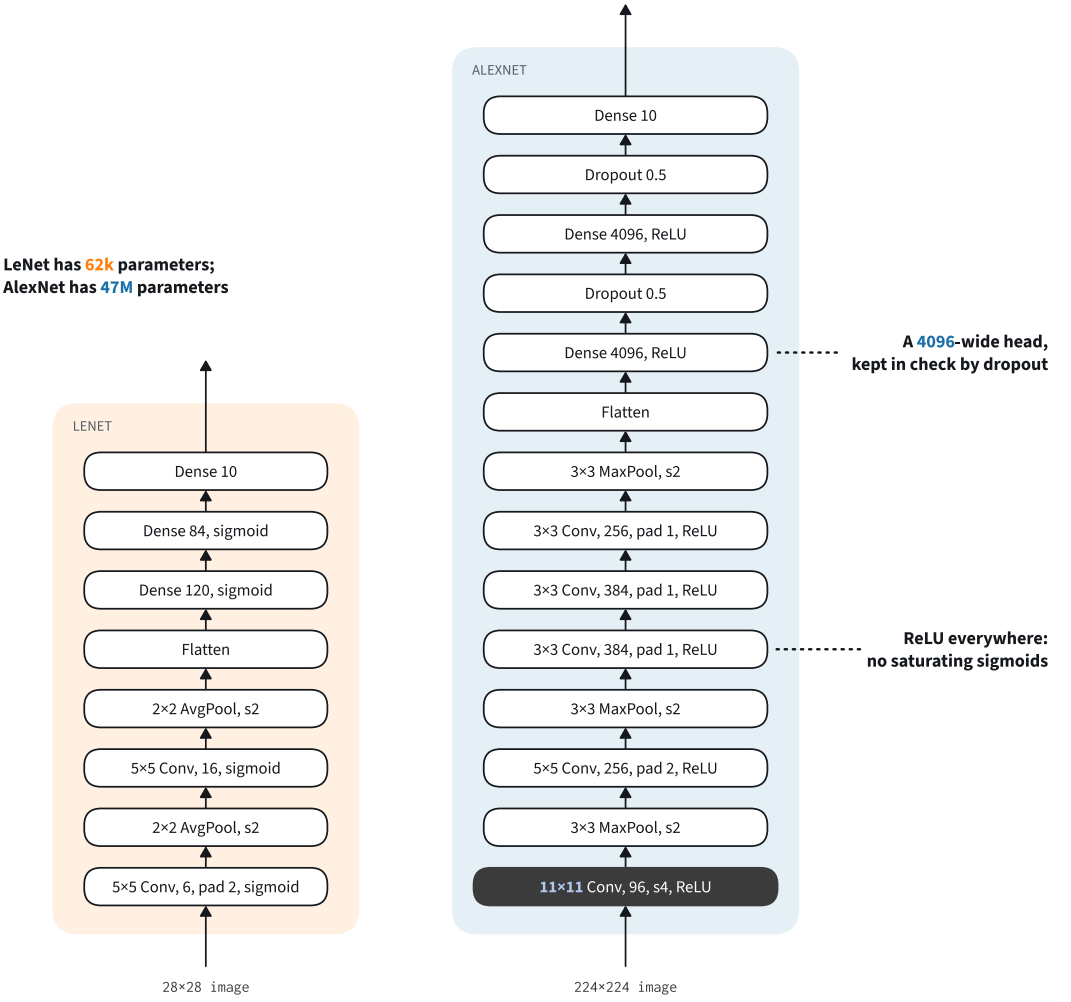


Figure 7.2: LeNet as built in Section 6.6 (left) and AlexNet as built in this section (right), both for single-channel Fashion-MNIST images and ten classes; the input is at the bottom. AlexNet is deeper and wider, uses ReLU activations and max-pooling, and regularizes its dense layers with dropout.

units, reaching 25% CIFAR-10 training error six times faster than tanh (Krizhevsky et al. 2012).

2. **Dropout:** Applying dropout ($p = 0.5$, Section 4.6) after both 4096-unit dense layers prevents the classifier head from memorizing the training set.
3. **He initialization:** AlexNet drew its weights from a Gaussian with a fixed standard deviation of 0.01. We replace it by the initialization that He et al. (2015) derived for ReLU networks three years later. Because ReLU zeroes out roughly half its inputs, we draw weights from $\mathcal{N}(0, 2/n_{\text{in}})$ (Section 4.4.2.2, with $n_{\text{in}} = c_{\text{i}}k^2$ for convolutions) and initialize biases to zero.

The lazy layers have no weights until the first minibatch fixes their shapes. As for LeNet, the `Trainer` then calls the model's `weight_init` on every layer.

```
class AlexNet(d2l.Classifier):
    def __init__(self, lr=0.1, num_steps=4690, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(
            nn.LazyConv2d(96, kernel_size=11, stride=4), nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.LazyConv2d(256, kernel_size=5, padding=2), nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.LazyConv2d(384, kernel_size=3, padding=1), nn.ReLU(),
            nn.LazyConv2d(384, kernel_size=3, padding=1), nn.ReLU(),
            nn.LazyConv2d(256, kernel_size=3, padding=1), nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2), nn.Flatten(),
            nn.LazyLinear(4096), nn.ReLU(), nn.Dropout(p=0.5),
            nn.LazyLinear(4096), nn.ReLU(), nn.Dropout(p=0.5),
            nn.LazyLinear(num_classes))

    def weight_init(self, module):
        # The Trainer calls this after the first minibatch fixed the shapes
        if type(module) in (nn.Linear, nn.Conv2d):
            nn.init.kaiming_normal_(module.weight, nonlinearity='relu')
            nn.init.zeros_(module.bias)
```

To verify that our code matches Figure 7.2, we feed a single-channel 224×224 dummy image through `layer_summary` and inspect each output tensor shape.

```
AlexNet().layer_summary((1, 1, 224, 224))
```

```
Conv2d                  (1, 96, 54, 54)
ReLU                    (1, 96, 54, 54)
MaxPool2d               (1, 96, 26, 26)
Conv2d                  (1, 256, 26, 26)
ReLU                    (1, 256, 26, 26)
MaxPool2d               (1, 256, 12, 12)
Conv2d                  (1, 384, 12, 12)
ReLU                    (1, 384, 12, 12)
Conv2d                  (1, 384, 12, 12)
ReLU                    (1, 384, 12, 12)
Conv2d                  (1, 256, 12, 12)
```

```
ReLU                 (1, 256, 12, 12)
MaxPool2d            (1, 256, 5, 5)
Flatten              (1, 6400)
Linear               (1, 4096)
ReLU                 (1, 4096)
Dropout              (1, 4096)
Linear               (1, 4096)
ReLU                 (1, 4096)
Dropout              (1, 4096)
Linear               (1, 10)
```

Every dimension matches the formulas of Section 6.3. The initial stride-4 convolution maps 224 to $\lfloor (224 - 11)/4 \rfloor + 1 = 54$, and each overlapping 3×3 max-pooling layer with stride 2 maps n to $\lfloor (n - 3)/2 \rfloor + 1$ ($54 \to 26 \to 12 \to 5$). Because the intermediate padded convolutions preserve spatial dimensions, the final convolutional stage flattens into a $256 \times 5 \times 5 = 6400$-element vector for the dense classifier head.

7.1.2.2 Parameters and Multiply-Adds

With the layer shapes confirmed, we can tally parameters and arithmetic cost just as we did for LeNet (Section 6.6): a convolutional layer reuses each of its $c_\text{i} c_\text{o} k^2$ weights at every spatial output position, whereas a fully connected layer multiplies each weight once per example. Table 7.1 summarizes the results for our single-channel, ten-class AlexNet.

Table 7.1: Parameters and multiply-adds per example of the layers of AlexNet, for one single-channel 224×224 image and ten classes. Pooling, activation, and dropout layers have no parameters.

layer	output	parameters	multiply-adds
11×11 convolution, 1 to 96, stride 4	96@54×54	11,712	33.9M
5×5 convolution, 96 to 256	256@26×26	614,656	415.3M
3×3 convolution, 256 to 384	384@12×12	885,120	127.4M
3×3 convolution, 384 to 384	384@12×12	1,327,488	191.1M
3×3 convolution, 384 to 256	256@12×12	884,992	127.4M
fully connected, 6400 to 4096	4096	26,218,496	26.2M
fully connected, 4096 to 4096	4096	16,781,312	16.8M
fully connected, 4096 to 10	10	40,970	0.04M
total		46,764,746	938.1M

The split between parameters and computation is even more lopsided than in LeNet. The five convolutional layers account for only 8% of AlexNet's 47 million parameters yet execute 95% of its multiply-adds. Meanwhile, the three fully connected layers consume 92% of the parameters (roughly 170 MB in 32-bit floating point) while performing just 5% of the arithmetic. Compared with LeNet's 61,706 parameters and 416,520 multiply-adds per image,

AlexNet scales parameters up by roughly $760\times$ and arithmetic by $2{,}250\times$. On ImageNet (3 RGB channels and 1000 classes), the full model reaches roughly 60 million parameters, still heavily concentrated in the same dense layers.

7.1.3 Training on Fashion-MNIST

Training AlexNet on ImageNet takes hours on a modern GPU, and it needs a dataset of about 150 GB and a fast input pipeline to keep that GPU fed. We therefore train it on Fashion-MNIST to keep experiments interactive. Since Fashion-MNIST's native 28×28 images are too small to survive an 11×11 stride-4 first layer and three pooling stages, we upsample them to 224×224 via the `resize` argument of `d2l.FashionMNIST`. In real applications, upsampling 64-fold adds zero visual information while multiplying compute; here we accept that overhead solely so we can run the unmodified AlexNet architecture.

We train with minibatch SGD for 10 epochs (a third of LeNet's 30) using a batch size of 128. Thanks to ReLU activations, gradients flow without sigmoid attenuation, so a much smaller learning rate than LeNet's 2.0 works cleanly.

7.1.3.1 Learning-Rate Schedule

A fixed learning rate is rarely optimal at both ends of training. Early on, when weights are random, an overly aggressive step can destabilize optimization; near the end, a large step keeps the parameters bouncing around a minimum rather than settling into it. AlexNet's original authors handled decay manually, dividing the learning rate by 10 whenever validation error plateaued. Instead, we adopt a smooth, hands-free schedule that has become a modern standard—and we reuse this exact schedule throughout the rest of the chapter so all models can be compared cleanly. Specifically, a linear *warmup* ramps the learning rate from 0 to its peak η_0 over the first t_{w} steps, followed by a half-cosine decay back to 0 at the final step T:

$$\eta_t = \frac{\eta_0}{2}\left(1 + \cos\frac{\pi(t - t_{\mathrm{w}})}{T - t_{\mathrm{w}}}\right) \quad \text{for } t \geq t_{\mathrm{w}}. \tag{7.2}$$

At $t = t_{\mathrm{w}}$, the cosine term equals 1 and $\eta_t = \eta_0$; halfway through the decay it drops to $\eta_0/2$; and at $t = T$ it reaches 0. Warming up gently allows us to use a higher peak learning rate than would be safe from step 1. Here we train AlexNet with peak rate $\eta_0 = 0.03$ (each architecture in Section 7.2 likewise uses the highest peak rate at which it trains reliably). Below, `cosine_warmup` computes the multiplicative schedule factor at step t; we save it in `d2l` for reuse across the chapter and study learning-rate schedules systematically in Section 9.7. Figure 7.3 shows the shape of the schedule.

```
def cosine_warmup(t, num_steps, warmup_steps):
    """Learning-rate factor: linear warmup from 0, then a half-cosine to 0."""
    if t < warmup_steps:
        return t / warmup_steps
    progress = min((t - warmup_steps) / (num_steps - warmup_steps), 1)
    return 0.5 * (1 + math.cos(math.pi * progress))
```

`AlexNet` updates this schedule after every minibatch: `num_steps` specifies the total step count T across the run, and warmup spans the first tenth of training (one epoch out of ten).

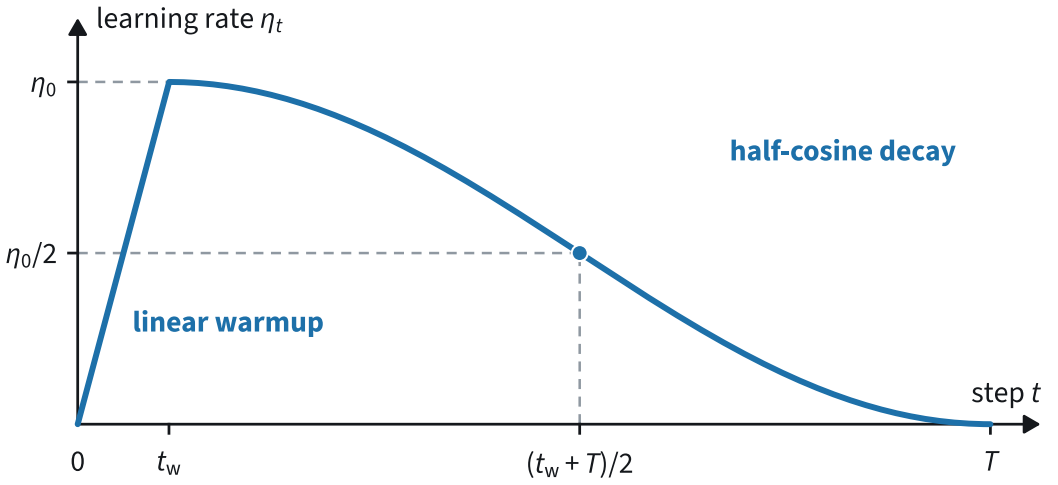


Figure 7.3: The learning-rate schedule used throughout this chapter. A linear warmup raises the rate from 0 to its peak η_0 over the first t_w steps, here a tenth of training. The half-cosine of Equation 7.2 then lowers it to 0 at the final step T, passing $\eta_0/2$ halfway through the decay.

```
@d2l.add_to_class(AlexNet)
def configure_optimizers(self):
    optim = torch.optim.SGD(self.parameters(), lr=self.lr)
    schedule = torch.optim.lr_scheduler.LambdaLR(optim, lambda t: (
        cosine_warmup(t, self.num_steps, self.num_steps // 10)))
    return optim, schedule  # The Trainer steps it after every minibatch
```

7.1.3.2 Comparison with LeNet

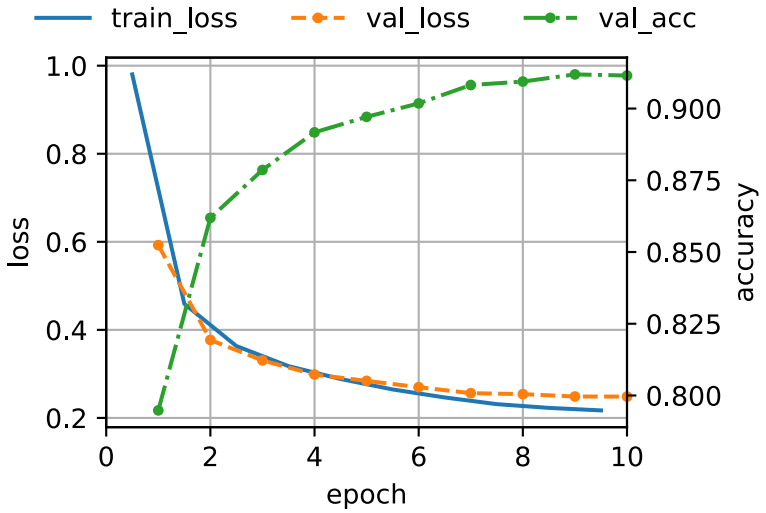
Before running the training loop, it helps to calibrate expectations. Fashion-MNIST has neither the dataset scale nor the visual complexity needed to exercise AlexNet's 47 million parameters, so we should expect only a modest accuracy gain over LeNet—the point here is to verify the implementation, inspect its compute profile, and examine what the first layer learns. To make that inspection possible, we snapshot the initial 96 first-layer kernels before training begins so we can measure exactly how optimization modifies them.

```
def first_kernels(model):
    """The 96 kernels of 11 x 11 weights of the first layer, as NumPy."""
    return d2l.numpy(model.net[0].weight)[:, 0]

data = d2l.FashionMNIST(batch_size=128, resize=(224, 224))
model = AlexNet(lr=0.03, num_steps=10 * len(data.train_dataloader()))
model.apply_init([next(iter(data.train_dataloader()))[0][:1]],
                 model.weight_init)  # Create the weights now
K0 = first_kernels(model)
```

With the initial 11×11 kernels recorded in `K0`, we now fit `AlexNet` on the resized Fashion-

MNIST dataset `data` for 10 epochs.

```
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
trainer.fit(model, data)
```



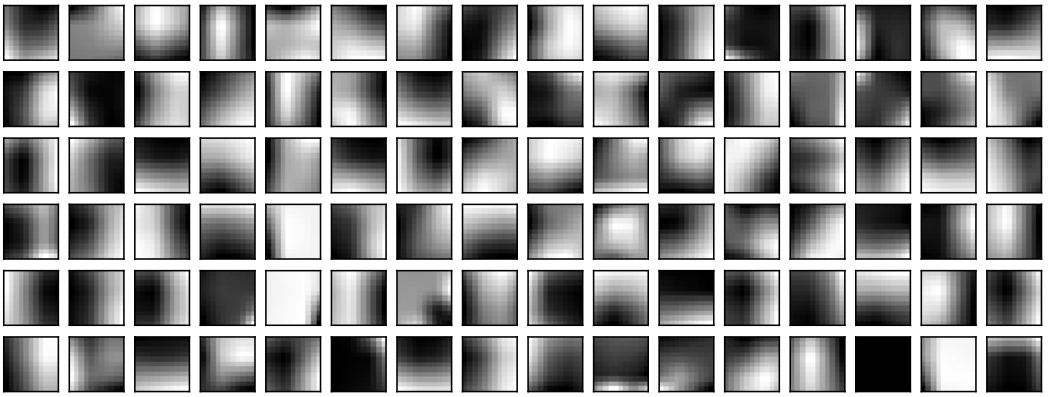
After 10 epochs, AlexNet reaches roughly 91% validation accuracy, with a training loss near 0.22 and a validation loss near 0.25. Because the cosine schedule anneals the learning rate toward zero over the final epochs, accuracy fluctuates by less than a percentage point at the end, making the final epoch a reliable summary of the run. Compared with LeNet's ~88–90% after 30 epochs, AlexNet edges ahead by one or two percentage points in a third as many epochs, at the price of $2{,}250\times$ more arithmetic per image—and even that modest gain reflects its ReLU activations, He initialization, and learning-rate schedule as much as its larger capacity.

7.1.3.3 The Learned First Layer

Whereas LeNet's 5×5 kernels were too tiny to read visually (Section 6.6), AlexNet's 96 kernels of size 11×11 are large enough to interpret directly as small images, as we saw on ImageNet in Figure 7.1. To see what our model learned on Fashion-MNIST, we subtract the initial random weights `K0` from the trained first-layer kernels and plot each weight difference on its own grayscale range.

```
change = first_kernels(model) - K0
d2l.show_images(change, 6, 16, scale=0.6, cmap='gray')
print(f'mean size of the change: {np.abs(change).mean() / np.abs(K0).mean():.3f}'
      ' of the mean size of the initial weights')
```

```
mean size of the change: 0.044 of the mean size of the initial weights
```



Ten epochs shift the first-layer weights by only about 4% of their initial magnitude, so plotting the raw trained kernels would still look mostly like initial random noise. The *weight updates* themselves, however, are distinctly structured: smooth brightness gradients and broad oriented bars—low-frequency counterparts of the crisp ImageNet edge filters in Figure 7.1. Both missing ingredients (fine high-frequency edges and color blobs) follow directly from the input data: our images are grayscale, and eightfold upsampling leaves no genuine detail finer than roughly eight pixels, which an 11×11 kernel perceives as a gentle gradient. In short, the first layer adapts faithfully to the spatial statistics of its dataset.

Finally, recall from Table 7.1 that 92% of AlexNet's parameters reside in its massive fully connected head, which contributes little arithmetic yet requires heavy dropout to curb memorization. In the next section, we ask how to build deep architectures systematically from reusable blocks—and whether that giant dense head is needed at all.

7.1.4 Summary

End-to-end learned visual representations surpassed hand-crafted feature pipelines when three factors aligned in 2012: large-scale labeled datasets (ImageNet), massively parallel GPUs capable of training tens of millions of parameters in days, and modern training components (ReLU activations, dropout, and data augmentation). Scaling up LeNet's basic template with these tools, AlexNet's competition entry slashed ImageNet top-5 error from 26.2% to 15.3%. Accounting for its layers reveals a stark asymmetry: the convolutional trunk executes 95% of the arithmetic while the dense head holds 92% of the parameters. On Fashion-MNIST, which cannot utilize such vast capacity, AlexNet surpasses LeNet by one to two percentage points in a third of the epochs, at roughly $2{,}250\times$ the arithmetic per image.

7.1.5 Exercises

1. **Memory and compute ledger.** Extend Table 7.1 to the memory that `AlexNet` needs in 32-bit floating point.
    1. Compute the number of activations that the forward pass of one 224×224 image stores, layer by layer, counting the output of every convolutional, pooling, and fully connected layer. Where is activation memory concentrated, and where is parameter memory concentrated?
    2. During training, parameters, their gradients, and stored activations for a whole minibatch must fit in memory. At what batch size does activation memory exceed

the memory for parameters and gradients?

3. Repeat the parameter count of Table 7.1 for the ImageNet version (3 input channels, 1000 classes, 227×227 inputs so the final convolutional stage is 6×6 instead of 5×5), and compare your total against the 60 million parameters reported by Krizhevsky et al. (2012) .

2. **Receptive fields of the early layers.** A stack of layers with kernel sizes k_i and strides s_i gives its top unit a receptive field of side $r = 1 + \sum_i (k_i - 1) \prod_{j<i} s_j$ (Section 6.3). Counting each max-pooling layer with its own kernel size and stride, compute the receptive field of one unit after each of the five convolutional layers of `AlexNet`. What fraction of the 224×224 input side does each receptive field span, and does a unit of any convolutional layer see the entire image?

3. **Arithmetic intensity.** ● The *arithmetic intensity* of a layer is the number of multiply-adds it performs per byte of parameters read from memory. Using Table 7.1:

   1. Compute the arithmetic intensity of every layer of `AlexNet` for a batch of 1 image and for a batch of 128 images (assuming each parameter is read once per batch).
   2. Consider an accelerator that performs 10 multiply-adds in the time it reads one byte from memory, where each layer takes the maximum of its compute time and parameter-read time. Does doubling compute throughput or doubling memory bandwidth speed up single-image inference ($B = 1$) more? Which speeds up training at $B = 128$ more?

4. code **Batch size, throughput, and memory.** ● Train `AlexNet` as in this section while varying only the batch size.

   1. Using problem 1, predict the largest training batch size that fits in your GPU's memory.
   2. Starting from batch size 32 and doubling, find the largest batch size that trains without running out of memory and explain the main source of any gap against your prediction.
   3. Measure training throughput (images/second) across at least four batch sizes up to that limit. Beyond which batch size does throughput plateau, and why?

7.2 Blocks, Bottlenecks, and Branches: VGG, NiN, GoogLeNet

AlexNet (Section 7.1) was crafted layer by layer: an 11×11 kernel here, 384 channels there, and max-pooling after the first, second, and fifth convolutions. Hand-tuning every layer is manageable for an eight-layer model, but scaling to twenty or more layers calls for systematic design rules. Equally striking, AlexNet concentrates 92% of its 47 million parameters inside its fully connected classification head—where they contribute only 5% of the arithmetic and rely on heavy dropout to avoid memorizing the training set. Between 2013 and 2014, three landmark architectures tackled these bottlenecks in turn. Even though practitioners rarely deploy these exact models today, the structural ideas they introduced appear in virtually every modern convnet.

First, *VGG* (Simonyan and Zisserman 2015) shifted design from individual layers to repeatable *blocks* of small 3×3 convolutions followed by pooling, turning an entire architecture into a compact list of block specifications. Second, *Network in Network* (NiN) (Lin et al. 2013) eliminated the bloated fully connected head altogether, mixing channels at every pixel via

1×1 convolutions and reducing the final feature maps directly to class scores with global average pooling. Third, *GoogLeNet* (Szegedy et al. 2015) avoided committing to a single kernel size by running multiple filter sizes in parallel inside each block, keeping the larger convolutions affordable through 1×1 channel-reduction bottlenecks. Below we build all three architectures, count their parameters and multiply-adds just as we did for AlexNet, and train them on Fashion-MNIST under the same cosine schedule so we can compare them side by side.

7.2.1 VGG: Blocks as the Unit of Design

7.2.1.1 Stacks of 3×3 Convolutions

If every convolutional layer were followed by stride-2 pooling, an input of width d could support at most $\lfloor \log_2 d \rfloor$ layers before collapsing to 1×1 ($224 \to 112 \to 56 \to 28 \to 14 \to 7 \to 3 \to 1$). Going deeper requires stacking several stride-1, shape-preserving convolutions at each spatial resolution before downsampling.

VGG (Simonyan and Zisserman 2015) standardized on 3×3 kernels—the smallest symmetric filter that captures left/right and up/down neighbors. Two stacked 3×3 layers cover the same 5×5 receptive field as one 5×5 layer using $2 \cdot 9c^2 = 18c^2$ weights instead of $25c^2$; three stacked 3×3 layers cover a 7×7 window with $27c^2$ weights instead of $49c^2$ while inserting three ReLU nonlinearities instead of one. A *VGG block* stacks `num_convs` padded 3×3 convolutions with ReLU, followed by 2×2 stride-2 max-pooling:

```
def vgg_block(num_convs, out_channels):
    layers = []
    for _ in range(num_convs):
        layers.append(nn.LazyConv2d(out_channels, kernel_size=3, padding=1))
        layers.append(nn.ReLU())
    layers.append(nn.MaxPool2d(kernel_size=2, stride=2))
    return nn.Sequential(*layers)
```

7.2.1.2 The VGG Network

A complete VGG network chains a sequence of these blocks together and attaches AlexNet's dense classification head—two 4096-unit hidden layers with dropout followed by the output layer. Figure 7.4 contrasts VGG-11 with AlexNet. Five pooling stages halve a 224×224 input down to 7×7, while channel width doubles from stage to stage until reaching 512.

Because every VGG block is completely specified by two integers (its number of convolutions and its output channel count), the entire convolutional trunk can be passed in as a list of `(num_convs, out_channels)` tuples (`arch`). Rather than defining one rigid architecture, VGG defines a *parameterized family*: VGG-11, VGG-16, and VGG-19 share the exact same class and differ only in `arch`. Treating network topology as modular data rather than hand-wired code is VGG's most enduring legacy. As in Section 7.1, we initialize all networks in this section with He-normal weights and zero biases.

As for AlexNet, the `Trainer` calls `weight_init` on every layer once the first minibatch has fixed the shapes of the lazy layers.

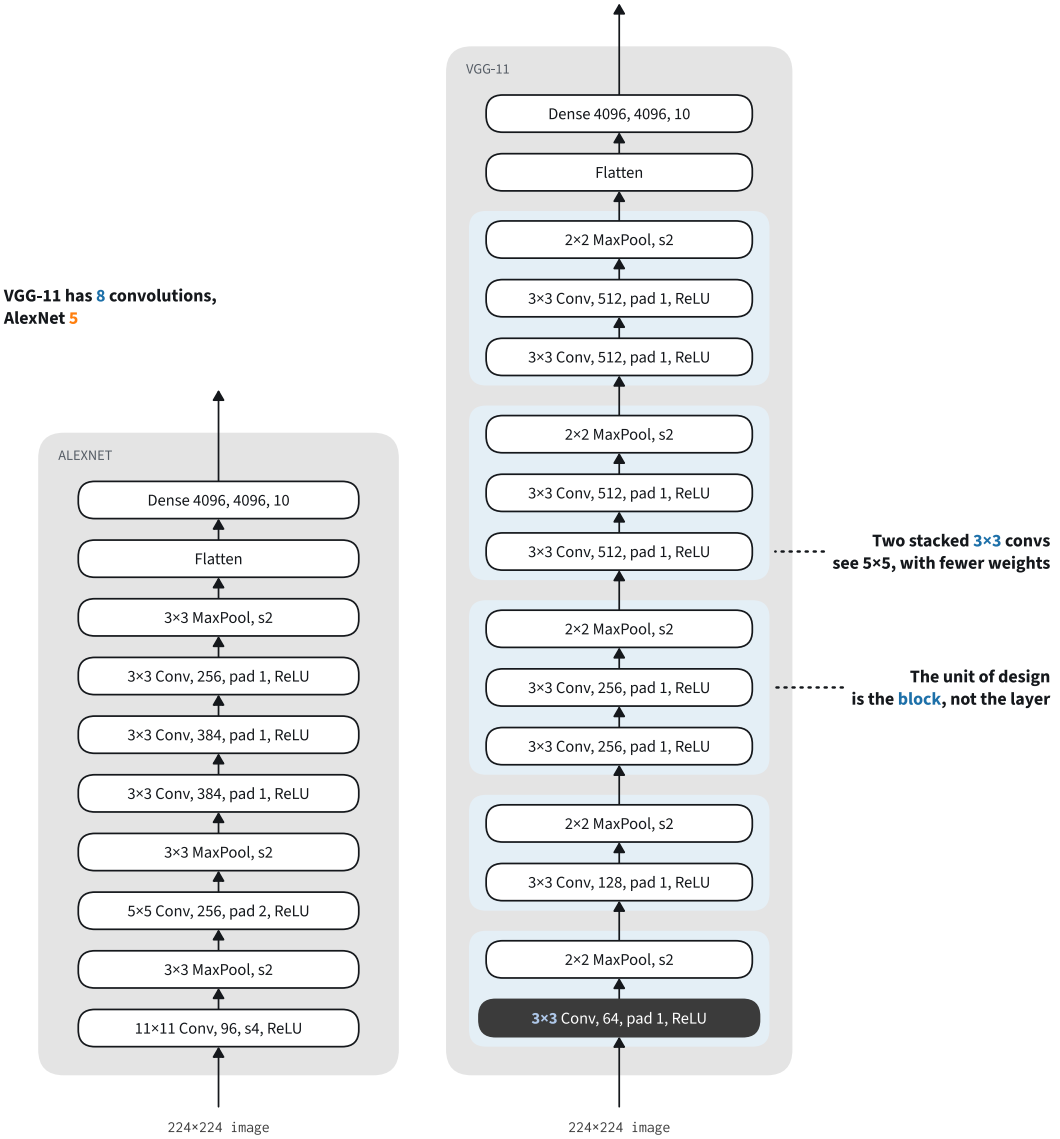


Figure 7.4: AlexNet (left) and VGG-11 (right) for single-channel 224×224 images and ten classes; the input is at the bottom. AlexNet's layers are chosen one by one; VGG repeats one block, several 3×3 convolutions followed by max-pooling, at growing channel counts.

```
class VGG(d2l.Classifier):
    def __init__(self, arch, lr=0.1, num_steps=4690, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        conv_blks = [vgg_block(num_convs, out_channels)
                     for (num_convs, out_channels) in arch]
        self.net = nn.Sequential(*conv_blks, *self.head())

    def head(self):  # AlexNet's fully connected head
        return [nn.Flatten(),
                nn.LazyLinear(4096), nn.ReLU(), nn.Dropout(0.5),
                nn.LazyLinear(4096), nn.ReLU(), nn.Dropout(0.5),
                nn.LazyLinear(self.num_classes)]

    def weight_init(self, module):  # He-normal weights, zero biases
        if type(module) in (nn.Linear, nn.Conv2d):
            nn.init.kaiming_normal_(module.weight, nonlinearity='relu')
            nn.init.zeros_(module.bias)
```

VGG-11, the smallest variant in the original paper, stacks five blocks with (1, 1, 2, 2, 2) convolutional layers and (64, 128, 256, 512, 512) channels—eight convolutional layers plus three dense layers, totaling 11 parameterized layers. Let us inspect the intermediate tensor shapes on a single 224×224 image.

```
VGG(arch=((1, 64), (1, 128), (2, 256), (2, 512), (2, 512))).layer_summary(
    (1, 1, 224, 224))
```

```
Sequential            (1, 64, 112, 112)
Sequential            (1, 128, 56, 56)
Sequential            (1, 256, 28, 28)
Sequential            (1, 512, 14, 14)
Sequential            (1, 512, 7, 7)
Flatten               (1, 25088)
Linear                (1, 4096)
ReLU                  (1, 4096)
Dropout               (1, 4096)
Linear                (1, 4096)
ReLU                  (1, 4096)
Dropout               (1, 4096)
Linear                (1, 10)
```

The final convolutional block outputs a $512 \times 7 \times 7$ tensor, flattening to 25,088 features entering the dense head. Tallying the layers as in AlexNet shows that VGG-11 contains 128.8 million parameters and performs 7.5 billion multiply-adds per image—eight times AlexNet's compute. Once again, parameters and computation live in completely different places: the dense head holds 93% of all weights, with its first $25{,}088 \times 4096$ matrix alone accounting for 103 million parameters.

7.2.1.3 Training

Since running eight times AlexNet's compute is unnecessarily heavy for a Fashion-MNIST demonstration, we train a quarter-width VGG-11 with channel counts 16, 32, 64, 128, and

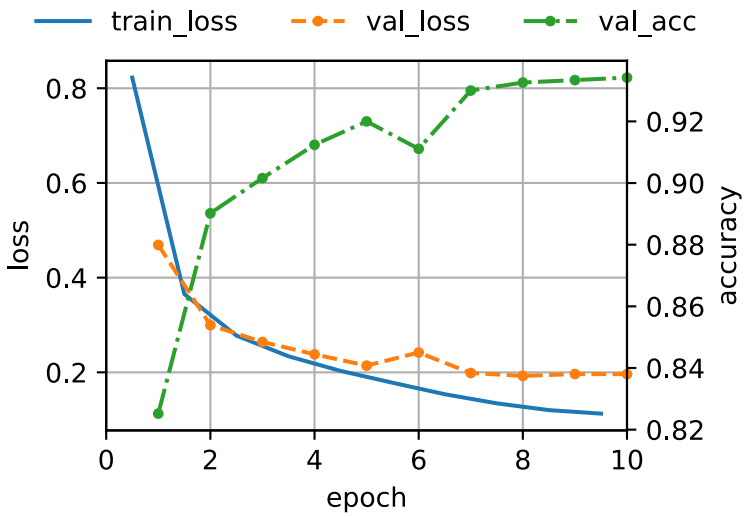
128, which executes 512 million multiply-adds per image—roughly half of AlexNet's 938 million. All other training settings match Section 7.1: 224×224 upsampled images, batch size 128, and 10 epochs of SGD with the warmup-and-cosine schedule of Section 7.1, reusing AlexNet's `configure_optimizers`.

The only hyperparameter that varies across networks in this section is the peak learning rate. At a peak of 0.1, several of the unnormalized architectures below sit dangerously close to instability, occasionally diverging or stalling at chance; those models are trained at 0.03 instead. Our quarter-width VGG trains smoothly at 0.1 and gains 1.5 to 2 percentage points over 0.03, so we run it at 0.1.

```
@d2l.add_to_class(VGG)
def configure_optimizers(self):
    optim = torch.optim.SGD(self.parameters(), lr=self.lr)
    schedule = torch.optim.lr_scheduler.LambdaLR(optim, lambda t: (
        d2l.cosine_warmup(t, self.num_steps, self.num_steps // 10)))
    return optim, schedule  # The Trainer steps it after every minibatch
```

With the cosine warmup schedule attached, we train our quarter-width `VGG` on 224×224 Fashion-MNIST for 10 epochs and store its validation trajectory in `val_acc['VGG']`.

```
data = d2l.FashionMNIST(batch_size=128, resize=(224, 224))
model = VGG(arch=((1, 16), (1, 32), (2, 64), (2, 128), (2, 128)), lr=0.1,
            num_steps=10 * len(data.train_dataloader()))
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
trainer.fit(model, data)
val_acc = {'VGG': trainer.history['val_acc']}  # for the comparison below
```



Our quarter-width VGG reaches roughly 93% validation accuracy, two percentage points ahead of AlexNet with roughly half the arithmetic. To be fair, much of that lead reflects the higher learning rate: at AlexNet's peak of 0.03, the same VGG finishes just under 92%, as we verify shortly. Notice, however, what narrowing the convolutional trunk failed to fix:

although the convolutions shrank to just 0.6 million parameters, the dense head still maps $128 \times 7 \times 7 = 6272$ activations to 4096 hidden units, accounting for 42.5 million of the model's 43.1 million parameters—an astonishing 99%. Blocks regularized the body of the network, leaving the head as bloated as ever.

7.2.2 NiN: 1×1 Convolutions and Global Average Pooling

A dense classification head consumes tens of millions of parameters because it assigns a separate weight linking every spatial coordinate of every channel to every hidden unit. All of that capacity serves a single task: summarizing the final feature maps into class logits. *Network in Network* (NiN) asks whether we can dispense with that dense head entirely, while simultaneously enriching the nonlinearity within each convolutional block. Recall that a standard convolution followed by a ReLU applies only a thresholded linear filter to each local patch. To compute a richer nonlinear function of the same spatial window without enlarging the kernel, we can stack per-pixel channel-mixing layers—precisely the 1×1 convolutions introduced in Section 6.4.3.

7.2.2.1 NiN Blocks

Because a 1×1 convolution applies the same fully connected layer independently at every pixel, following a spatial convolution with two 1×1 convolutions (each paired with ReLU) places a miniature one-hidden-layer MLP at every spatial location—hence the name *network in network*. Figure 7.5 contrasts VGG's dense head with the end of NiN.

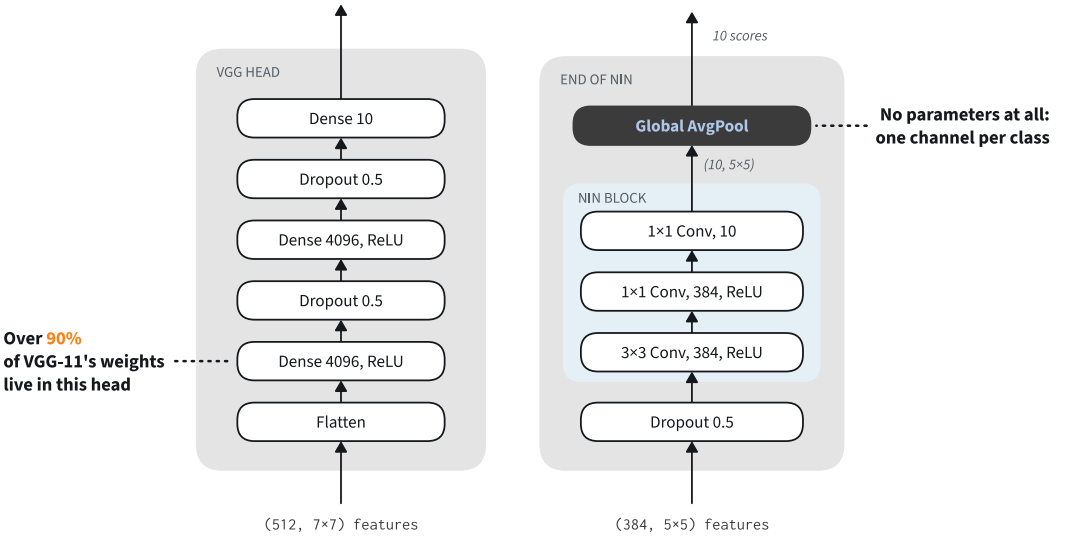


Figure 7.5: The head of VGG (left) and the end of NiN (right), both for ten classes; the input is at the bottom. VGG flattens its last feature map into fully connected layers. After dropout, NiN's last block, a 3×3 convolution and two 1×1 convolutions with no ReLU after the last one, emits one channel per class, and global average pooling turns each channel into one score.

```
def nin_block(out_channels, kernel_size, strides, padding, num_classes=None):
    layers = [
        nn.LazyConv2d(out_channels, kernel_size, strides, padding), nn.ReLU(),
        nn.LazyConv2d(out_channels, kernel_size=1), nn.ReLU()]
    if num_classes is None:
        layers += [nn.LazyConv2d(out_channels, kernel_size=1), nn.ReLU()]
    else:  # Last block: one map of scores per class, no ReLU
        layers.append(nn.LazyConv2d(num_classes, kernel_size=1))
    return nn.Sequential(*layers)
```

7.2.2.2 Global Average Pooling

For its backbone, NiN mirrors AlexNet's early kernel sizes and channel counts: blocks starting with 11×11, 5×5, and 3×3 convolutions of widths 96, 256, and 384, each followed by 3×3 max-pooling with stride 2. At the top of the network, however, NiN breaks decisively with AlexNet and VGG. After dropout, a fourth 384-channel NiN block terminates in a 1×1 convolution whose output channel count equals the number of classes (10 for Fashion-MNIST), and *global average pooling* averages each of those ten feature maps across all spatial coordinates to produce the ten class logits directly. With zero parameters in the pooling head, the final block itself must learn to produce high average spatial responses in the channel corresponding to the true class.

Notice an important implementation detail: when `num_classes` is passed to `nin_block`, that final 1×1 convolution omits the ReLU activation because its outputs are raw class logits. Placing a ReLU on the logits would risk permanently killing a class: if every spatial position of a class channel ever became negative across the dataset, its logit and gradient would vanish identically, preventing SGD from ever reviving it.

Global average pooling brings a second conceptual advantage alongside its parameter savings. Because convolutions are translation-equivariant and averaging over all spatial coordinates is invariant to permutations, the resulting class scores do not depend on *where* an object appears in the image (up to border and striding effects). By contrast, a flattened dense head assigns distinct weights to each spatial cell and must relearn every object at every position.

```
class NiN(d2l.Classifier):
    def __init__(self, lr=0.1, num_steps=4690, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(
            nin_block(96, kernel_size=11, strides=4, padding=0),
            nn.MaxPool2d(3, stride=2),
            nin_block(256, kernel_size=5, strides=1, padding=2),
            nn.MaxPool2d(3, stride=2),
            nin_block(384, kernel_size=3, strides=1, padding=1),
            nn.MaxPool2d(3, stride=2),
            nn.Dropout(0.5),
            nin_block(384, kernel_size=3, strides=1, padding=1,
                      num_classes=num_classes),
            nn.AdaptiveAvgPool2d((1, 1)),
            nn.Flatten())
```

```
    weight_init = VGG.weight_init  # He-normal weights, zero biases
```

Let us trace the tensor shapes on a single 224×224 input.

```
NiN().layer_summary((1, 1, 224, 224))
```

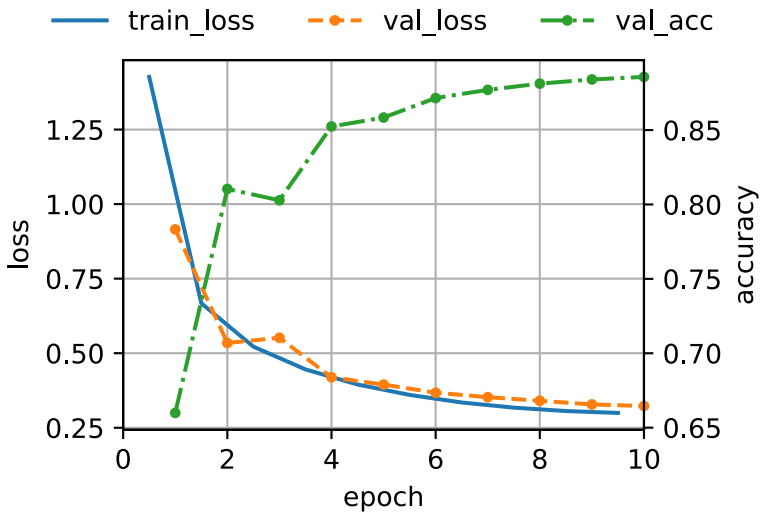
```
Sequential            (1, 96, 54, 54)
MaxPool2d             (1, 96, 26, 26)
Sequential            (1, 256, 26, 26)
MaxPool2d             (1, 256, 12, 12)
Sequential            (1, 384, 12, 12)
MaxPool2d             (1, 384, 5, 5)
Dropout               (1, 384, 5, 5)
Sequential            (1, 10, 5, 5)
AdaptiveAvgPool2d     (1, 10, 1, 1)
Flatten               (1, 10)
```

Just as in AlexNet, the spatial dimensions step down from 224 to 54, 26, 12, and 5, after which the final NiN block emits ten 5×5 score maps that global average pooling collapses to ten logits. Eliminating the dense head slashes the parameter count to just 3.4 million (only 7% of AlexNet's footprint), while total arithmetic stays comparable at 798 million multiply-adds per image (versus 938 million for AlexNet), because the added 1×1 convolutions shift computation into the convolutional body.

7.2.2.3 Training NiN

We train NiN with the same optimizer schedule as VGG, using a peak learning rate of 0.03 because runs at 0.1 occasionally diverge.

```
NiN.configure_optimizers = VGG.configure_optimizers  # AlexNet's schedule
data = d2l.FashionMNIST(batch_size=128, resize=(224, 224))
model = NiN(lr=0.03, num_steps=10 * len(data.train_dataloader()))
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
trainer.fit(model, data)
val_acc['NiN'] = trainer.history['val_acc']
```



After 10 epochs, NiN reaches roughly 89% validation accuracy—about two points below AlexNet and four below our quarter-width VGG. Its training loss finishes near 0.30 (compared with ~0.22 for AlexNet) with its validation loss tracking closely, indicating that in 10 epochs NiN underfits slightly without overfitting. As an architecture on its own, NiN is modest for its arithmetic cost; its lasting impact comes from the two reusable primitives it popularized: 1×1 channel-mixing convolutions and global average pooling heads.

7.2.2.4 A Pooled Head for VGG

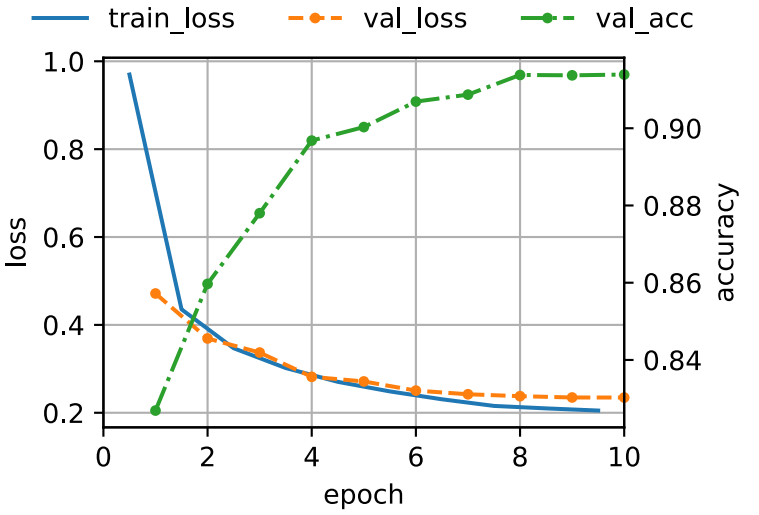
Because NiN alters both the convolutional body and the classifier head simultaneously, its score alone does not tell us how much the dense head was actually contributing. To isolate the head cleanly, we keep the exact same quarter-width VGG body and override only `VGG.head`. In `PooledVGG`, global average pooling summarizes the 128 output channels of the final block, and a single linear layer maps those 128 features directly to the ten classes—cutting away 42.5 million of the network's 43.1 million parameters (and the same number of multiply-adds).

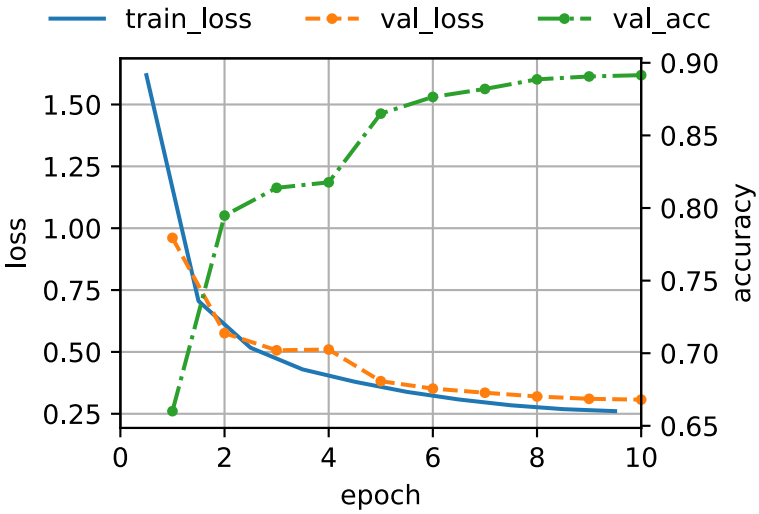
```
class PooledVGG(VGG):
    def head(self):  # Global average pooling and one fully connected layer
        return [nn.AdaptiveAvgPool2d((1, 1)), nn.Flatten(),
                nn.LazyLinear(self.num_classes)]
```

Because `PooledVGG`, like `NiN`, can diverge or collapse at 0.1, we train both `VGG` and `PooledVGG` at the matched peak learning rate `lr=0.03` and store their final validation accuracies in `heads`.

```
heads = {}  # Both heads at peak 0.03, the rate the pooled head tolerates
for net in (VGG, PooledVGG):
    model = net(arch=((1, 16), (1, 32), (2, 64), (2, 128), (2, 128)),
                lr=0.03, num_steps=10 * len(data.train_dataloader()))
    trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
    trainer.fit(model, data)
```

```
    heads[net.__name__] = round(trainer.history['val_acc'][-1], 3)
val_acc['PooledVGG'] = trainer.history['val_acc']
heads
```






```
{'VGG': 0.914, 'PooledVGG': 0.891}
```

`PooledVGG` finishes at 89–90% validation accuracy, roughly two percentage points behind the dense-headed `VGG` (91–92% at the same 0.03 peak), with a higher training loss (~0.25 vs. ~0.20). Thus, for this narrow convolutional body trained for 10 epochs on Fashion-MNIST, the extra capacity of the dense head is worth roughly two accuracy points—at the expense of 99% of the model's parameters.

7.2.3 GoogLeNet: Multi-Branch Blocks

Every architecture so far has forced the designer to pick a single kernel size at each layer: 1×1 mixes channels locally, 3×3 captures fine textures, and 5×5 spans broader neighborhoods, yet objects in real photographs appear at every scale. GoogLeNet sidesteps that choice altogether: its *Inception block* processes the same input through multiple kernel sizes in parallel and concatenates their outputs along the channel axis. Running multiple filter sizes at once would naively explode both parameters and computation, so GoogLeNet borrows NiN's 1×1 convolution as a dimensionality-reduction bottleneck ahead of every expensive spatial filter.

7.2.3.1 The Inception Block

As shown in Figure 7.6, each Inception block splits its input across four parallel branches: (1) a 1×1 convolution; (2) a 1×1 channel reduction followed by a 3×3 convolution; (3) a 1×1 channel reduction followed by a 5×5 convolution; and (4) a 3×3 stride-1 max-pooling layer followed by a 1×1 convolution. Every branch pads its input to preserve spatial height and width so the four outputs can be concatenated cleanly along the channel dimension. The per-branch channel counts act as tuning knobs allocating capacity across spatial scales. (The block's name playfully nods to the film *Inception* and the deep-learning meme "we need to go deeper.")

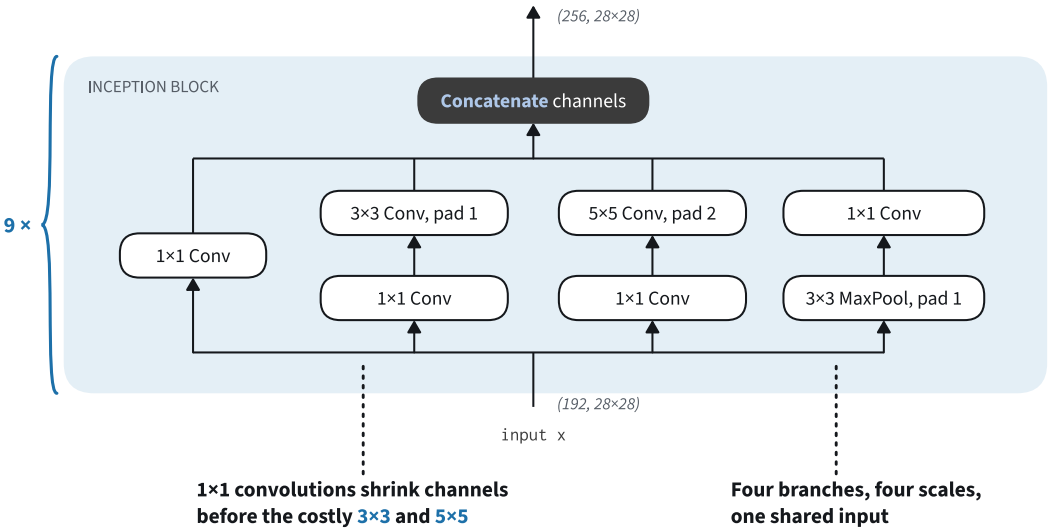


Figure 7.6: The Inception block; the input is at the bottom. Four parallel branches read the same input with different kernel sizes; 1×1 convolutions reduce the number of channels before the 3×3 and 5×5 convolutions; the branch outputs are concatenated along the channel dimension. The shapes are those of the first Inception block of GoogLeNet on a 224×224 input.

```
class Inception(nn.Module):
    # c1--c4 are the numbers of output channels of the four branches
    def __init__(self, c1, c2, c3, c4):
```

```
        super().__init__()
        self.b1_1 = nn.LazyConv2d(c1, kernel_size=1)
        self.b2_1 = nn.LazyConv2d(c2[0], kernel_size=1)
        self.b2_2 = nn.LazyConv2d(c2[1], kernel_size=3, padding=1)
        self.b3_1 = nn.LazyConv2d(c3[0], kernel_size=1)
        self.b3_2 = nn.LazyConv2d(c3[1], kernel_size=5, padding=2)
        self.b4_1 = nn.MaxPool2d(kernel_size=3, stride=1, padding=1)
        self.b4_2 = nn.LazyConv2d(c4, kernel_size=1)

    def forward(self, x):
        b1 = F.relu(self.b1_1(x))
        b2 = F.relu(self.b2_2(F.relu(self.b2_1(x))))
        b3 = F.relu(self.b3_2(F.relu(self.b3_1(x))))
        b4 = F.relu(self.b4_2(self.b4_1(x)))
        return torch.cat((b1, b2, b3, b4), dim=1)
```

GoogLeNet's first Inception block vividly illustrates the savings of 1×1 channel reduction. On a 224×224 input, it receives 192 channels at 28×28 resolution (Figure 7.6). With branch widths $c_1 = 64$, $c_2 = (96, 128)$, $c_3 = (16, 32)$, and $c_4 = 32$, it outputs $64 + 128 + 32 + 32 = 256$ channels at the same 28×28 resolution. Applying a 5×5 convolution directly from 192 channels to 32 would require $25 \cdot 192 \cdot 32 = 153{,}600$ weights. Projecting 192 channels down to 16 first costs only $192 \cdot 16 = 3{,}072$ weights for the 1×1 reduction plus $25 \cdot 16 \cdot 32 = 12{,}800$ for the 5×5 filter—15,872 weights in all, a nearly tenfold reduction in both parameters and multiply-adds. Placing a narrow 1×1 projection ahead of an expensive spatial convolution is called a *bottleneck*, and it became one of the most widely adopted primitives in deep vision, reappearing in Section 7.4.5.1 and Section 7.5.

7.2.3.2 Stem, Body, and Head

At a macro level, GoogLeNet established the three-part *stem–body–head* blueprint shared by almost all modern vision backbones. The *stem* ingests raw pixels through a short stack of standard convolutions and pooling layers (here three convolutions and two max-poolings), rapidly reducing spatial resolution eightfold while expanding to 192 channels. The *body* chains nine Inception blocks arranged in three stages of two, five, and two blocks, separated by stride-2 max-pooling. Finally, the *head* adopts NiN's compact design: global average pooling followed by a single linear classifier layer.

The per-branch widths of the nine Inception blocks were hand-tuned in the original paper rather than derived from a simple formula. Just as in VGG, we encode the entire schedule as structured data—a tuple of stages, each holding one configuration tuple per Inception block:

```
googlenet_arch = (
    ((64, (96, 128), (16, 32), 32), (128, (128, 192), (32, 96), 64)),
    ((192, (96, 208), (16, 48), 64), (160, (112, 224), (24, 64), 64),
     (128, (128, 256), (24, 64), 64), (112, (144, 288), (32, 64), 64),
     (256, (160, 320), (32, 128), 128)),
    ((256, (160, 320), (32, 128), 128), (384, (192, 384), (48, 128), 128)))
```

Passing `arch` into `GoogLeNet` assembles the stem, the three Inception stages separated by max-pooling, and the global-pooling head.

```
class GoogLeNet(d2l.Classifier):
    def __init__(self, arch=googlenet_arch, lr=0.1, num_steps=4690,
                 num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        pool = lambda: nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        stem = nn.Sequential(
            nn.LazyConv2d(64, kernel_size=7, stride=2, padding=3), nn.ReLU(),
            pool(),
            nn.LazyConv2d(64, kernel_size=1), nn.ReLU(),
            nn.LazyConv2d(192, kernel_size=3, padding=1), nn.ReLU(), pool())
        body = [nn.Sequential(*[Inception(*c) for c in group])
                for group in arch]
        head = nn.Sequential(nn.AdaptiveAvgPool2d((1, 1)), nn.Flatten(),
                             nn.LazyLinear(num_classes))
        self.net = nn.Sequential(stem, body[0], pool(), body[1], pool(),
                                 body[2], head)

    weight_init = VGG.weight_init  # He-normal weights, zero biases
```

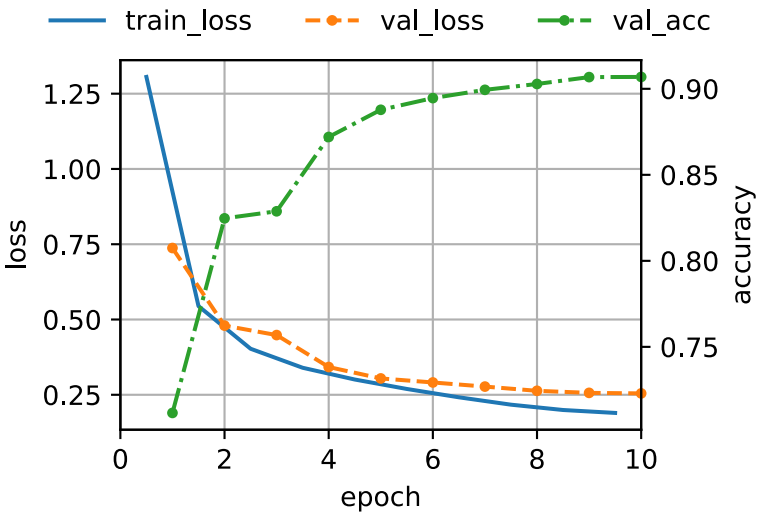
With five stride-2 downsampling operations in total, three inside the stem (the initial 7×7 convolution plus two max-poolings) and two between the Inception stages, GoogLeNet shrinks spatial resolution by a factor of 32. To keep Fashion-MNIST training brisk, we resize inputs to 96×96, so the three Inception stages operate at 12×12, 6×6, and 3×3 while channel depth climbs from 192 after the stem to 480, 832, and 1024.

```
GoogLeNet().layer_summary((1, 1, 96, 96))
```

```
Sequential              (1, 192, 12, 12)
Sequential              (1, 480, 12, 12)
MaxPool2d               (1, 480, 6, 6)
Sequential              (1, 832, 6, 6)
MaxPool2d               (1, 832, 3, 3)
Sequential              (1, 1024, 3, 3)
Sequential              (1, 10)
```

GoogLeNet is by far the deepest architecture in this section, spanning 22 parameterized layers along its longest path (three in the stem, two per Inception block across nine blocks, and the linear head). As with NiN, we train it at a peak learning rate of 0.03 because occasional runs at 0.1 stall at chance. Keep one experimental detail in mind for the comparison below: because we train on 96×96 rather than 224×224 inputs, GoogLeNet's 6.0 million parameters require only 276 million multiply-adds per image.

```
GoogLeNet.configure_optimizers = VGG.configure_optimizers
data = d2l.FashionMNIST(batch_size=128, resize=(96, 96))
model = GoogLeNet(lr=0.03, num_steps=10 * len(data.train_dataloader()))
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
trainer.fit(model, data)
val_acc['GoogLeNet'] = trainer.history['val_acc']
```



After 10 epochs, GoogLeNet finishes near 91% validation accuracy, matching AlexNet at the same peak learning rate, with a slightly lower training loss (just below 0.20) and a matching validation loss (~0.25).

7.2.4 Comparing the Networks

Because the cosine schedule decays the learning rate smoothly to zero, final-epoch validation accuracy is stable across all four models and lets us compare them directly:

```
{name: round(acc[-1], 3) for name, acc in val_acc.items()}  # After 10 epochs
```

```
{'VGG': 0.934, 'NiN': 0.886, 'PooledVGG': 0.891, 'GoogLeNet': 0.907}
```

Table 7.2 brings together LeNet, AlexNet, and all four architectures from this section, listing input resolution, parameter count, multiply-adds per image, peak learning rate, and Fashion-MNIST validation accuracy.

Table 7.2: Parameters, multiply-adds, and Fashion-MNIST validation accuracy of the networks of Section 6.6, Section 7.1, and this section. Multiply-adds are per image at the training input size. Accuracies are after 10 epochs at the peak learning rate given, as ranges over repeated runs; a single value means that the runs agree to within half a point. LeNet's are after 30 epochs at a constant learning rate.

network	input	parameters	multiply-adds	peak rate	validation accuracy
LeNet	28×28	0.06M	0.4M	2.0	88–90% (30 epochs)
AlexNet	224×224	46.8M	938M	0.03	91%
VGG, quarter width	224×224	43.1M	512M	0.1	93%
VGG, quarter width	224×224	43.1M	512M	0.03	91–92%

network	input	parameters	multiply-adds	peak rate	validation accuracy
VGG, quarter width, pooled head	224×224	0.6M	470M	0.03	89–90%
NiN	224×224	3.4M	798M	0.03	89%
GoogLeNet	96×96	6.0M	276M	0.03	91%

Three lessons stand out from Table 7.2:

1. **Raw arithmetic does not dictate accuracy:** Quarter-width VGG achieves the highest accuracy at half of AlexNet's multiply-adds, whereas NiN spends nearly as much compute as AlexNet yet finishes two points lower.
2. **Bottlenecks and stem downsampling slash parameter and pixel budgets:** GoogLeNet matches AlexNet's 91% accuracy with $8\times$ fewer parameters (6.0M vs. 46.8M) and 96×96 inputs (276M multiply-adds, though at 224×224 GoogLeNet performs ~1.5B multiply-adds).
3. **Global average pooling trades a small accuracy margin on narrow trunks for massive parameter savings:** Swapping `VGG`'s dense head for global average pooling (`PooledVGG`) removes 99% of its parameters (43.1M $\rightarrow$ 0.6M) at a cost of ~2 percentage points on Fashion-MNIST, while on ImageNet GoogLeNet's deeper body with a pooled head decisively beat 144M-parameter VGG models.

Notice also a shared ceiling across all four networks: none normalizes intermediate activations, none exceeds ~22 layers without auxiliary heads or staged pretraining, and NiN, `PooledVGG`, and GoogLeNet all become unstable when the peak learning rate rises from 0.03 to 0.1. In Section 7.3, we fix this scale fragility directly.

7.2.5 Summary

After AlexNet, deep vision shifted from hand-designing individual layers to composing modular blocks. VGG introduced repeatable stacks of padded 3×3 convolutions and pooling, showing that small stacked kernels capture wide receptive fields with fewer weights and more nonlinearities while turning network topology into data. NiN introduced 1×1 per-pixel channel mixing and replaced massive dense classifier heads (which hold over 90% of the weights in AlexNet and VGG) with parameter-free global average pooling. GoogLeNet combined multi-scale parallel branches with 1×1 bottlenecks inside a clean stem–body–head layout. On Fashion-MNIST under a shared cosine warmup schedule, quarter-width VGG leads in accuracy, GoogLeNet matches AlexNet with an eighth of its parameters, and swapping VGG's dense head for global average pooling saves 99% of its parameters at a cost of roughly two percentage points.

7.2.6 Exercises

1. **VGG-16 from its configuration.** VGG-16 stacks 13 convolutional layers in five blocks with `arch=((2, 64), (2, 128), (3, 256), (3, 512), (3, 512))` and three fully connected layers of 4096, 4096, and 1000 units on 224×224 RGB images.
    1. Count its parameters and multiply-adds by hand, separately for the convolutional body and the fully connected head, and verify your total against the 138 million parameters reported by Simonyan and Zisserman (2015) .

   2. Which single layer holds the most parameters? Which layers perform the most multiply-adds, and why do several convolutional layers tie?
2. code **Native resolution.** Fashion-MNIST images are 28×28, so upsampling to 224×224 spends compute without adding information.
   1. Train quarter-width `VGG` (`arch=((1, 16), (1, 32), (2, 64), (2, 128), (2, 128))`, `lr=0.1`) on images resized to 56×56 and 84×84. State the spatial size of the feature map entering the dense head at each resolution, and compare validation accuracy and epoch time against the 224×224 run.
   2. `GoogLeNet` accepts a 28×28 input as written because every stride-2 step pads to halve resolution (rounding up). Compute the spatial resolution at which each of the three Inception stages operates on a 28×28 input. What do the 3×3 and 5×5 branches compute at the final stage?
3. **Where the savings come from.** ● Using the layer counts of this section and Table 7.1:
   1. For VGG-11, `NiN`, and `GoogLeNet` on a single-channel 224×224 image with 10 classes, split the parameters and multiply-adds between the convolutional layers and the classification head.
   2. Relative to VGG-11, does `NiN` save more parameters in its body or in its head? What about `GoogLeNet`? Answer the same questions for multiply-adds and name the design choice responsible for each saving.
4. code **Single-branch Inception ablation.** ● Build a variant of `GoogLeNet` whose Inception blocks keep only the 3×3 branch with its 1×1 reduction: let the 3×3 convolution emit as many channels as the full four-branch block, and choose the 1×1 reduction width so each block's parameter count stays within 10% of the original. Train both networks on 96×96 Fashion-MNIST and compare validation accuracy. What does the result say about mixing kernel sizes within a block?

7.3 Normalization Layers

When we trained LeNet in Section 6.6, the learning rate that worked smoothly for an MLP broke down: at 0.1, validation accuracy lingered at chance for several epochs and reached only about 73% after ten, forcing us to boost the learning rate twentyfold to 2.0 and train for 30 epochs to reach ~89%. We encountered the same fragility in Section 7.2, where a peak learning rate of 0.1 trained VGG cleanly yet caused NiN and GoogLeNet to stall or diverge. A five-layer network ought not to be this sensitive to its step size. Why does the usable learning rate depend so delicately on architecture, and can we insert a layer inside the network to eliminate that fragility?

The root cause is how signal and gradient scales drift as they propagate through depth. Each of LeNet's four sigmoid layers shrinks backpropagated gradients by at least a factor of $1/4$, leaving early layers with tiny updates compared with later ones so that no single learning rate fits the whole network. Adding depth compounds the issue because every layer rescales the activations entering the next. Although ReLU activations and careful initialization (Section 4.4.2.2) balance activation variance at step zero, nothing keeps those scales aligned once gradient updates begin moving the weights.

Batch normalization (Ioffe and Szegedy 2015) solves this dynamically throughout training. Placed right before a layer's nonlinearity, it standardizes each feature using the mean and variance of the current minibatch, then applies a learned scale and shift. That simple transformation stabilizes both forward activations and backward gradients while rendering each normalized layer invariant to weight rescaling. As we will observe below, adding batch nor-

malization lets LeNet train cleanly not just at the MLP's learning rate, but across a 300-fold range of step sizes. Alongside the residual connections of the next section, batch normalization unlocked networks over a hundred layers deep—even though researchers still debate the exact mechanisms behind its full empirical power. Because batch normalization couples examples within a minibatch, however, it degrades when minibatches are very small; in the second half of this section we introduce *layer normalization* and *group normalization*, which normalize each example on its own.

7.3.1 Batch Normalization

7.3.1.1 Standardizing over a Minibatch

Consider a single pre-activation feature computed by a fully connected layer across a minibatch $\mathcal{B}$ of m examples, producing scalar values $x_1, \dots, x_m$. Batch normalization computes their sample mean and sample variance across the minibatch,

$$\hat{\mu}_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} x_i,$$
$$\hat{\sigma}^2_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m}(x_i - \hat{\mu}_{\mathcal{B}})^2,$$

and transforms each value via

$$\mathrm{BN}(x_i) = \gamma\,\frac{x_i - \hat{\mu}_{\mathcal{B}}}{\sqrt{\hat{\sigma}^2_{\mathcal{B}} + \epsilon}} + \beta. \tag{7.3}$$

The fraction standardizes the feature across the minibatch to zero mean and variance $\hat{\sigma}^2_{\mathcal{B}}/(\hat{\sigma}^2_{\mathcal{B}} + \epsilon)$, just shy of 1. The tiny constant $\epsilon > 0$ (typically 10^{-5}) guards against division by zero whenever all entries in the minibatch coincide. In a layer with multiple features, each feature is normalized independently using its own minibatch mean and variance.

As a concrete check, the four values $(2, 4, 6, 8)$ have mean 5 and variance $(9{+}1{+}1{+}9)/4 = 5$, standardizing (up to ϵ) to $(x_i - 5)/\sqrt{5} \approx (-1.34, -0.45, 0.45, 1.34)$. Notice that $(20, 40, 60, 80)$ and $(102, 104, 106, 108)$ map to the exact same four numbers: standardization strips away both the arbitrary scale and the offset of the incoming feature.

Zero mean and unit variance, however, are not necessarily ideal for every downstream activation function—a sigmoid, for example, behaves nearly linearly over unit-variance inputs. To restore full representational freedom, Equation 7.3 introduces a learnable *scale* γ and *shift* β for each feature, initialized to $\gamma = 1$ and $\beta = 0$ and trained alongside the network weights. Because setting γ and β to the dataset's true standard deviation and mean recovers the original unnormalized feature (up to minibatch sampling fluctuations), batch normalization does not reduce the expressive power of the network.

7.3.1.2 Scale Invariance

Stripping away feature scale produces two powerful effects on optimization. The first is immediate and directly explains how batch normalization rescues LeNet: whenever a feature varies very little across a minibatch (for instance, after passing through saturating sigmoids

clustered near $1/2$), dividing by its tiny standard deviation $\hat{\sigma}_{\mathcal{B}}$ restores unit variance on the forward pass, and by the chain rule amplifies its backpropagated gradient by the same factor $1/\hat{\sigma}_{\mathcal{B}}$. Vanishing signals and vanishing gradients are both automatically rescaled.

The second effect is subtler and acts on the layer weights themselves. Write $x_i = \mathbf{w}^\top \mathbf{h}_i + b$ for the pre-activation of example i. Scaling $\mathbf{w}$ and b by any positive constant $a > 0$ scales every x_i, their mean, and their standard deviation by a, leaving the normalized output unchanged (ignoring ϵ):

$$\mathrm{BN}(a\mathbf{w}^\top \mathbf{h}_i + ab) = \mathrm{BN}(\mathbf{w}^\top \mathbf{h}_i + b).$$

Consequently, the loss depends only on the *direction* of $\mathbf{w}$, not its Euclidean norm. Differentiating both sides with respect to $\mathbf{w}$ reveals that the gradient at $a\mathbf{w}$ equals $1/a$ times the gradient at $\mathbf{w}$ (Ioffe and Szegedy 2015): small weights receive larger gradients while large weights receive smaller ones.

Consider what happens if we choose an overly aggressive learning rate η. Because the loss is constant along the radial ray of $\mathbf{w}$, the gradient $\mathbf{g}$ is orthogonal to $\mathbf{w}$, so every SGD step strictly increases the weight norm: $\|\mathbf{w} - \eta\mathbf{g}\|^2 = \|\mathbf{w}\|^2 + \eta^2\|\mathbf{g}\|^2$. An excessively large step inflates $\|\mathbf{w}\|$ rapidly, which in turn shrinks all subsequent gradients by the same factor—creating a built-in self-stabilizing feedback loop for large learning rates (Arora et al. 2019).

Keep the caveats of this analysis in mind. Strictly speaking, it assumes plain SGD without weight decay (under weight decay, weight norms reach an equilibrium where outward gradient growth balances inward decay); it applies only to layers immediately followed by normalization (not the final output layer of a classifier); and it cannot stop divergence during the very first few steps before weight norms grow. Even with those caveats, scale invariance provides our clearest explanation for why normalized networks tolerate vastly wider learning-rate ranges.

7.3.1.3 Fully Connected and Convolutional Layers

For a fully connected layer with weights $\mathbf{W}$, bias $\mathbf{b}$, and nonlinearity ϕ, batch normalization is placed between the affine transformation and the activation function, as proposed in Ioffe and Szegedy (2015):

$$\mathbf{h} = \phi(\mathrm{BN}(\mathbf{W}\mathbf{x} + \mathbf{b})). \tag{7.4}$$

Here each output unit is treated as a separate feature with its own minibatch mean, variance, γ_j, and β_j.

For a convolutional layer, batch normalization respects translation equivariance by treating each output *channel* as one feature, computing a single mean and variance across both the m minibatch examples *and* all $p \times q$ spatial locations ($m \cdot p \cdot q$ values per channel). In LeNet's first convolutional layer (6 channels of 28×28 with batch size 128), each channel's statistics pool $128 \cdot 28 \cdot 28 = 100{,}352$ entries—far more stable than the 128 entries per unit in a dense layer. Each channel likewise shares a single scalar γ and β across all spatial coordinates so that the normalized layer remains shift-equivariant.

7.3.1.4 Training and Prediction

At inference time, a deployed model often scores one image at a time, and its prediction for a given input must be deterministic rather than dependent on whatever other images happen to share its minibatch. Batch normalization therefore operates in two distinct modes. In *training mode*, it normalizes using the current minibatch's statistics as in Equation 7.3, while updating exponential moving averages $\hat{\mu}$ and $\hat{\sigma}^2$ (initialized to 0 and 1) after every step:

$$\begin{aligned}\hat{\mu} &\leftarrow \rho\,\hat{\mu} + (1-\rho)\,\hat{\mu}_{\mathcal{B}},\\ \hat{\sigma}^2 &\leftarrow \rho\,\hat{\sigma}^2 + (1-\rho)\,\hat{\sigma}^2_{\mathcal{B}}.\end{aligned} \tag{7.5}$$

The *decay* ρ governs the effective window of these running statistics: a minibatch from k steps ago carries weight $(1-\rho)\rho^k$. At $\rho = 0.9$, weights halve roughly every seven steps, averaging over roughly the most recent $1/(1-\rho) = 10$ minibatches (out of 469 minibatches of size 128 per Fashion-MNIST epoch). In *prediction mode*, the layer freezes these running estimates and normalizes every input with them, reducing batch normalization to a fixed, deterministic affine transformation. Like dropout (Section 4.6), batch normalization thus behaves differently during training and evaluation.

Watch out for a notorious API inconsistency across libraries: some frameworks name the parameter ρ `momentum`, whereas others name $1-\rho$ `momentum`—and neither has anything to do with optimization momentum. Throughout this chapter we standardize on $\rho = 0.9$ and $\epsilon = 10^{-5}$, passing them explicitly wherever a framework's defaults differ.

7.3.2 Implementation from Scratch

Below, `batch_norm` implements both Equation 7.3 and Equation 7.5 across training and prediction modes for both fully connected (2D) and convolutional (4D) inputs.

Images are stored as (examples, channels, height, width), so the channel axis is 1.

```
def batch_norm(X, gamma, beta, moving_mean, moving_var, eps, momentum,
               training):
    if not training:
        # Prediction mode: normalize with the running estimates
        X_hat = (X - moving_mean) / torch.sqrt(moving_var + eps)
    else:
        assert len(X.shape) in (2, 4)
        # Fully connected: average over examples. Convolutional: over
        # examples and positions, one mean and variance per channel
        dims = 0 if len(X.shape) == 2 else (0, 2, 3)
        mean = X.mean(dim=dims, keepdim=True)
        var = ((X - mean) ** 2).mean(dim=dims, keepdim=True)
        X_hat = (X - mean) / torch.sqrt(var + eps)
        # Update the running estimates; `momentum` is the decay rho
        moving_mean = momentum * moving_mean + (1.0 - momentum) * mean
        moving_var = momentum * moving_var + (1.0 - momentum) * var
    Y = gamma * X_hat + beta  # Scale and shift
    return Y, moving_mean.detach(), moving_var.detach()
```

Wrapping `batch_norm` in a `BatchNorm` layer requires tracking two distinct kinds of state: learnable parameters (`gamma` and `beta`) updated by the optimizer, and non-trainable buffers

(`moving_mean` and `moving_var`) that the optimizer ignores yet the module still saves to disk and transfers to GPUs alongside its weights.

```
class BatchNorm(nn.Module):
    # `num_features`: outputs of a fully connected layer or channels of a
    # convolutional layer. `num_dims`: 2 or 4, the number of input axes
    def __init__(self, num_features, num_dims):
        super().__init__()
        shape = (1, num_features) if num_dims == 2 else (1, num_features, 1, 1)
        # Scale and shift, learned, start at 1 and 0
        self.gamma = nn.Parameter(torch.ones(shape))
        self.beta = nn.Parameter(torch.zeros(shape))
        # Running estimates, not learned, start at 0 and 1
        self.register_buffer('moving_mean', torch.zeros(shape))
        self.register_buffer('moving_var', torch.ones(shape))

    def forward(self, X):
        Y, moving_mean, moving_var = batch_norm(
            X, self.gamma, self.beta, self.moving_mean, self.moving_var,
            eps=1e-5, momentum=0.9, training=self.training)
        self.moving_mean.copy_(moving_mean)
        self.moving_var.copy_(moving_var)
        return Y
```

`register_buffer` registers a tensor that `model.to(device)` moves and `state_dict()` saves, but that `parameters()` does not return. The attribute `self.training` is set by `model.train()` and `model.eval()`, which the Trainer calls before training and validation.

To verify our from-scratch implementation, we run both `BatchNorm` and the framework's built-in layer (each with $\rho = 0.9$ and $\epsilon = 10^{-5}$) on an identical minibatch of 128 six-channel 28×28 feature maps drawn with mean 2 and standard deviation 3. Below we check that every output channel has mean 0 and variance 1, that our output matches the framework's built-in layer, and that the running means update properly after one step.

```
X = 2 + 3 * torch.randn(128, 6, 28, 28)
bn, bn_ref = BatchNorm(6, num_dims=4), nn.BatchNorm2d(6)  # rho 0.9, eps 1e-5
with torch.no_grad():
    Y, Y_ref = bn(X), bn_ref(X)  # New layers are in training mode
{'largest channel mean': float(Y.mean(dim=(0, 2, 3)).abs().max()),
 'average channel variance': float(Y.var(dim=(0, 2, 3),
                                         unbiased=False).mean()),
 'largest difference': float((Y - Y_ref).abs().max()),
 'running means': bn.moving_mean.reshape(-1)}
```

```
{'largest channel mean': 1.8056253736631334e-07,
 'average channel variance': 0.9999989867210388,
 'largest difference': 7.152557373046875e-07,
 'running means': tensor([0.1994, 0.1993, 0.2000, 0.2007, 0.1991, 0.1981])}
```

Every channel now has zero mean and unit variance up to single-precision roundoff, and our implementation matches the built-in layer to within 10^{-6}. After a single minibatch, the running means have moved a fraction $1 - \rho = 0.1$ of the way from 0 toward the minibatch mean of 2.

`nn.BatchNorm2d(6)` uses PyTorch's default `momentum=0.1`, which is our $\rho = 0.9$.

7.3.3 LeNet with Batch Normalization

Let us now return to LeNet (Section 6.6) and put batch normalization to the test. Keeping the exact same Xavier initialization, `BNLeNet` inserts a built-in batch normalization layer after each of LeNet's four hidden layers (the 6- and 16-channel convolutions and the 120- and 84-unit dense layers) right before each sigmoid activation. Notice that we disable biases on those four layers: subtracting the minibatch mean completely cancels any additive bias immediately preceding batch normalization, while β takes over its role. Across all $6 + 16 + 120 + 84 = 226$ normalized features, trading 226 biases for 226 (γ, β) pairs adds just 226 trainable parameters to LeNet's 61,706.

```
class BNLeNet(d2l.Classifier):
    """LeNet with batch normalization before each hidden sigmoid."""
    def __init__(self, lr=0.1, num_steps=4690, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(  # No biases before batch normalization
            nn.LazyConv2d(6, kernel_size=5, padding=2, bias=False),
            nn.LazyBatchNorm2d(), nn.Sigmoid(),
            nn.AvgPool2d(kernel_size=2, stride=2),
            nn.LazyConv2d(16, kernel_size=5, bias=False),
            nn.LazyBatchNorm2d(), nn.Sigmoid(),
            nn.AvgPool2d(kernel_size=2, stride=2),
            nn.Flatten(),
            nn.LazyLinear(120, bias=False), nn.LazyBatchNorm1d(), nn.Sigmoid(),
            nn.LazyLinear(84, bias=False), nn.LazyBatchNorm1d(), nn.Sigmoid(),
            nn.LazyLinear(num_classes))

    weight_init = d2l.LeNet.weight_init  # Xavier weights, zero biases
```

`LazyBatchNorm2d` normalizes each channel of a four-axis input and `LazyBatchNorm1d` each feature of a two-axis one. Their defaults, `momentum=0.1` and `eps=1e-5`, are our $\rho = 0.9$ and ϵ. `BNLeNet` reuses LeNet's `weight_init`; only the output layer has a bias for it to set to zero.

Before training, notice a subtle interaction between prediction mode and the learning rate. Because the running averages lag behind by roughly ten minibatches, whenever the network weights are still changing rapidly those averages describe a slightly stale snapshot of the network. This is not merely theoretical: under a constant learning rate of 0.1, `BNLeNet`'s prediction-mode validation accuracy can suddenly dip by 10 percentage points or more on individual epochs, including the final one. Decaying the learning rate smoothly to zero fixes this cleanly: as the step size shrinks, the weights settle and the running averages lock onto the final network. We therefore reuse the warmup-and-cosine schedule from Section 7.1 (studied further in Chapter 9), warming up over the first tenth of `num_steps`, and wrap `d2l.LeNet` in a subclass `CosineLeNet` so both models share the exact same schedule.

```
@d2l.add_to_class(BNLeNet)
def configure_optimizers(self):
    optim = torch.optim.SGD(self.parameters(), lr=self.lr)
    schedule = torch.optim.lr_scheduler.LambdaLR(optim, lambda t: (
```

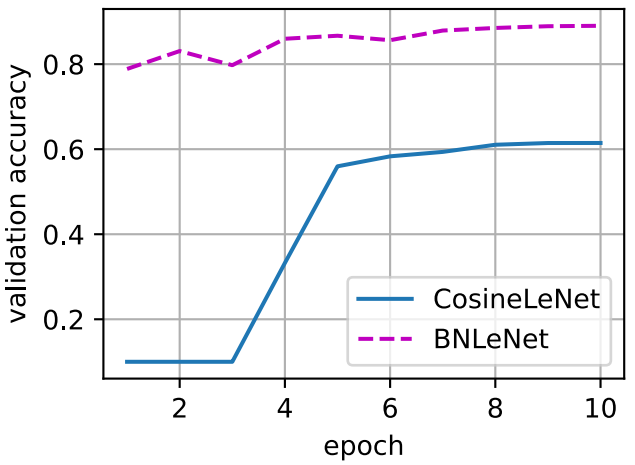
```
            d2l.cosine_warmup(t, self.num_steps, self.num_steps // 10)))
        return optim, schedule  # The Trainer steps it after every minibatch

class CosineLeNet(d2l.LeNet):
    def __init__(self, lr=0.1, num_steps=4690, num_classes=10):
        super().__init__(lr, num_classes)
        self.num_steps = num_steps

    configure_optimizers = BNLeNet.configure_optimizers
```

Let us train both `CosineLeNet` and `BNLeNet` under identical settings (peak learning rate 0.1, batch size 128, and 10 epochs) and track validation accuracy epoch by epoch.

```
data = d2l.FashionMNIST(batch_size=128)
num_steps = 10 * len(data.train_dataloader())
curves = {}
for model in (CosineLeNet(0.1, num_steps), BNLeNet(0.1, num_steps)):
    trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
    trainer.board.display = False  # We plot the accuracies ourselves
    trainer.fit(model, data)
    curves[type(model).__name__] = trainer.history['val_acc']
```

Plotting the epoch-by-epoch validation accuracy trajectories stored in `curves` contrasts the convergence speed of `CosineLeNet` and `BNLeNet` directly.

```
d2l.plot(list(range(1, 11)), list(curves.values()), 'epoch',
         'validation accuracy', legend=list(curves))
```



We also print the final 10-epoch validation accuracy `acc[-1]` from `curves` for both models to compare their endpoint performance.

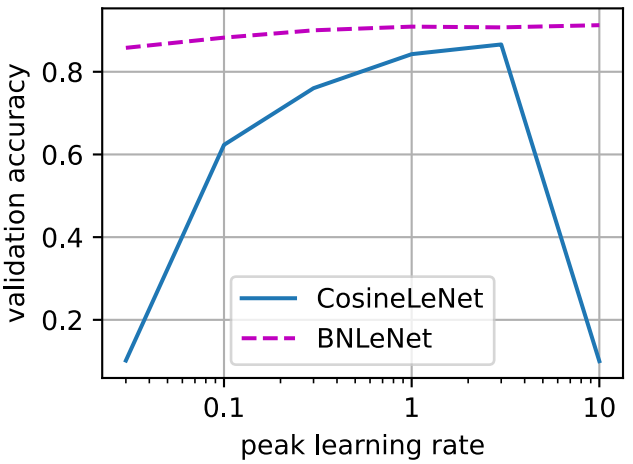
```
{name: round(acc[-1], 3) for name, acc in curves.items()}  # After 10 epochs
```

```
{'CosineLeNet': 0.615, 'BNLeNet': 0.89}
```

Without batch normalization, LeNet stalls at chance (10%) for the first two to five epochs and reaches only around 60% after ten (lower than the 73% in Section 6.6, plausibly because the cosine decay halves the average step size). With batch normalization, the exact same architecture clears 70% in its very first epoch and climbs to ~88% after ten—nearly matching the 88–90% that unnormalized LeNet required twenty times the learning rate and three times as many epochs to attain. Notice also how smoothly the curve flattens over the final epochs as the learning rate anneals to zero. Adding just 226 parameters did not expand the network's representational capacity; it simply brought every layer's activations and gradients onto a healthy scale.

Comparing at a single learning rate, however, leaves open an important question: did batch normalization merely shift the sweet spot to a different step size, or does it truly widen the range of workable learning rates? To find out, we sweep peak learning rates across a 300-fold span from 0.03 to 10 and plot final 10-epoch validation accuracy for both models.

```
lrs = (0.03, 0.1, 0.3, 1, 3, 10)
final = {'CosineLeNet': [], 'BNLeNet': []}
for lr in lrs:
    for cls in (CosineLeNet, BNLeNet):
        trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
        trainer.board.display = False
        trainer.fit(cls(lr, num_steps), data)
        final[cls.__name__].append(trainer.history['val_acc'][-1])
d2l.plot(list(lrs), list(final.values()), 'peak learning rate',
         'validation accuracy', legend=list(final), xscale='log')
```



The sweep gives a decisive answer. Plain LeNet succeeds only in a narrow band around 1 to 3 (reaching 84–87%): below that window it learns too slowly in 10 epochs, and at 10 it becomes erratic, sometimes stalling at chance. By contrast, `BNLeNet` clears ~85% at *every* learning rate from 0.03 to 10, and reaches ~91% at rates of 1 and above (three full points higher than at 0.1). Where the unnormalized network must be tuned within a factor of three, the normalized network works smoothly across a factor of 300—just as our scale-invariance analysis predicted (though the ceiling cannot extend forever, since the final output layer

remains unnormalized).

Why batch normalization helps deep networks so dramatically goes beyond internal covariate shift (Ioffe and Szegedy 2015): subsequent studies showed that it smooths the loss landscape and gradients (Santurkar et al. 2018), keeps activation and gradient scales balanced across depth, renders hidden weights scale-invariant so large learning rates self-stabilize, and injects mild minibatch noise that regularizes like dropout (Luo et al. 2018; Brock et al. 2021).

7.3.4 Normalization without Batch Statistics

Coupling an example's output to the other examples in its minibatch is an unusual property for a neural network layer, and it introduces two practical weaknesses. First, minibatch statistics deteriorate as batch size shrinks. In the extreme case of a fully connected layer with a minibatch of one example, every entry equals its own mean, forcing every standardized activation to 0 so the layer outputs a constant β regardless of the input. Memory-heavy tasks such as object detection and semantic segmentation often fit only one or two high-resolution images per GPU, precisely where batch statistics are noisiest. Second, batch normalization computes two different functions in training and prediction modes; whenever running estimates drift out of sync with test-time inputs (for instance, after fine-tuning across domains), accuracy can degrade silently in ways that are notoriously hard to debug.

Both limitations vanish if we compute normalization statistics *within each example independently*. Figure 7.7 compares the resulting family of normalization layers on a convolutional activation tensor with batch axis N, channel axis C, and spatial axes $H \cdot W$: every variant uses the exact same standardization formula and differs solely in which entries are grouped together to share a mean and variance.

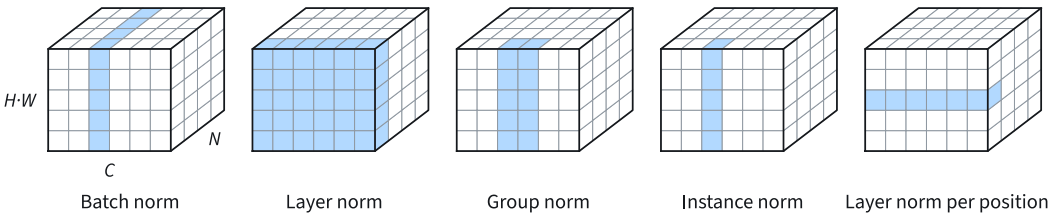


Figure 7.7: Normalization methods for activations with examples N, channels C, and spatial positions $H \cdot W$. The shaded numbers share one mean and one variance. Batch normalization averages each channel over the minibatch and all positions; layer normalization averages each example over all channels and positions; group normalization over a group of channels and all positions; instance normalization over a single channel. The layer normalization of transformers and ConvNeXt averages over the channels at one position.

7.3.4.1 Layer Normalization

Layer normalization (Ba et al. 2016) standardizes all n features $\mathbf{x} = (x_1, \ldots, x_n)$ of a single example independently:

$$\mathrm{LN}(\mathbf{x})_j = \gamma_j \frac{x_j - \hat{\mu}}{\sqrt{\hat{\sigma}^2 + \epsilon}} + \beta_j,$$
$$\text{with} \quad \hat{\mu} = \frac{1}{n}\sum_{j=1}^{n} x_j \quad \text{and} \quad \hat{\sigma}^2 = \frac{1}{n}\sum_{j=1}^{n}(x_j - \hat{\mu})^2.$$

This is Equation 7.3 transposed: instead of averaging one feature across examples, we average one example across its features while keeping the learnable scale γ_j and shift β_j per feature. Because no other example enters the computation, layer normalization behaves identically at every batch size and in both training and prediction modes, requiring no running statistics at all. Note that in practice the exact axes pooled by "layer normalization" depend on the architecture: in convolutional tensors it may pool across all channels and spatial positions of an example (second panel of Figure 7.7), whereas Transformers normalize each token's feature vector independently and ConvNeXt (Section 7.7) similarly normalizes the channel vector at each individual pixel without pooling across space (rightmost panel).

7.3.4.2 Group and Instance Normalization

Tailored specifically to convolutional layers, *group normalization* (Wu and He 2018) partitions the c channels into G groups of c/G channels and standardizes each example over each group's $(c/G) \cdot p \cdot q$ entries, followed by a per-channel learned scale and shift. It smoothly interpolates between two extremes: $G = 1$ pools across every channel and position (spatial layer normalization in Figure 7.7), whereas $G = c$ normalizes each channel of each example on its own, known as *instance normalization*. Intermediate choices (typically $G = 32$) provide enough elements per group for reliable statistics without forcing unrelated feature channels to share a single mean and variance. Below we verify on a batch of 32 tensors (16 channels, 4 groups) that every `(example, group)` slice is standardized to zero mean and unit variance and that normalizing the first example alone produces the exact same output as normalizing it inside the full batch.

```
X = torch.randn(32, 16, 8, 8)
gn = nn.GroupNorm(4, 16).requires_grad_(False)  # 4 groups of 4 channels
Y = gn(X)
G = Y.reshape(32, 4, -1)  # Collect each (example, group) pair's numbers
(G.mean(dim=2).abs().max(), G.var(dim=2, unbiased=False).mean(),
 (gn(X[:1]) - Y[:1]).abs().max())
```

```
(tensor(3.3528e-08), tensor(1.0000), tensor(0.))
```

As expected, every group achieves zero mean and unit variance up to roundoff, and the first example's output is strictly independent of the rest of the batch. Does this independence carry an accuracy tradeoff? At large batch sizes, only a tiny one: on ImageNet with ResNet-50 and 32 images per GPU, Wu and He (2018) report 23.6% top-1 error for batch normalization versus 24.1% for group normalization. Drop to 2 images per GPU, however, and batch normalization's error spikes to 34.7% while group normalization holds steady at 24.1%—making group normalization the standard choice for detection and segmentation models trained with small per-GPU batches.

7.3.4.3 Normalizer-Free Networks

Activation normalization is not the only way to keep signal scales well behaved in deep convnets. *Normalizer-free networks* (NFNets) (Brock et al. 2021) trained deep ResNet-style architectures without a single normalization layer to state-of-the-art ImageNet accuracy (among models trained solely on ImageNet). Rather than normalizing activations, NFNets standardize each convolutional kernel's weights directly, preserving throughout training the variance balance that initialization (Section 4.4.2.2) establishes only at step zero, and pair this with adaptive gradient clipping relative to each weight tensor's norm. Although we will not implement NFNets here, their success offers illuminating evidence: the fact that deep convnets train just as well when signal scale is controlled through weights instead of activations strongly supports the view that scale control is the core job of a normalization layer.

7.3.5 Summary

Batch normalization standardizes each feature of a dense layer, or each channel of a convolutional layer, using the mean and variance of the current minibatch, followed by a learned scale and shift. By keeping forward signals and backward gradients properly scaled and making normalized layers invariant to weight norms, it lets LeNet train smoothly across peak learning rates spanning 0.03 to 10 (a 300-fold band compared with a 3-fold band without it), even though the full theoretical explanation for its deep-network benefits remains open. At inference time, batch normalization switches to frozen exponential moving averages of the training statistics. Because coupling across examples breaks down at tiny batch sizes, layer normalization and group normalization compute their statistics strictly within each example, eliminating both batch-size sensitivity and the train–test mode split. In the next section, we combine batch normalization with residual connections to push network depth past a hundred layers.

7.3.6 Exercises

1. **Bias before batch norm.** `BNLeNet` builds its four normalized layers without a bias. Show from Equation 7.3 that adding a bias $\mathbf{b}$ before batch normalization changes neither the training-mode output nor the weight gradients, and that $\partial L/\partial \mathbf{b} = \mathbf{0}$. What happens to such a bias in prediction mode?

2. code **Learning-rate ceiling.** In the sweep of this section, `BNLeNet` trained cleanly at every peak rate up to 10.

    1. Extend the sweep for `BNLeNet` to peak rates of 30, 100, and 300. Where does training break down?
    2. The output linear layer is the only weight layer not followed by batch normalization. Test whether it sets the ceiling by fixing the output layer's learning rate at 0.1 while sweeping the normalized layers' peak rate across 30, 100, and 300.
    3. Replace the sigmoids in both `CosineLeNet` and `BNLeNet` with ReLUs (using He initialization) and repeat the learning-rate sweep. How does the workable band of the unnormalized network change, and why?

3. code **Running averages and recomputed statistics.** The decay ρ in Equation 7.5 determines how many recent minibatches the running statistics average over.

    1. After how many steps does a minibatch's weight in the running mean fall below 1% of the newest minibatch's weight for $\rho = 0.9$ and $\rho = 0.99$? Compare both

horizons against the 469 steps of one Fashion-MNIST epoch.
2. Train `BNLeNet` for 10 epochs at a constant learning rate of 0.1 (`d2l.Classifier.configure_optimizers`). After every epoch, evaluate the model twice: once in prediction mode using its running averages, and once after recomputing fresh statistics over 100 training minibatches with weights frozen. Plot both validation curves alongside the cosine-schedule curve and explain the gap.

4. code **Minibatch coupling.** Because batch normalization couples examples within a minibatch, evaluate a trained `BNLeNet` on the validation set in three ways (measuring prediction mode first so training-mode passes do not overwrite the running statistics): (i) prediction mode, (ii) training mode with validation batch size 128, and (iii) training mode with validation batch size 2 (or 1). Explain the three resulting accuracies.

5. code **Small-batch training: BatchNorm vs. GroupNorm/LayerNorm.** ● Build a variant of `BNLeNet` that replaces `BatchNorm` with group normalization (2 groups) after the two convolutional layers and layer normalization (Section 7.3.4.1) after the two hidden dense layers. Train both `BNLeNet` and this variant for 2 epochs at batch size 4 and at batch size 128. How does each model's validation accuracy change when the batch size drops from 128 to 4, and why?

6. code **Backward pass of batch norm.** ● Because $\hat{\mu}_{\mathcal{B}}$ and $\hat{\sigma}^2_{\mathcal{B}}$ in Equation 7.3 depend on every element of the minibatch, the backward pass of batch normalization couples all m examples.
    1. Derive by hand the gradient $\partial L/\partial x_i$ of a scalar loss L with respect to each input x_i of a 1D batch-norm layer, and show from your formula that $\sum_{i=1}^{m} \partial L/\partial x_i = 0$ identically.
    2. Implement your backward formula and verify that it matches automatic differentiation to machine precision.

7.4 Residual Networks

Up to this point on ImageNet, deeper networks consistently meant higher accuracy, from AlexNet's 8 layers to GoogLeNet's 22, and batch normalization (Section 7.3) seemed to clear the remaining barrier to scaling depth even further. Naturally, one would expect stacking still more layers to keep lowering training error, at worst until overfitting sets in. Yet when He et al. (2016a) tested this intuition by training *plain* VGG-style stacks of convolutions with batch normalization and ReLU at increasing depths, they discovered the exact opposite. On CIFAR-10, a 56-layer plain network finished with a substantially higher *training* error than a 20-layer network. Because it fits even the training set worse, this *degradation problem* is not overfitting; it is an optimization failure. The puzzle is acute: in principle, the 56-layer network could simply copy the 20-layer solution and act as the identity mapping across its remaining 36 layers, guaranteeing a training loss at least as low. Why does SGD fail to find such a solution, and how can we redesign the building block so it does?

Residual networks (ResNets) resolve the paradox with a deceptively simple reparameterization. Instead of asking each block to learn a full output transformation from scratch, a residual block adds a learned correction to its input—so driving the correction to zero immediately recovers the identity mapping. That single shortcut unlocked unprecedented depth: on ImageNet, where a 34-layer plain network trailed an 18-layer one, a 34-layer ResNet beat its 18-layer counterpart by nearly three percentage points, and 152-layer ResNets won the 2015 ImageNet competition.

In this section, we first frame depth through nested function classes and design the residual block. Next, we reproduce the degradation phenomenon directly on Fashion-MNIST by training 20- and 56-layer plain and residual networks side by side, watching degradation appear and vanish before our eyes. We then assemble and train ResNet-18, and conclude with two influential descendants: *ResNeXt*, which uses grouped convolutions to widen blocks at matched cost, and *DenseNet*, which concatenates feature maps rather than adding them.

7.4.1 Function Classes and Depth

Consider the *function class* $\mathcal{F}$ of an architecture—the set of input–output mappings achievable as its parameters vary. When is a larger architecture $\mathcal{F}'$ guaranteed to contain a solution with training loss at least as low as the best in $\mathcal{F}$? Whenever the classes are *nested*, $\mathcal{F} \subseteq \mathcal{F}'$ (Figure 7.8). If $\mathcal{F} \nsubseteq \mathcal{F}'$, a larger class can drift away from the target f^*.

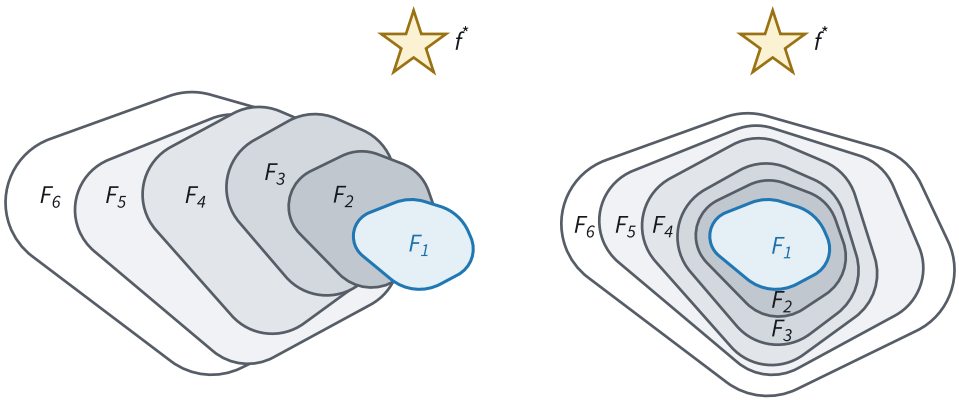


Figure 7.8: Function classes of increasing size. On the left the classes are not nested: a larger class need not contain the functions of a smaller one, and need not come closer to the target f^*. On the right each class contains the previous ones, so the best function available can only improve as the class grows.

Adding a layer nests the function classes whenever the added layer can represent the identity mapping $f(\mathbf{x}) = \mathbf{x}$. A plain block $\mathbf{x} \mapsto \mathrm{ReLU}(\mathrm{BN}(\mathbf{W}\mathbf{x} + \mathbf{b}))$ *can* represent the identity on nonnegative inputs, so plain networks already lie in the nested right-hand panel of Figure 7.8—their degradation is not a capacity failure, but an optimization failure: SGD struggles to approximate the identity across dozens of nonlinear layers. A residual block reparameterizes the layer so the identity sits at the origin $g(\mathbf{x}) = 0$ where the residual branch weights vanish.

7.4.2 Residual Blocks

Instead of learning a target mapping $f(\mathbf{x})$ directly, a residual block (Figure 7.9) learns the *residual* $g(\mathbf{x}) = f(\mathbf{x}) - \mathbf{x}$ and adds the input back through a *shortcut* (or *skip*) connection, outputting $\mathbf{x} + g(\mathbf{x})$. Driving the block to the identity requires only zeroing out the final layer of g.

Shortcuts also change how gradients travel. For two stacked blocks $\mathbf{x}_1 = \mathbf{x}_0 + g_1(\mathbf{x}_0)$ and $\mathbf{x}_2 = \mathbf{x}_1 + g_2(\mathbf{x}_1)$, the Jacobian $\partial \mathbf{x}_2 / \partial \mathbf{x}_0 = (\mathbf{I} + \partial g_2 / \partial \mathbf{x}_1)(\mathbf{I} + \partial g_1 / \partial \mathbf{x}_0)$ expands into the identity matrix $\mathbf{I}$ plus branch terms, so loss gradients reach early blocks without multiplying

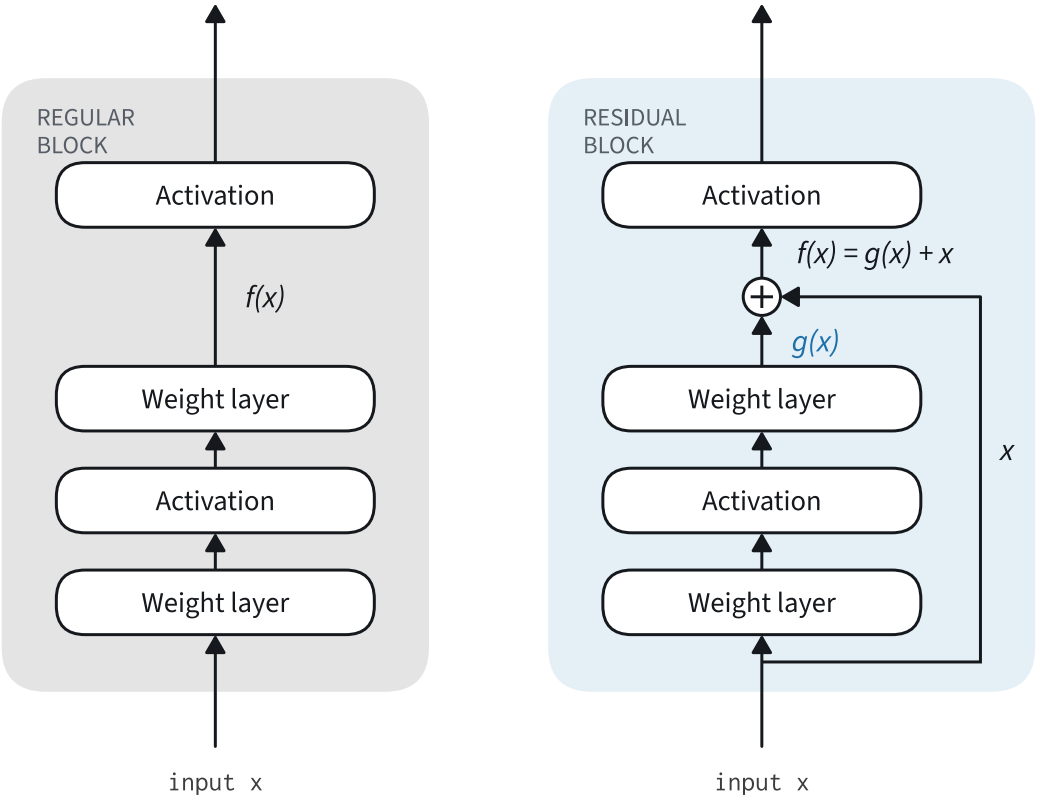


Figure 7.9: A regular block (left) must learn the mapping $f(\mathbf{x})$ directly. A residual block (right) learns the residual mapping $g(\mathbf{x}) = f(\mathbf{x}) - \mathbf{x}$ and adds the input back, so the identity mapping $f(\mathbf{x}) = \mathbf{x}$ corresponds to $g(\mathbf{x}) = 0$.

through every intermediate weight matrix. (From a continuous-time viewpoint, $\mathbf{x}_{l+1} = \mathbf{x}_l + g_l(\mathbf{x}_l)$ is a unit-step Euler discretization of an ODE $\dot{\mathbf{x}} = g(\mathbf{x}, t)$, Section 38.3.)

It is tempting to conclude that plain networks degrade because their gradients vanish. They do not: He et al. (2016a) verified that batch-normalized plain networks maintain healthy gradient norms at every depth. What fails is subtler, and later studies give two complementary accounts. In a deep plain network the gradients at nearby inputs decorrelate as depth grows until they resemble white noise, which leaves SGD without a consistent direction to follow, whereas with shortcuts the correlation decays far more slowly (Balduzzi et al. 2017). Seen from the weights, the loss surface of a deep plain network is rough and chaotic, while shortcuts keep it smooth around the minimizer (H. Li et al. 2018). The Jacobian above suggests why. Expanded over L blocks, the product of factors $\mathbf{I} + \partial g_l / \partial \mathbf{x}_{l-1}$ is a sum over 2^L paths, and most of the gradient flows along the short ones, so a ResNet trains like an ensemble of fairly shallow networks (Veit et al. 2016). Removing a single block from such an ensemble does little harm, a fact that the stochastic depth of Section 7.6 turns into a regularizer.

Concretely, ResNet's basic `Residual` block stacks two bias-free 3×3 convolutions with batch normalization and an intermediate ReLU, adds the shortcut input $\mathbf{x}$ to the second batch norm's output, and applies a final ReLU. Setting the second batch norm's $\gamma = \beta = 0$ zeroes out the branch and recovers $\mathrm{ReLU}(\mathbf{x}) = \mathbf{x}$ on nonnegative inputs. Large-scale recipes use this as an initialization: they start the last γ of every residual branch at 0, so that each block is the identity at the first step and a deep network begins training as a shallow one whose branches grow in gradually (Goyal et al. 2017; He et al. 2019). Our networks are shallow enough to train with the default $\gamma = 1$, which we keep. Whenever `use_1x1conv=True` or `strides != 1` changes channel width or spatial resolution, the shortcut applies a stride-matched 1×1 convolution and batch normalization so the addition is well defined.

```
class Residual(nn.Module):
    """The Residual block of ResNet models."""
    def __init__(self, num_channels, use_1x1conv=False, strides=1):
        super().__init__()
        # No conv biases: the batch normalization after each conv removes them
        self.conv1 = nn.LazyConv2d(num_channels, kernel_size=3, padding=1,
                                   stride=strides, bias=False)
        self.conv2 = nn.LazyConv2d(num_channels, kernel_size=3, padding=1,
                                   bias=False)
        # Auto-enable 1x1 conv when downsampling so the residual shape matches.
        if use_1x1conv or strides != 1:
            self.conv3 = nn.LazyConv2d(num_channels, kernel_size=1,
                                       stride=strides, bias=False)
            self.bn3 = nn.LazyBatchNorm2d()
        else:
            self.conv3 = None
        self.bn1 = nn.LazyBatchNorm2d()
        self.bn2 = nn.LazyBatchNorm2d()

    def forward(self, X):
        Y = F.relu(self.bn1(self.conv1(X)))
        Y = self.bn2(self.conv2(Y))
        if self.conv3:
            X = self.bn3(self.conv3(X))
        Y += X
```

```
        return F.relu(Y)
```

Figure 7.10 diagrams both configurations; passing `use_1x1conv=True` or setting a stride greater than 1 activates the projection shortcut on the right.

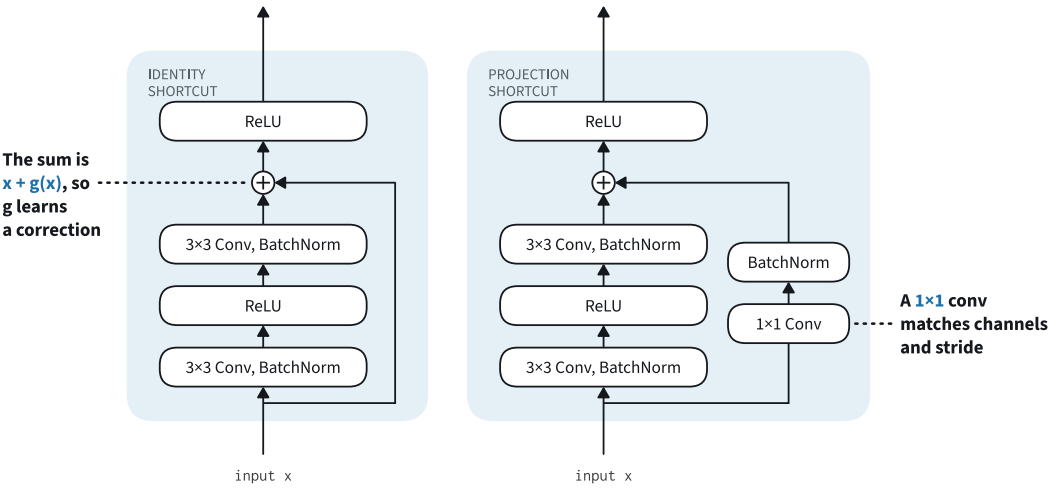


Figure 7.10: The two variants of the ResNet block: with an identity shortcut when input and output shapes match (left), and with a 1×1 convolution and batch normalization on the shortcut that adjust channels and resolution so that the addition is defined (right).

Let us verify the output shapes. With matching input and output channel counts and stride 1, the block preserves the `(4, 3, 6, 6)` tensor shape.

```
blk = Residual(3)
X = d2l.randn(4, 3, 6, 6)
blk(X).shape
```

```
torch.Size([4, 3, 6, 6])
```

Setting `num_channels=6`, `use_1x1conv=True`, and `strides=2` doubles the channels while halving both height and width—the exact downsampling block that opens stages 2, 3, and 4 of ResNet below.

```
blk = Residual(6, use_1x1conv=True, strides=2)
blk(X).shape
```

```
torch.Size([4, 6, 3, 3])
```

7.4.3 Plain and Residual Networks Compared

Can we reproduce the degradation problem directly on Fashion-MNIST and watch residual shortcuts eliminate it? To run a clean controlled test, we construct two network families that differ *solely* in the presence or absence of shortcut connections. Adapting the CIFAR-10 design of He et al. (2016a) to Fashion-MNIST's native 28×28 images keeps deep training

fast. A *plain block* mirrors `Residual` layer for layer without the bypass addition: two 3×3 convolutions, each followed by batch normalization and ReLU.

```
def plain_block(num_channels, strides=1):
    """The layers of `Residual` without the shortcut."""
    return nn.Sequential(
        nn.LazyConv2d(num_channels, kernel_size=3, padding=1, stride=strides,
                      bias=False), nn.LazyBatchNorm2d(), nn.ReLU(),
        nn.LazyConv2d(num_channels, kernel_size=3, padding=1, bias=False),
        nn.LazyBatchNorm2d(), nn.ReLU())
```

Each network begins with a 16-channel 3×3 convolution, batch normalization, and ReLU, followed by three stages of n blocks with 16, 32, and 64 channels (where the first block of stages 2 and 3 downsamples with stride 2), and terminates in global average pooling and a linear classifier. Counting the stem, two convolutions per block, and the linear head yields $6n + 2$ layers in total: 20 layers for $n = 3$ and 56 layers for $n = 9$. Passing either `plain_block` or `Residual` as `block` builds the matched pair. As before, we initialize all weights with He initialization (Section 4.4.2.2) and train using `d2l.cosine_warmup` (Section 7.1) with peak learning rate 0.1—a rate that destabilized unnormalized networks in Section 7.2 but trains smoothly here thanks to batch normalization after every convolution (Section 7.3).

```
class DeepNet(d2l.Classifier):
    """6n + 2 layers: a 3x3 stem, three stages of n blocks, a linear head."""
    def __init__(self, block, n, lr=0.1, num_steps=2345, num_classes=10):
        super().__init__()
        self.save_hyperparameters(ignore=['block'])
        layers = [nn.LazyConv2d(16, kernel_size=3, padding=1, bias=False),
                  nn.LazyBatchNorm2d(), nn.ReLU()]
        for i, num_channels in enumerate((16, 32, 64)):
            for j in range(n):  # The first block of stages 2 and 3 downsamples
                strides = 2 if i > 0 and j == 0 else 1
                layers.append(block(num_channels, strides=strides))
        layers += [nn.AdaptiveAvgPool2d((1, 1)), nn.Flatten(),
                   nn.LazyLinear(num_classes)]
        self.net = nn.Sequential(*layers)

    def weight_init(self, module):
        # The Trainer calls this after the first minibatch fixed the shapes
        if type(module) in (nn.Linear, nn.Conv2d):
            nn.init.kaiming_normal_(module.weight, nonlinearity='relu')  # He
            if module.bias is not None:
                nn.init.zeros_(module.bias)

    def configure_optimizers(self):
        optim = torch.optim.SGD(self.parameters(), lr=self.lr)
        schedule = torch.optim.lr_scheduler.LambdaLR(optim, lambda t: (
            d2l.cosine_warmup(t, self.num_steps, self.num_steps // 10)))
        return optim, schedule  # The Trainer steps it after every minibatch
```

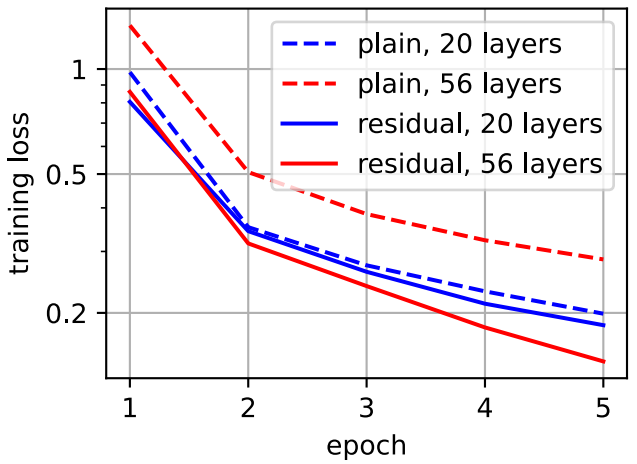
As for LeNet in Section 6.6, the Trainer calls the model's `weight_init` once the first minibatch has fixed the shapes of the lazy layers. It applies He initialization (`kaiming_normal_`) to every convolutional and fully connected layer, including those inside the `Residual` blocks.

We train all four configurations (plain vs. residual at 20 and 56 layers) for five epochs with batch size 128 and record their full training histories.

```
data = d2l.FashionMNIST(batch_size=128)
num_steps = 5 * len(data.train_dataloader())
history = {}
for name, block in (('plain', plain_block), ('residual', Residual)):
    for n in (3, 9):
        model = DeepNet(block, n, lr=0.1, num_steps=num_steps)
        trainer = d2l.Trainer(max_epochs=5, num_gpus=1)
        trainer.board.display = False  # We plot the losses ourselves
        trainer.fit(model, data)
        history[f'{name}, {6 * n + 2} layers'] = trainer.history
```

Let us overlay all four training loss curves on a logarithmic y-axis (blue for 20 layers, red for 56 layers; dashed for plain networks, solid for residual networks).

```
d2l.plot(list(range(1, 6)), [h['train_loss'] for h in history.values()],
         'epoch', 'training loss', legend=list(history), yscale='log',
         fmts=('b--', 'r--', 'b-', 'r-'))
```



We also inspect the final epoch's `train_loss` and `val_acc` recorded in `history` for all four 20- and 56-layer configurations.

```
{name: (round(h['train_loss'][-1], 3), round(h['val_acc'][-1], 3))
 for name, h in history.items()}  # Training loss, validation accuracy
```

```
{'plain, 20 layers': (0.199, 0.913),
 'plain, 56 layers': (0.285, 0.879),
 'residual, 20 layers': (0.184, 0.907),
 'residual, 56 layers': (0.145, 0.916)}
```

The degradation phenomenon appears unmistakably in the plain networks. The 56-layer plain network finishes with a training loss around 0.25–0.30, higher than the 20-layer plain

network's ~0.20, and sits strictly *above* the shallower curve at every single epoch. Because its training loss is worse, overfitting is ruled out. Turning on residual shortcuts flips the ordering completely: the 56-layer residual network drops to roughly 0.15–0.17, outperforming both 20-layer networks (which end near 0.19 with or without shortcuts). At 20 layers, shortcuts barely register; at 56 layers, they determine whether 36 additional layers hinder optimization or improve it. Since these four runs vary only depth and shortcut connections, the shortcut is unambiguously responsible. Validation accuracy mirrors the pattern more gently: the 56-layer plain network trails at 88–90%, while the other three cluster tightly around 90.5–91.5% after five epochs.

7.4.4 ResNet-18

7.4.4.1 Stem, Stages, and Head

At full ImageNet scale, ResNet adopts the stem–body–head architecture established by GoogLeNet (Section 7.2.3). Its *stem* opens with a 64-channel 7×7 convolution of stride 2, batch normalization, ReLU, and 3×3 max-pooling of stride 2, cutting spatial dimensions fourfold right away. Below, `ResNet.stem` builds this entry module.

```
class ResNet(d2l.Classifier):
    def stem(self):
        return nn.Sequential(
            nn.LazyConv2d(64, kernel_size=7, stride=2, padding=3, bias=False),
            nn.LazyBatchNorm2d(), nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1))

    weight_init = DeepNet.weight_init  # He initialization
```

The *body* consists of four *stages*, each stacking `num_residuals` residual blocks at a fixed channel width. Because the stem has already downsampled the input fourfold, stage 1 keeps its resolution and 64 channels intact; every subsequent stage opens with a stride-2 projection block that halves height and width while doubling channel count. The `stage` method constructs one such stage.

```
@d2l.add_to_class(ResNet)
def stage(self, num_residuals, num_channels, first_block=False):
    blk = []
    for i in range(num_residuals):
        if i == 0 and not first_block:
            blk.append(Residual(num_channels, use_1x1conv=True, strides=2))
        else:
            blk.append(Residual(num_channels))
    return nn.Sequential(*blk)
```

Rounding out the model, the *head* follows NiN and GoogLeNet: global average pooling followed by a single linear classifier layer. Just like `VGG`, the `arch` argument specifies `(num_residuals, num_channels)` for each stage, all convolutional and linear weights use He initialization, and the lone bias in the output layer starts at zero.

```
@d2l.add_to_class(ResNet)
def __init__(self, arch, lr=0.1, num_steps=4690, num_classes=10):
    super(ResNet, self).__init__()
    self.save_hyperparameters()
    self.net = nn.Sequential(self.stem())
    for i, b in enumerate(arch):
        self.net.add_module(f'stage{i+1}', self.stage(*b, first_block=(i==0)))
    self.net.add_module('head', nn.Sequential(
        nn.AdaptiveAvgPool2d((1, 1)), nn.Flatten(),
        nn.LazyLinear(num_classes)))
```

Placing two residual blocks in each of the four stages yields *ResNet-18*: four stages $\times$ four 3×3 convolutions (excluding the 1×1 shortcut projections) plus the 7×7 stem convolution and the final linear layer add up to 18 parameterized layers. ResNet-34 uses `(3, 4, 6, 3)` blocks per stage, while deeper members (ResNet-50, -101, and -152) swap in three-layer *bottleneck* blocks ($1\times 1 \to 3\times 3 \to 1\times 1$), which coincide with the single-group case of the ResNeXt block introduced below. Figure 7.11 diagrams ResNet-18 end to end.

```
class ResNet18(ResNet):
    def __init__(self, lr=0.1, num_steps=4690, num_classes=10):
        super().__init__(((2, 64), (2, 128), (2, 256), (2, 512)),
                         lr, num_steps, num_classes)
```

Let us trace the tensor shapes of a single-channel 96×96 image through ResNet-18 as resolution steps down and channel width expands toward global average pooling.

```
ResNet18().layer_summary((1, 1, 96, 96))
```

```
Sequential              (1, 64, 24, 24)
Sequential              (1, 64, 24, 24)
Sequential              (1, 128, 12, 12)
Sequential              (1, 256, 6, 6)
Sequential              (1, 512, 3, 3)
Sequential              (1, 10)
```

7.4.4.2 Training

To train `ResNet18`, we first attach `DeepNet`'s cosine warmup SGD schedule via `configure_optimizers`.

```
ResNet.configure_optimizers = DeepNet.configure_optimizers  # Same schedule
```

We now fit `ResNet18` on Fashion-MNIST resized to 96×96 for 10 epochs with batch size 128 and peak learning rate `lr=0.1`.

```
data = d2l.FashionMNIST(batch_size=128, resize=(96, 96))
model = ResNet18(lr=0.1, num_steps=10 * len(data.train_dataloader()))
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
trainer.fit(model, data)
```

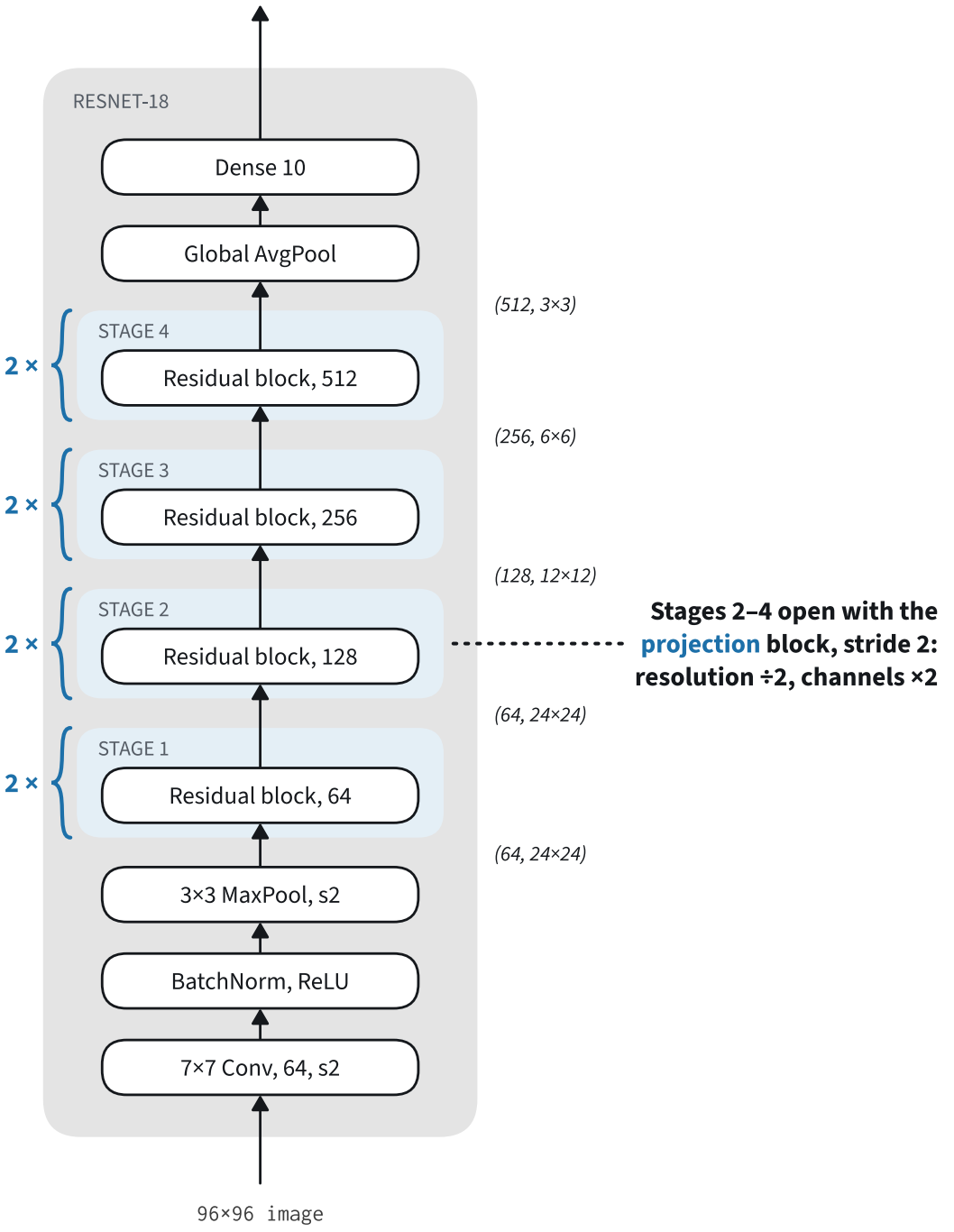
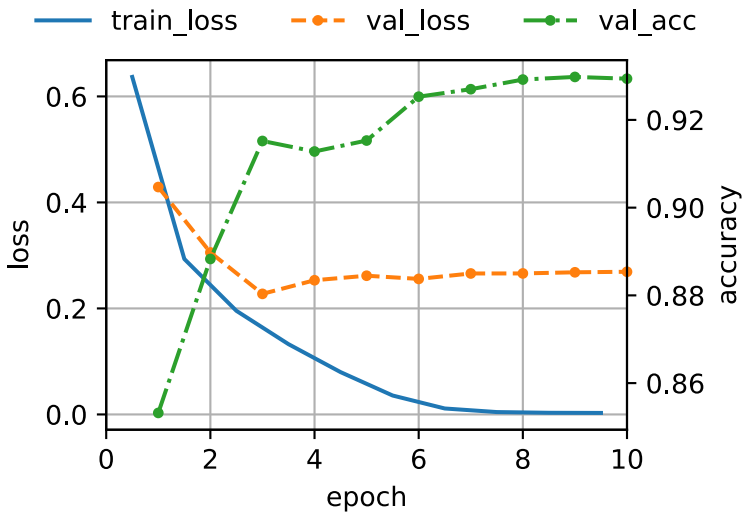


Figure 7.11: ResNet-18: a 7×7 convolutional stem, four stages of two residual blocks each, where the first block of stages 2, 3, and 4 halves the resolution and doubles the channels, and a head of global average pooling and one fully connected layer. Shapes are those of a 96×96 input.



Reading the final entry of `trainer.history['val_acc']` gives the validation accuracy at the end of the 10-epoch cosine schedule.

```
round(trainer.history['val_acc'][-1], 3)  # Validation accuracy after 10 epochs
```

```
0.929
```

After 10 epochs, ResNet-18 reaches roughly 93% validation accuracy (varying by less than a point across the final three epochs)—matching the quarter-width VGG from Section 7.2 with only a quarter as many parameters and at 96×96 instead of 224×224 pixels. As always, we should be careful not to over-interpret a single cross-family comparison where architecture, normalization, and input resolution all changed together; the clean proof of the residual connection's value lives in the paired 20- vs. 56-layer experiment of Section 7.4.3.

Notice also a classic sign of overfitting: ResNet-18 drives training loss below 0.01 while validation loss plateaus around 0.27 after the first few epochs. With 11.2 million parameters trained on 60,000 grayscale garments, memorizing the training set is easy; closing that generalization gap requires stronger data augmentation and regularization, which we tackle systematically in Section 7.6.

7.4.5 ResNeXt and DenseNet

Residual connections specify *how* a block forwards its input—by adding it to the block's learned transformation. Two prominent successors preserve the spirit of shortcut wiring while modifying the block itself: *ResNeXt* redesigns the block's internal convolutions to increase channel width without raising compute, while *DenseNet* replaces additive shortcuts with channel concatenation. We will revisit the ResNeXt block when exploring systematic network design spaces in Section 7.8.

7.4.5.1 ResNeXt

Wider layers learn richer representations, yet the cost of a standard convolution mapping c_i channels to c_o channels grows with $c_\text{i} c_\text{o}$ (Section 6.4), so doubling width quadruples both

weights and arithmetic. *ResNeXt* (Xie et al. 2017) adapts GoogLeNet's multi-branch Inception idea (Section 7.2.3) into a completely uniform structure where every branch has the exact same topology.

Specifically, a *grouped convolution* divides the c_i input channels into g disjoint groups of c_i/g channels and convolves each group independently into c_o/g output channels (assuming g divides both c_i and c_o). Each group requires a kernel of size $(c_\text{i}/g) \times (c_\text{o}/g) \times k \times k$, reducing total parameters and multiply-adds across all g groups by a factor of g, from $c_\text{i} c_\text{o} k^2$ down to $c_\text{i} c_\text{o} k^2/g$. (In practice, wall-clock speedup is typically less than g because memory traffic does not shrink at the same rate as arithmetic, a distinction we examine in Section 7.5.) Historically, grouped convolutions made their debut in AlexNet (Krizhevsky et al. 2012), which partitioned three of its five convolutional layers into two groups to fit across two GPUs.

Because independent groups cannot exchange information on their own, the ResNeXt block (Figure 7.12) sandwiches a grouped 3×3 convolution between two dense 1×1 convolutions: the first projects c input channels to b *bottleneck* channels, and the second projects back to c channels, mixing across all groups. The two 1×1 layers use $2cb$ weights and the grouped 3×3 layer uses $9b^2/g$. When downsampling, the stride-2 step sits on the 3×3 convolution rather than the first 1×1 projection, and the shortcut matches `Residual`.

To see what "wider at matched cost" looks like numerically, compare a standard ResNet-50 bottleneck block ($c = 256$, $b = 64$, $g = 1$), which uses $2 \cdot 256 \cdot 64 + 9 \cdot 64^2 = 69{,}632$ weights, against the ResNeXt block in Figure 7.12 ($b = 128$, $g = 32$), which uses $2 \cdot 256 \cdot 128 + 9 \cdot 128^2/32 = 70{,}144$ weights—doubling the internal bottleneck width for less than 1% extra weights. On ImageNet, this simple trade drops ResNet-50's top-1 error from 23.9% to 22.2% (Xie et al. 2017).

Below, `ResNeXtBlock` accepts the output width `num_channels` (c), the group count `groups` (g), and the bottleneck ratio `bot_mul` (b/c). Just as in `Residual`, passing `strides=2` halves height and width and automatically enables the 1×1 shortcut projection.

```
class ResNeXtBlock(nn.Module):
    """The ResNeXt block."""
    def __init__(self, num_channels, groups, bot_mul, use_1x1conv=False,
                 strides=1):
        super().__init__()
        bot_channels = int(round(num_channels * bot_mul))
        # No conv biases: the batch normalization after each conv removes them
        self.conv1 = nn.LazyConv2d(bot_channels, kernel_size=1, stride=1,
                                   bias=False)
        self.conv2 = nn.LazyConv2d(bot_channels, kernel_size=3,
                                   stride=strides, padding=1,
                                   groups=groups, bias=False)
        self.conv3 = nn.LazyConv2d(num_channels, kernel_size=1, stride=1,
                                   bias=False)
        self.bn1 = nn.LazyBatchNorm2d()
        self.bn2 = nn.LazyBatchNorm2d()
        self.bn3 = nn.LazyBatchNorm2d()
        # Auto-enable 1x1 conv when downsampling so the residual shape matches.
        if use_1x1conv or strides != 1:
            self.conv4 = nn.LazyConv2d(num_channels, kernel_size=1,
                                       stride=strides, bias=False)
            self.bn4 = nn.LazyBatchNorm2d()
```

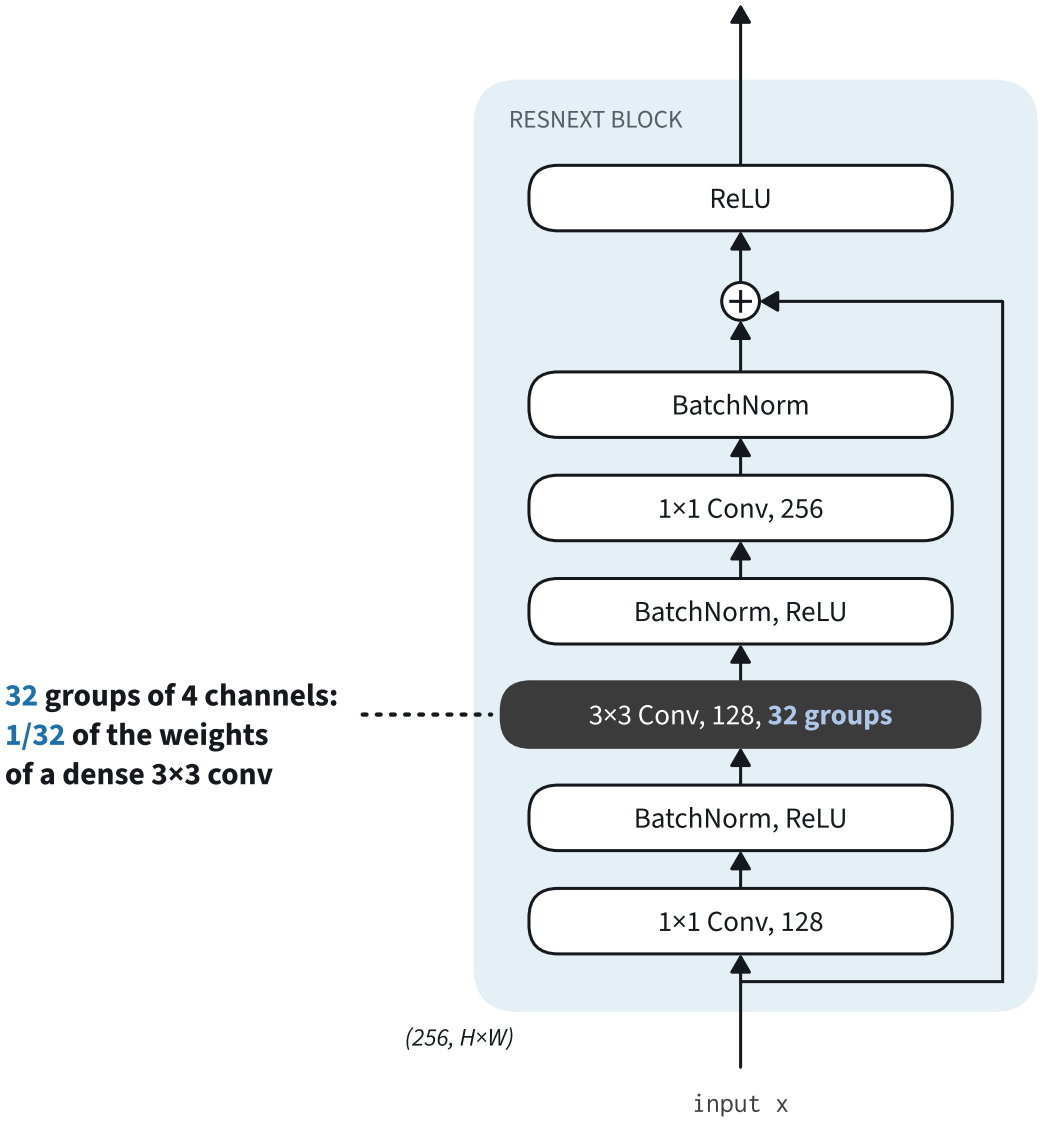


Figure 7.12: The ResNeXt block: a grouped 3×3 convolution with g groups between two 1×1 convolutions that mix information across the groups. The block shown has $c = 256$ channels, $b = 128$ bottleneck channels, and $g = 32$ groups of four channels.

```
            else:
                self.conv4 = None

        def forward(self, X):
            Y = F.relu(self.bn1(self.conv1(X)))
            Y = F.relu(self.bn2(self.conv2(Y)))
            Y = self.bn3(self.conv3(Y))
            if self.conv4:
                X = self.bn4(self.conv4(X))
            return F.relu(Y + X)
```

Let us check that a 32-channel block with 16 groups and `bot_mul=1` preserves its input tensor shape.

```
blk = ResNeXtBlock(32, 16, 1)
X = d2l.randn(4, 32, 96, 96)
blk(X).shape
```

```
torch.Size([4, 32, 96, 96])
```

7.4.5.2 DenseNet

Whereas a residual block sums its output into its input ($\mathbf{x}_k = \mathbf{x}_{k-1} + g_k(\mathbf{x}_{k-1})$), keeping channel width fixed and blending old and new features together, *DenseNet* (dense convolutional network) (Huang, Liu, et al. 2017) instead *concatenates* each layer's output onto its input along the channel dimension:

$$\mathbf{x}_k = \left[\mathbf{x}_{k-1}, f_k(\mathbf{x}_{k-1})\right],$$

where $[\cdot, \cdot]$ denotes channel-wise concatenation. Nothing is ever overwritten: $\mathbf{x}_k$ preserves the original block input together with the outputs of all k preceding layers, directly connecting every layer to every earlier layer in the block. Figure 7.13 illustrates a two-layer dense block.

Inside a dense block, each convolution adopts the pre-activation ordering (batch normalization $\rightarrow$ ReLU $\rightarrow$ bias-free convolution), explored for ResNet in the exercises below. The number of new channels appended by each layer (32 in Figure 7.13) is called the *growth rate*. Because channel width grows linearly inside every dense block, DenseNet inserts a *transition layer* between consecutive blocks—applying a 1×1 convolution to compress channel count and stride-2 average pooling to halve height and width. A complete DenseNet stacks four dense blocks separated by transition layers, mirroring ResNet-18's four stages (problem 7 walks through building and training one).

DenseNet matched ResNet's ImageNet accuracy using fewer parameters, a win its authors credited to extensive feature reuse. Its practical drawback, however, is activation memory. Even though the number of *distinct* feature maps grows linearly with block depth, a straightforward implementation allocates a fresh, wider concatenated tensor (and its normalized copy) at every layer and caches them for backpropagation, causing memory usage to scale *quadratically* with depth unless specialized implementations recompute intermediate tensors during the backward pass (Pleiss et al. 2017). Additive residual shortcuts avoid

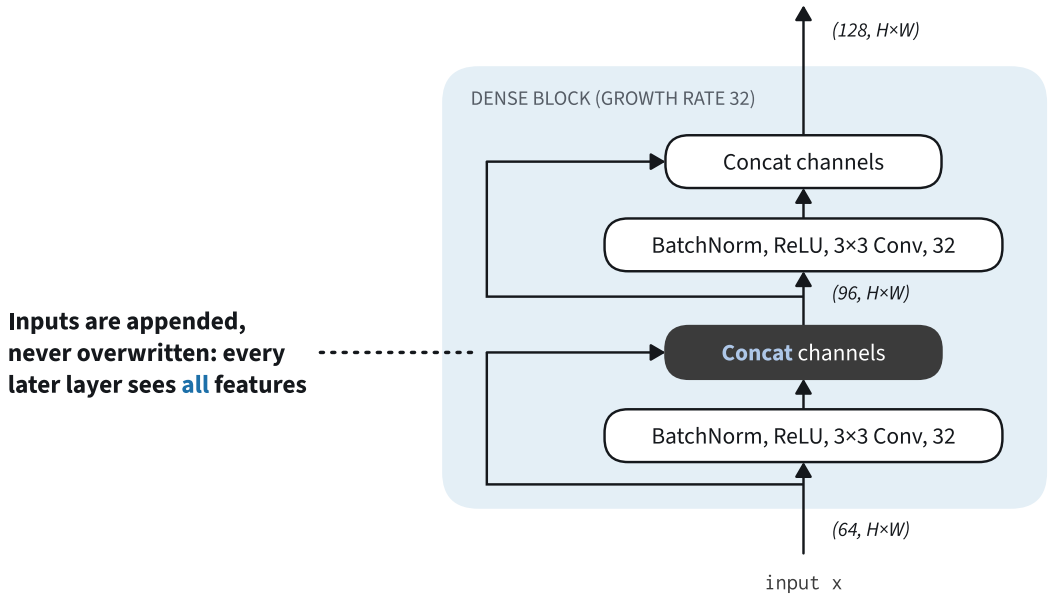


Figure 7.13: A dense block with two layers and growth rate 32. Each layer appends its 32 output channels to its input, so a 64-channel input grows to 96 and then 128 channels, and every layer sees the outputs of all layers before it.

this memory overhead entirely—one key reason why additive skip connections became the universal default across deep learning.

Concatenative skip connections nonetheless remain central in one major architectural family: the *U-Net* (Ronneberger et al. 2015) for image segmentation, which downsamples an image stage by stage and then upsamples back to full resolution, concatenating each upsampled feature map with the matching high-resolution map from the encoder path. Because concatenation happens only once per spatial scale rather than at every layer, memory stays modest; the same U-Net backbone later powered modern image diffusion models (Section 17.4).

7.4.6 Summary

Surprisingly, stacking more layers in a plain batch-normalized network can *increase* training loss relative to a shallower model, even though the deeper architecture's function class strictly contains the shallower one. Residual blocks resolve this degradation problem by learning an additive correction $\mathbf{x} + g(\mathbf{x})$: the identity mapping now sits at $g = 0$, and the shortcut keeps gradients coherent and the loss surface smooth as depth grows. On Fashion-MNIST, a 56-layer plain network trains worse than a 20-layer one, whereas with residual shortcuts the 56-layer model trains better. Arranged into four stages with batch normalization, residual blocks yield ResNet-18 (~93% validation accuracy on Fashion-MNIST in 10 epochs) and scale cleanly past 150 layers. ResNeXt widens bottleneck blocks at matched cost via grouped convolutions, while DenseNet trades additive shortcuts for feature concatenation at higher memory cost. Residual connections have since become an indispensable backbone of nearly every modern architecture, including the Transformers of Chapter 11.

7.4.7 Exercises

1. **Inception versus residual blocks.** In GoogLeNet's `arch` (Section 7.2.3), the second block of the middle stage has branch widths $c_1 = 160$, $c_2 = (112, 224)$, $c_3 = (24, 64)$, and $c_4 = 64$, mapping 512 input channels to 512 output channels. Compare it with a 512-channel `Residual` block with an identity shortcut:
   1. Count the convolutional weights and multiply-adds per output pixel of both blocks (ignoring biases and batch-norm parameters).
   2. A `Residual` block whose branch outputs zero computes the identity on nonnegative inputs (which hold whenever the block follows a ReLU); check what `bn2` outputs in training and prediction modes when `conv2` is zeroed, and state which `bn2` parameters make the block the exact identity in both modes. Can any choice of parameters make the four-branch Inception block compute the identity on nonnegative inputs?
2. **Pre-activation ordering.** He et al. (2016b) moved batch normalization and ReLU *in front* of each convolution (BN $\rightarrow$ ReLU $\rightarrow$ Conv) and removed the post-addition ReLU, so each block computes $\mathbf{x}_k = \mathbf{x}_{k-1} + g_k(\mathbf{x}_{k-1})$ with an untouched identity highway (the same ordering used inside DenseNet blocks in Section 7.4.5.2).
   1. For K stacked pre-activation blocks, express $\mathbf{x}_K$ in terms of $\mathbf{x}_0$ and the branch outputs, and show that the Jacobian $\partial \mathbf{x}_K / \partial \mathbf{x}_0$ contains the exact identity matrix $\mathbf{I}$ as an additive term.
   2. For K standard post-activation `Residual` blocks (each applying a ReLU after the addition), what factor replaces $\mathbf{I}$ in $\partial \mathbf{x}_K / \partial \mathbf{x}_0$?
3. code **Gradients of plain networks.** ● He et al. (2016a) noted that batch-normalized plain networks do not suffer from vanishing gradients despite degrading with depth. Verify this on the four 20- and 56-layer networks of Section 7.4.3:
   1. At initialization and after 1 epoch of training, compute the loss gradient on one minibatch and plot the Frobenius norm of each convolutional layer's weight gradient against its layer index for all four networks. Do the 56-layer plain network's gradients vanish toward the input?
   2. Train the 56-layer plain network for 20 epochs instead of 5. Does its training loss catch up with the 20-layer plain network?
4. code **ResNet-34 and ResNet-50 bottlenecks.** ● Table 1 of He et al. (2016a) defines ResNet-18 through ResNet-152 by stage depths and block type.
   1. Build ResNet-34 (`arch=((3, 64), (4, 128), (6, 256), (3, 512))`) using `ResNet` and compare its parameter count with `ResNet18`.
   2. ResNet-50 replaces the two-layer `Residual` block with a three-layer bottleneck block (1×1 reducing width $4\times$, 3×3, and 1×1 restoring width—implemented by `ResNeXtBlock` with `groups=1, bot_mul=0.25`) at $4\times$ wider stage outputs (`256, 512, 1024, 2048`). Compare the parameter count and multiply-adds of one ResNet-50 bottleneck block of width $4c$ against one ResNet-34 `Residual` block of width c, then override `stage` in a subclass of `ResNet` to build ResNet-50 and verify its total parameter count.
5. code **ResNet-D downsampling shortcut.** In a stride-2 `Residual` block, the 1×1 stride-2 shortcut convolution samples only one of every four spatial positions, discarding the other three. He et al. (2019) replace it with 2×2 stride-2 average pooling followed

by a stride-1 1×1 convolution and batch normalization. Implement this ResNet-D shortcut in a subclass of `Residual`, confirm that it leaves output shapes and parameter counts unchanged, and compare its 10-epoch Fashion-MNIST validation accuracy against `ResNet18` across several runs.

6. code **Pre-activation ResNet-18.** ● Implement the pre-activation block of He et al. (2016b) as a variant of `Residual`, override `stage` in a subclass of `ResNet` to stack it, and compare its training curve against `ResNet18`.

7. code **DenseNet.** ● Build and train the DenseNet of Section 7.4.5.2, then account for its parameters and activation memory.

    1. Implement a `DenseBlock` that applies `num_convs` units of $(\text{BN} \rightarrow \text{ReLU} \rightarrow \text{Conv}_{3\times3}$ without bias$)$ and concatenates each output of `growth_rate` channels to its input along the channel axis, together with a transition layer $(\text{BN} \rightarrow \text{ReLU} \rightarrow \text{Conv}_{1\times1}$ without bias $\rightarrow \text{AvgPool}_{2\times2,s=2})$. Check on a 3-channel input that a dense block with 2 convolutions and growth rate 10 returns $3 + 10 + 10 = 23$ channels, and a transition layer with 10 channels reduces them to 10 while halving height and width.
    2. Assemble a DenseNet from `ResNet`'s stem, four dense blocks of 4 convolutions each with growth rate 32, transition layers that halve channels between consecutive dense blocks, and BN $\rightarrow$ ReLU before `ResNet`'s head. Train it on 96×96 Fashion-MNIST and compare its parameter count, validation accuracy, and time per epoch with `ResNet18`.
    3. Consider a dense block whose n convolutions each emit r channels on a c-channel input, versus n `Residual` blocks of constant width c. In a naive implementation that keeps every concatenated input tensor for the backward pass, how many channels does the i-th convolution consume, and what is their sum over $i = 1, \dots, n$? Compare this sum with the $2nc$ channels kept by n residual blocks and the $c + nr$ distinct channels stored by Pleiss et al. (2017) for $c = 64, r = 32, n = 6$.

7.5 Efficient Convnets

The architectures we studied so far were engineered to top academic accuracy leaderboards using datacenter GPUs. In real-world deployment, however, vast numbers of convolutional networks run on smartphones, smart cameras, autonomous vehicles, and embedded boards where latency, memory footprint, and battery draw are hard budgets: a model that busts its budget simply cannot ship, regardless of its benchmark accuracy. Consider a phone classifying frames from its camera in real time. VGG-16 requires 15.3 billion multiply-adds for a single 224×224 frame, which at 30 frames per second translates to 460 billion multiply-adds every second, while its 138 million single-precision parameters consume 550 MB of RAM alongside every other app on the device. How can we shrink a convolutional network's cost by an order of magnitude, and what accuracy tradeoff does that entail?

In this section, we explore three major ideas behind efficient convnet design. *Factorization* decomposes each dense convolution into a lightweight per-channel spatial filter paired with a 1×1 channel mixer. *Scaling and search* fix a strong building block and optimize stage depths, widths, and input resolution systematically. *Structural re-parameterization* trains a multi-branch architecture for easy optimization and then algebraically merges those branches into a single fast convolution for deployment. Along the way, we bench-

mark factorization directly in both accuracy and wall-clock time—and uncover a sobering lesson: theoretical multiply-add counts rarely tell the whole speed story, and whether factorization beats simply narrowing a dense model depends strongly on the scale of the task.

```
import time
```

7.5.1 Depthwise-Separable Networks

Because dense spatial convolutions consume the lion's share of a convnet's arithmetic, the most direct path to efficiency is factorizing the convolution layer itself.

7.5.1.1 Depthwise-Separable Convolutions

A standard dense convolution scales with $c_\text{i}c_\text{o}$ because every output channel reads every input channel through its own spatial filter. In Section 7.4.5.1, ResNeXt's grouped convolution broke that all-to-all coupling by partitioning channels into g disjoint groups, restricting each output group to its matching input group and dividing both weights and multiply-adds by g. As Figure 7.14 illustrates, grouping turns the dense channel-connectivity matrix into a block-diagonal pattern.

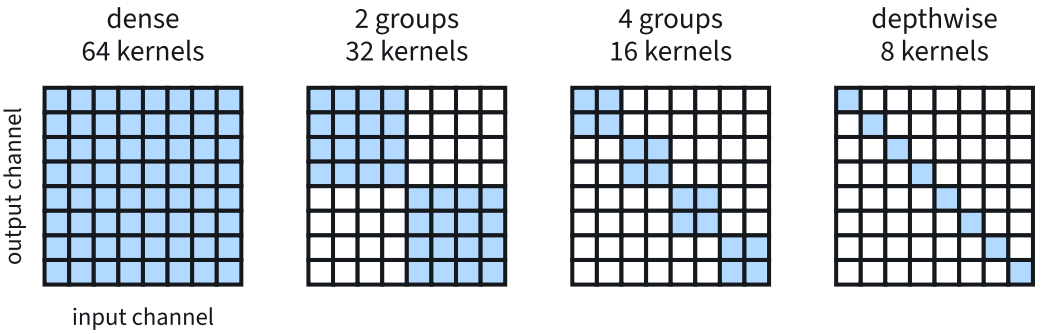


Figure 7.14: Input channels read by each output channel, for 8 input and 8 output channels. Each shaded cell is one $k_\text{h} \times k_\text{w}$ kernel; g groups keep $1/g$ of them.

Now push grouping all the way to its limit: $g = c_\text{i} = c_\text{o}$, so every group contains a single channel. This *depthwise convolution* applies one $k \times k$ spatial filter independently to each input channel with zero cross-channel mixing—the exact complement of a 1×1 convolution (Section 6.4.3), which mixes all channels at each pixel with zero spatial interaction. Chaining a depthwise $k \times k$ convolution and a *pointwise* 1×1 convolution recovers both spatial filtering and channel mixing in two cheap steps, yielding a *depthwise-separable convolution* (Chollet 2017; Howard et al. 2017). Figure 7.15 contrasts this two-stage factorization against the dense convolution it replaces.

Let us quantify the savings. For an $h \times w$ output and a $k \times k$ kernel, a dense convolution requires $hw \cdot c_\text{i}c_\text{o}k^2$ multiply-adds, whereas the depthwise-separable pair requires $hw \cdot c_\text{i}k^2$ for spatial filtering plus $hw \cdot c_\text{i}c_\text{o}$ for pointwise mixing. Their ratio is

$$\frac{hw \cdot c_\text{i}k^2 + hw \cdot c_\text{i}c_\text{o}}{hw \cdot c_\text{i}c_\text{o}k^2} = \frac{1}{c_\text{o}} + \frac{1}{k^2}, \tag{7.6}$$

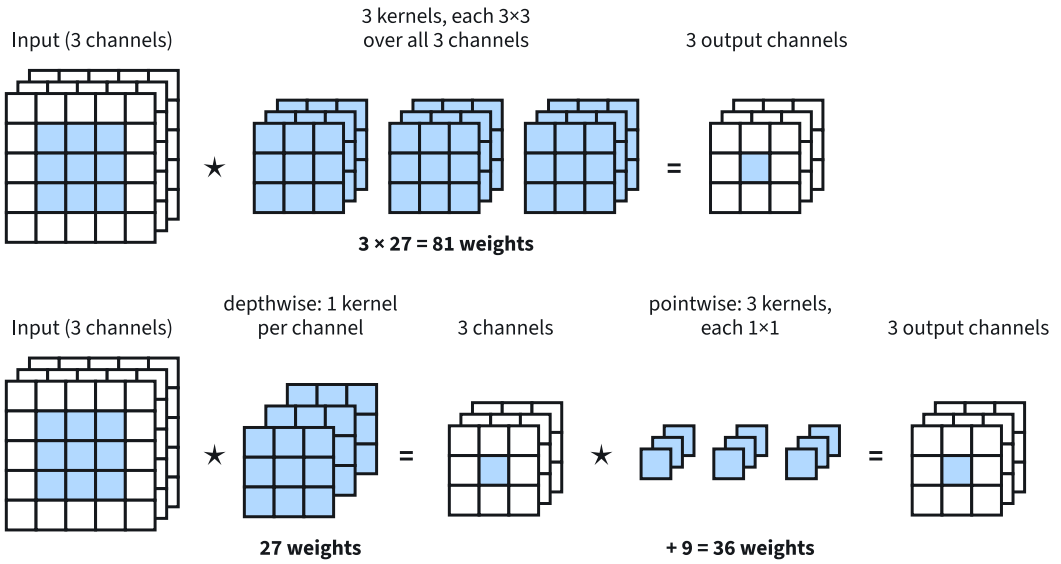


Figure 7.15: Three input and three output channels with 3×3 kernels. Top: a dense convolution, in which every output channel reads every input channel through a kernel stack of its own. Bottom: a depthwise convolution filters each channel separately, once for all output channels, and a pointwise convolution mixes the channels with one 1×1 kernel stack per output channel.

and the exact same ratio holds for parameter counts. In Figure 7.15 ($c_\text{i} = c_\text{o} = k = 3$), the dense layer uses $3 \cdot 27 = 81$ weights versus $27 + 9 = 36$ for the separable pair (ratio $1/3 + 1/9 = 4/9$). Adding one more output channel costs 27 new weights in the dense layer versus only 3 in the separable layer. Once c_o is large and $k = 3$, the ratio approaches $1/9$: nearly a ninefold reduction in both weights and arithmetic.

Naturally, these savings are not free—they come from a structural constraint on the effective filter tensor. Because the separable pair applies a single spatial filter $\mathbf{D}_c$ to input channel c and then linearly combines those filtered maps with scalar pointwise weights $P_{o,c}$, it is equivalent to a dense convolution whose kernels are forced into the factored form $\mathbf{K}_{o,c} = P_{o,c}\,\mathbf{D}_c$. Every output channel must share the same spatial filter on input channel c (up to a scalar multiple), whereas a dense convolution can learn an arbitrary independent $k \times k$ filter for every (o, c) pair. Efficient networks accept this rank-1 restriction and reinvest the freed budget into deeper or wider stages.

Let us verify the parameter ratio in code for a bias-free 3×3 layer mapping 128 channels to 128 channels, where Equation 7.6 predicts a reduction factor of $(1/128 + 1/9)^{-1} \approx 8.4$.

```
c_i, c_o, k = 128, 128, 3
X = d2l.normal(0, 1, (1, c_i, 32, 32))
dense = nn.LazyConv2d(c_o, kernel_size=k, padding=1, bias=False)
depthwise = nn.LazyConv2d(c_i, kernel_size=k, padding=1, groups=c_i,
                          bias=False)
pointwise = nn.LazyConv2d(c_o, kernel_size=1, bias=False)
assert dense(X).shape == pointwise(depthwise(X)).shape
```

```
p_dense = dense.weight.numel()
p_sep = depthwise.weight.numel() + pointwise.weight.numel()
p_dense, p_sep, p_dense / p_sep
```

```
(147456, 17536, 8.408759124087592)
```

The dense layer uses 147,456 weights versus 17,536 for the separable pair—an exact $8.4\times$ reduction.

In PyTorch, `groups=c_i` turns `nn.Conv2d` into a depthwise convolution; any `groups` that divides both channel counts gives a grouped convolution.

7.5.1.2 MobileNet

MobileNet (Howard et al. 2017) scaled depthwise-separable convolutions into a complete mobile-grade backbone. After a single dense stem convolution, every layer uses the block in Figure 7.16: a depthwise 3×3 convolution followed by a pointwise 1×1 convolution, each paired with batch normalization and ReLU. Spatial downsampling is handled directly by stride-2 depthwise convolutions (avoiding separate pooling layers), and the model ends with NiN's global average pooling and a single linear layer (Section 7.2.2). The ImageNet payoff was striking: MobileNet came within one percentage point of VGG-16 (70.6% vs. 71.5% top-1 accuracy) using only 0.57 billion multiply-adds and 4.2 million parameters—$27\times$ less arithmetic and $33\times$ fewer parameters.

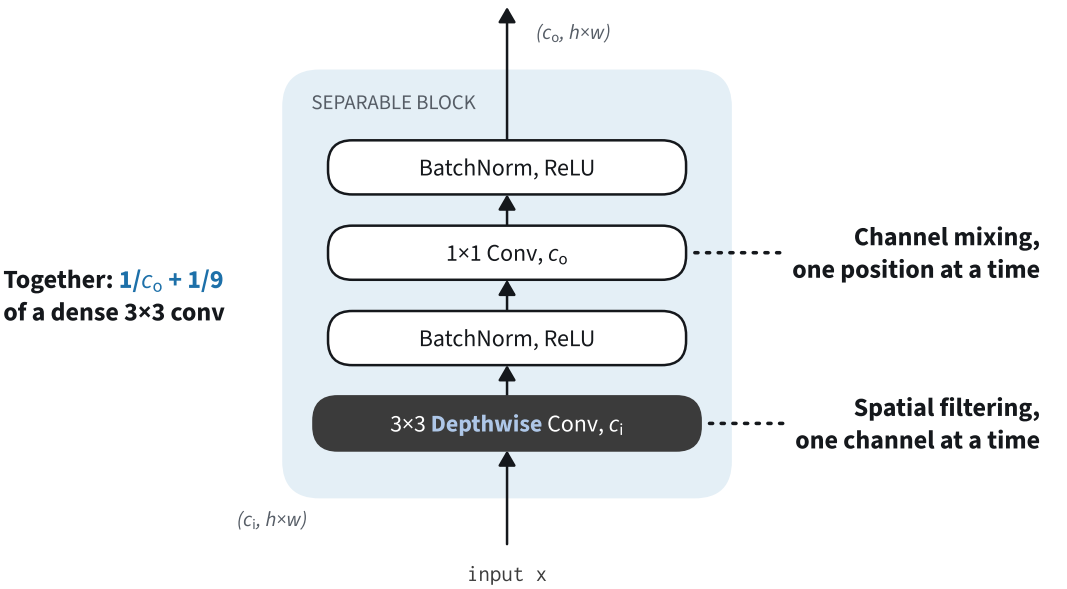


Figure 7.16: A depthwise-separable block from c_i to c_o channels. The depthwise 3×3 convolution filters each input channel separately; the pointwise 1×1 convolution mixes the channels at each position; batch normalization and a ReLU follow each convolution.

7.5.1.3 The Inverted Bottleneck

Recall that a classic ResNet or ResNeXt bottleneck (Section 7.4.5.1) *compresses* channels before its spatial filter: a 1×1 layer squeezes a wide representation into a narrow subspace so that the expensive dense 3×3 convolution stays affordable, and a second 1×1 layer expands back out. When the 3×3 layer is *depthwise*, however, that logic flips completely. A depthwise convolution is inexpensive even on wide feature maps, and because it learns only one spatial filter per channel, it actually benefits from running at high width.

MobileNetV2 (Sandler et al. 2018) therefore *inverts* the bottleneck. The residual highway stays narrow (say, 24 channels); inside the block, a 1×1 expansion lifts it sixfold to 144 channels, a depthwise 3×3 convolution filters that expanded space, and a linear 1×1 projection compresses back down to 24 channels. Crucially, the final projection omits ReLU: applying a ReLU to a very thin bottleneck zeroes out negative coordinates and permanently discards information that later layers cannot recover. Figure 7.17 places the standard and inverted bottlenecks side by side.

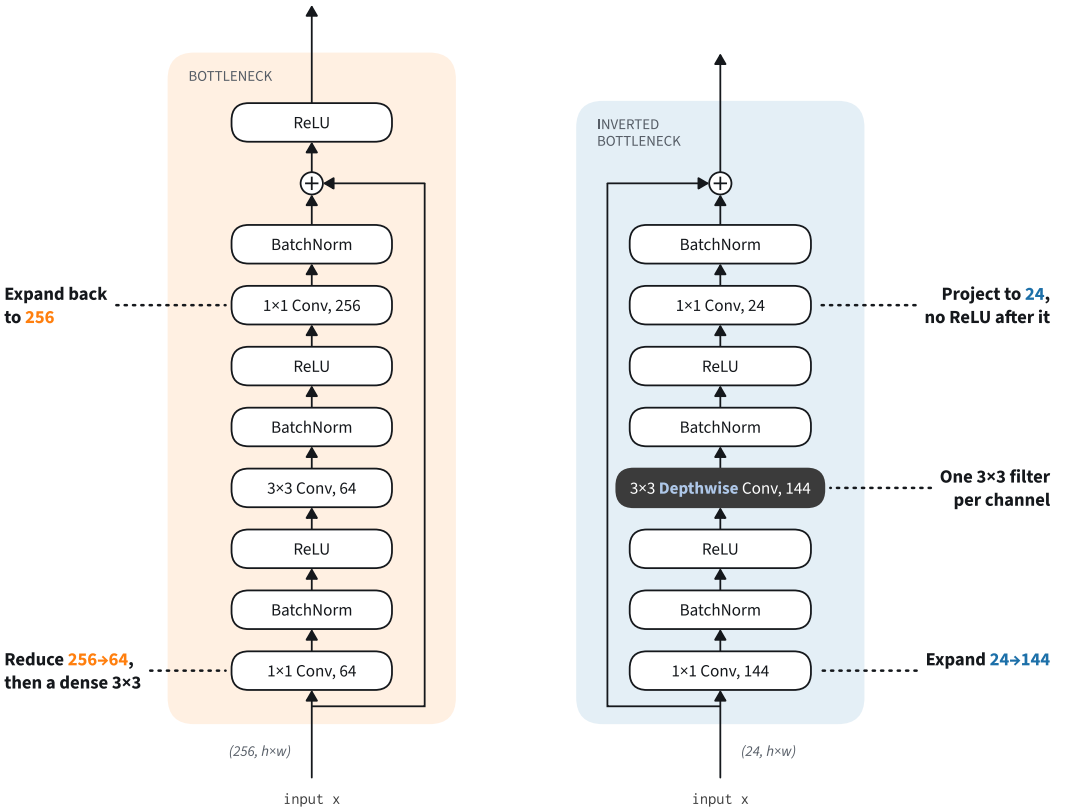


Figure 7.17: A ResNet bottleneck block (left) and a MobileNetV2 inverted bottleneck (right); the input is at the bottom. The bottleneck narrows 256 channels to 64 for a dense 3×3 convolution and widens them again. The inverted bottleneck widens 24 channels to 144 for a depthwise 3×3 convolution, which learns one filter per channel, and projects back to 24 without a ReLU; the residual connection joins the thin ends.

Because the residual shortcut connects the thin ends rather than the wide interiors, only compact tensors persist in memory between blocks—a major win on mobile processors

where activation RAM is often tighter than flash storage for weights. We will encounter this exact inverted-bottleneck geometry again (with a larger 7×7 depthwise kernel moved to the front) in ConvNeXt (Section 7.7).

7.5.1.4 A Mini-MobileNet

To test depthwise-separable factorization firsthand on Fashion-MNIST, we build a compact MobileNetV1-style network using the clean block of Figure 7.16 so the factorization itself stays front and center. Because every convolution is followed by batch normalization, all convolutions omit biases, and all convolutional and linear weights use He initialization (Section 4.4).

```
def dws_block(in_channels, out_channels, stride=1):
    """Depthwise 3x3 and pointwise 1x1 convolutions, each with BN and ReLU."""
    return nn.Sequential(
        nn.Conv2d(in_channels, in_channels, kernel_size=3, stride=stride,
                  padding=1, groups=in_channels, bias=False),
        nn.BatchNorm2d(in_channels), nn.ReLU(),
        nn.Conv2d(in_channels, out_channels, kernel_size=1, bias=False),
        nn.BatchNorm2d(out_channels), nn.ReLU())

def he_init(module):
    """Normal weights with variance 2/n_in and zero biases."""
    if isinstance(module, (nn.Conv2d, nn.Linear)):
        nn.init.kaiming_normal_(module.weight)
        if module.bias is not None:
            nn.init.zeros_(module.bias)
```

As in VGG (Section 7.2.1), the architecture is specified cleanly as a list of `(out_channels, stride)` pairs. A stride-2 3×3 stem lifts the single-channel input to 32 channels at half resolution; seven depthwise-separable blocks then double the channels and halve spatial resolution three times; and global average pooling plus a linear head emit the ten class logits.

Here we also upgrade the optimizer to SGD with momentum 0.9, which accumulates an exponentially weighted moving average of past gradients so effective step sizes can reach up to ten times a plain SGD step at the same learning rate (Section 9.4). Because of this change in optimizer, do not compare the raw accuracies here directly with the plain-SGD ResNet-18 runs of Section 7.4. As before, the learning rate anneals smoothly to zero along a half-cosine (Section 7.1) over the `num_steps` training steps (4,690 minibatches for 10 epochs), here using each framework's built-in cosine scheduler without warmup.

```
class MiniMobileNet(d2l.Classifier):
    def __init__(self, arch=((64, 1), (128, 2), (128, 1), (256, 2), (256, 1),
                             (512, 2), (512, 1)),
                 lr=0.05, num_steps=4690, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        layers = [nn.Conv2d(1, 32, kernel_size=3, stride=2, padding=1,
                            bias=False),
                  nn.BatchNorm2d(32), nn.ReLU()]
```

```
        c = 32
        for c_out, stride in arch:
            layers.append(dws_block(c, c_out, stride))
            c = c_out
        layers += [nn.AdaptiveAvgPool2d((1, 1)), nn.Flatten(),
                   nn.Linear(c, num_classes)]
        self.net = nn.Sequential(*layers)
        self.net.apply(he_init)

    def configure_optimizers(self):
        optim = torch.optim.SGD(self.parameters(), lr=self.lr, momentum=0.9)
        return optim, torch.optim.lr_scheduler.CosineAnnealingLR(
            optim, self.num_steps)  # Stepped after every minibatch
```

Let us count `MiniMobileNet`'s trainable parameters and compare against a dense counterpart; we save `num_params` in `d2l` for reuse in later sections.

```
def num_params(model):
    """The number of trainable parameters."""
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

model = MiniMobileNet()
assert model(torch.zeros(1, 1, 96, 96)).shape == (1, 10)
num_params(model)
```

```
542474
```

`MiniMobileNet` contains 542,474 trainable parameters. Replacing each separable block with an ordinary dense 3×3 convolution between the same widths would require $9 \sum_l c_{\mathrm{i},l}\, c_{\mathrm{o},l} = 4{,}663{,}296$ weights in the seven blocks alone—$8.8\times$ the 530,528 weights of our separable layers, approaching the asymptotic $9\times$ ceiling as channel width reaches 512.

7.5.1.5 Arithmetic, Accuracy, and Time

How much accuracy does factorization trade away, and how much wall-clock speed does it actually buy? To find out, we first build `VGGSmall`, a dense VGG-style network (Section 7.2.1) matched to `MiniMobileNet`'s *parameter count*. Because each dense 3×3 layer uses roughly nine times as many weights as a separable block, matching the parameter budget caps `VGGSmall` at 128 channels where `MiniMobileNet` reaches 512. Both models share batch normalization, He initialization, a global-average-pooling head, and the exact same training schedule. Keep in mind that matching parameters lets both the layer type and channel width vary together.

```
class VGGSmall(d2l.Classifier):
    """A dense VGG-style network with the same parameter budget."""
    def __init__(self, arch=((2, 32), (2, 64), (2, 128), (2, 128)),
                 lr=0.05, num_steps=4690, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        layers, c = [], 1
        for num_convs, c_out in arch:
            for _ in range(num_convs):
```

```
                layers += [nn.Conv2d(c, c_out, kernel_size=3, padding=1,
                                     bias=False),
                           nn.BatchNorm2d(c_out), nn.ReLU()]
                c = c_out
            layers.append(nn.MaxPool2d(2))
        layers += [nn.AdaptiveAvgPool2d((1, 1)), nn.Flatten(),
                   nn.Linear(c, num_classes)]
        self.net = nn.Sequential(*layers)
        self.net.apply(he_init)

    def configure_optimizers(self):
        optim = torch.optim.SGD(self.parameters(), lr=self.lr, momentum=0.9)
        return optim, torch.optim.lr_scheduler.CosineAnnealingLR(
            optim, self.num_steps)  # Stepped after every minibatch
```

Before launching training, let us tally the exact multiply-add counts on 96×96 inputs. With "same" padding and stride s, an $h \times h$ input maps to side $\lceil h/s \rceil$, and a grouped $k \times k$ convolution with g groups costs $k^2 c_\mathrm{i}/g$ multiply-adds per output element.

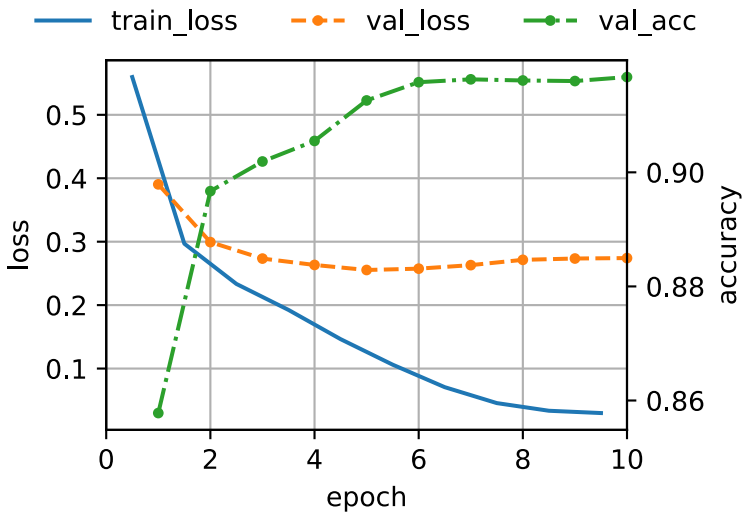
```
def conv_macs(h, c_in, c_out, k=3, stride=1, groups=1):
    """Output side and multiply-adds of a k x k convolution, 'same' padding."""
    h = -(-h // stride)  # ceil(h / stride)
    return h, h * h * k * k * c_in * c_out // groups

def mobilenet_macs(arch, h=96, num_classes=10):
    h, total = conv_macs(h, 1, 32, stride=2)  # The stem
    c = 32
    for c_out, stride in arch:
        h, depthwise = conv_macs(h, c, c, stride=stride, groups=c)
        h, pointwise = conv_macs(h, c, c_out, k=1)
        total, c = total + depthwise + pointwise, c_out
    return total + c * num_classes

def vgg_macs(arch, h=96, num_classes=10):
    total, c = 0, 1
    for num_convs, c_out in arch:
        for _ in range(num_convs):
            h, macs = conv_macs(h, c, c_out)
            total, c = total + macs, c_out
        h //= 2  # 2 x 2 max-pooling
    return total + c * num_classes

narrow = ((2, 11), (2, 23), (2, 46), (2, 46))  # VGGSmall's widths / 2.8
macs = {'Mini-MobileNet': mobilenet_macs(MiniMobileNet().arch),
        'VGG-style': vgg_macs(VGGSmall().arch),
        'VGG-style, narrow': vgg_macs(narrow)}
macs, macs['VGG-style'] / macs['Mini-MobileNet']
```

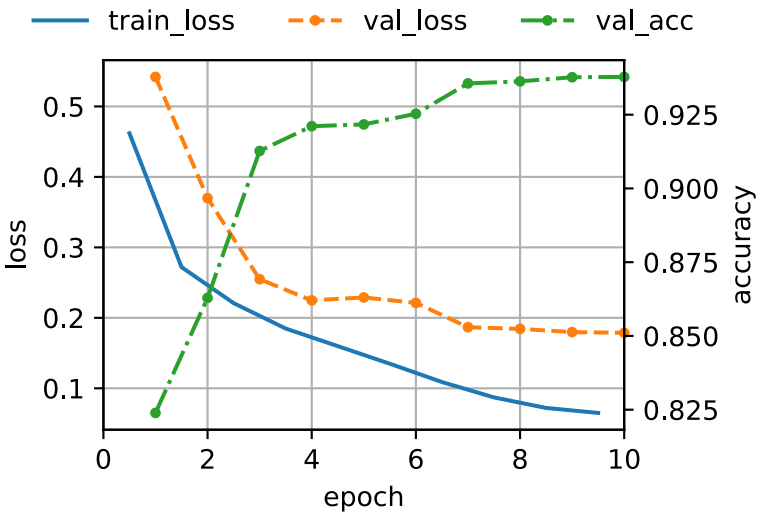
```
({'Mini-MobileNet': 50259968,
  'VGG-style': 384861440,
  'VGG-style, narrow': 49103308},
 7.657415142007253)
```

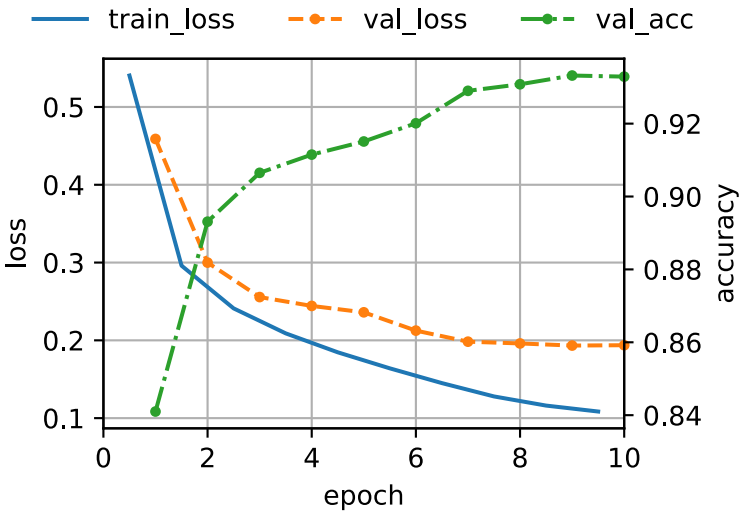
Despite matching in parameter count, `VGGSmall` executes 385 million multiply-adds per image while `MiniMobileNet` needs only 50 million—$7.7\times$ less arithmetic. That gap reflects *where* each model places its weights: `VGGSmall` applies dense filters across high-resolution 96×96 to 24×24 feature maps, whereas nearly three-quarters of `MiniMobileNet`'s weights sit in its final two pointwise layers acting on tiny 6×6 grids.

On edge hardware, however, the decisive comparison is at *equal arithmetic*. To bring `VGGSmall` down to `MiniMobileNet`'s 50 million multiply-adds, we shrink every channel count by a factor of ~2.8 to `(11, 23, 46, 46)` (`narrow` above), leaving just 76,000 parameters—one-seventh of `MiniMobileNet`'s weight count. Let us train all three models for 10 epochs on 96×96 Fashion-MNIST.

```
data = d2l.FashionMNIST(batch_size=128, resize=(96, 96))
num_steps = 10 * len(data.train_dataloader())  # 10 epochs of 469 steps
models = {'Mini-MobileNet': MiniMobileNet(lr=0.05, num_steps=num_steps),
          'VGG-style': VGGSmall(lr=0.05, num_steps=num_steps),
          'VGG-style, narrow': VGGSmall(narrow, lr=0.05, num_steps=num_steps)}
accuracy = {}
for name, model in models.items():
    trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
    trainer.fit(model, data)
    accuracy[name] = trainer.history['val_acc'][-1]
```







Operation counts measure arithmetic, not elapsed time. To see how multiply-adds translate to real latency, we benchmark inference forward passes on batches of 128 images on the GPU—warming up once and synchronizing the device before reading the wall clock (Section 5.7.2.1).

```
def forward_ms(model, num_reps=50):
    """Milliseconds per forward pass of a batch of 128 images."""
    X = torch.randn(128, 1, 96, 96, device=d2l.try_gpu())
    model.eval()
    with torch.inference_mode():
        sync = torch.cuda.synchronize if X.is_cuda else lambda: None
        model(X)
        sync()  # Warm up and wait
        start = time.perf_counter()
```

```
        for _ in range(num_reps):
            model(X)
        sync()
    return 1000 * (time.perf_counter() - start) / num_reps

for name, model in models.items():
    print(f'{name}: {num_params(model):,} parameters, '
          f'{macs[name] / 1e6:.1f}M multiply-adds, '
          f'accuracy {accuracy[name]:.3f}, '
          f'{forward_ms(model):.1f} ms/batch')
```

```
Mini-MobileNet: 542,474 parameters, 50.3M multiply-adds, accuracy 0.917, 1.8 ms/batch
VGG-style: 583,594 parameters, 384.9M multiply-adds, accuracy 0.938, 4.9 ms/batch
VGG-style, narrow: 75,854 parameters, 49.1M multiply-adds, accuracy 0.933, 1.3 ms/batch
```

Table 7.3 summarizes all three models across repeated runs, reporting validation accuracy alongside each model's forward-pass latency normalized to `MiniMobileNet`.

:Mini-MobileNet against a dense VGG-style network with about the same number of parameters and a narrow one with about the same number of multiply-adds (Fashion-MNIST at 96×96, 10 epochs; validation accuracy and forward-pass time ratio over repeated runs).

Table 7.3: Dws Vs Dense

Model	Parameters	Multiply-adds	Validation accuracy	Forward time
Mini-MobileNet	542,474	50.3 million	91–92%	1
VGG-style	583,594	384.9 million	93.5–94%	2–3.5
VGG-style, narrow	75,854	49.1 million	93%	0.5–0.8

Three lessons jump out of Table 7.3:

1. **Matching parameters is not matching compute:** At ~550K parameters, `VGGSmall` beats `MiniMobileNet` by ~2 percentage points while performing $7.7\times$ more multiply-adds (keeping full 96×96 resolution in its opening stage and learning unrestricted dense kernels).
2. **At matched arithmetic on a small dataset, narrowing beats factorizing:** Shrinking `VGGSmall` to 50M multiply-adds (`VGG-style, narrow`, 76K parameters) beats `MiniMobileNet` by ~1 point *and* runs faster. By Equation 7.6, depthwise factorization pays off most at wide channel counts (saving $5\times$ at $c = 11$ vs. nearly $9\times$ at $c = 512$); 10-class Fashion-MNIST fits comfortably inside narrow dense layers, whereas 1,000-class ImageNet requires wide stages where narrowing a dense convnet starves capacity.
3. **Multiply-adds are a weak proxy for GPU wall-clock latency:** A $7.7\times$ arithmetic reduction yields only a 2–$3.5\times$ speedup because a dense 3×3 layer performs $9c_o$ multiply-adds per input read (high arithmetic intensity), whereas a depthwise 3×3 layer performs only 9 multiply-adds per input read, leaving the GPU memory-bandwidth bound.

7.5.2 Scaling and Searching

MobileNet fixes an efficient building block and selects stage widths by hand. Two natural questions follow: how should we scale a base architecture up when more compute becomes available, and can automated optimization choose the stage widths, depths, and resolutions better than human trial and error?

7.5.2.1 Compound Scaling

Counting compute clarifies the scaling question immediately: a convnet's arithmetic grows linearly with depth, quadratically with channel width (via $c_i c_o$), and quadratically with input resolution (via hw). Pushing any single dimension alone quickly hits diminishing returns—a very deep model on tiny thumbnails lacks fine detail to analyze, while high-resolution images require more layers so receptive fields can span large objects. *Compound scaling* (Tan and Le 2019) scales all three dimensions in lockstep: each scaling step multiplies depth by 1.2, width by 1.1, and resolution by 1.15, roughly doubling total compute ($1.2 \cdot 1.1^2 \cdot 1.15^2 \approx 1.92$). A single integer knob now generates an entire family: three steps produce a network $1.7\times$ deeper and $1.3\times$ wider on $1.5\times$ larger images at roughly $7\times$ the compute.

7.5.2.2 Architecture Search

To design the base network itself, EfficientNet used *neural architecture search* (NAS) over stacks of inverted-bottleneck blocks, optimizing an objective balancing accuracy against multiply-add counts. As our GPU timing benchmark just demonstrated, optimizing theoretical multiply-adds can mislead real hardware. *EfficientNetV2* (Tan and Le 2021) replaced multiply-adds with measured training speed and parameter count in its search objective; notably, the search automatically swapped out depthwise blocks in early high-resolution stages for standard dense 3×3 convolutions, which run faster on accelerators despite performing more arithmetic. In both generations, human engineering designed the core primitives while automated search allocated widths, depths, and resolutions across stages.

7.5.2.3 Squeeze-and-Excitation Gates

The inverted-bottleneck blocks searched by EfficientNet include one extra lightweight module right after the depthwise convolution: a *squeeze-and-excitation* (SE) gate (Hu et al. 2018). Whereas a convolution aggregates information only over a local spatial window, an SE gate lets the network dynamically reweight entire feature channels using global image context. It *squeezes* each of the c channels to a single scalar via global average pooling, passes that c-dimensional summary through a tiny two-layer bottleneck MLP (hidden width c/r with ReLU, output width c with sigmoid, the *excitation*), and scales every channel of the feature map by its predicted gate in $(0, 1)$. With reduction ratio r, the MLP adds only ~$2c^2/r$ parameters and operates on a single vector per image rather than every pixel, making its arithmetic cost negligible. Conceptually, SE acts as a simple channel-wise attention mechanism—foreshadowing the general attention layers of Chapter 10.

Because SE gates deliver consistent accuracy gains at almost zero compute cost, they appear throughout MobileNetV3 and RegNetY (Section 7.8, where problem 3 adds SE to RegNet). Implementing an SE block takes only a handful of lines: global pool, two linear layers, and channel-wise rescaling.

```
class SE(nn.Module):
    def __init__(self, num_channels, ratio=4):
        super().__init__()
        self.fc = nn.Sequential(
            nn.LazyLinear(num_channels // ratio), nn.ReLU(),
            nn.LazyLinear(num_channels), nn.Sigmoid())

    def forward(self, X):
        s = self.fc(X.mean(dim=(2, 3)))
        return X * s[:, :, None, None]

SE(32)(d2l.randn(2, 32, 16, 16)).shape
```

```
torch.Size([2, 32, 16, 16])
```

With $c = 32$ and $r = 4$, the SE module adds just $2 \cdot 32^2/4 = 512$ weights and 40 biases while preserving the exact tensor shape, allowing it to drop into any block seamlessly.

7.5.3 Structural Re-parameterization

7.5.3.1 The RepVGG Block

ResNet (Section 7.4) established that multi-branch blocks with shortcut paths are dramatically easier to optimize than plain sequential stacks. At inference time, however, parallel branches carry real systems overhead: every branch must be launched separately and kept alive in memory until the summation point, lowering hardware utilization. A plain sequential stack of 3×3 convolutions, by contrast, is ideal for accelerators—one dense, highly optimized kernel per layer and a single live activation buffer. *RepVGG* (X. Ding et al. 2021) achieves the best of both worlds by decoupling training topology from inference topology. During training, every layer sums three parallel branches before the ReLU: a 3×3 convolution, a 1×1 convolution, and an identity path, each with its own batch normalization (Figure 7.18). Once training finishes, every branch is purely linear in its input, so all three are algebraically *fused into a single* 3×3 *convolution* that computes the exact same function. We train a multi-branch residual network and deploy a fast plain VGG stack.

```
class RepVGGBlock(nn.Module):
    """Train-time block: 3x3, 1x1, and identity branches, each with BN."""
    def __init__(self, num_channels):
        super().__init__()
        self.conv3 = nn.Conv2d(num_channels, num_channels, kernel_size=3,
                               padding=1, bias=False)
        self.conv1 = nn.Conv2d(num_channels, num_channels, kernel_size=1,
                               bias=False)
        self.bn3, self.bn1, self.bn0 = (nn.BatchNorm2d(num_channels)
                                        for _ in range(3))

    def forward(self, X):
        return F.relu(self.bn3(self.conv3(X)) + self.bn1(self.conv1(X))
                      + self.bn0(X))
```

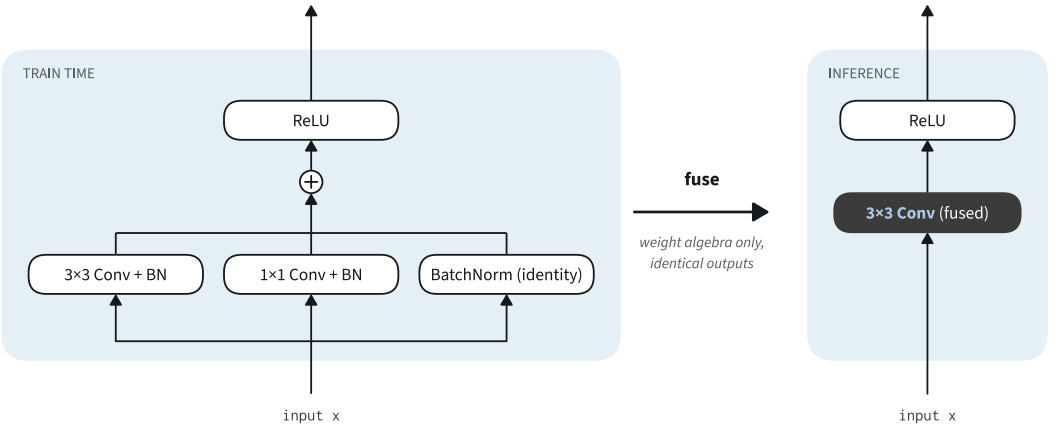


Figure 7.18: Structural re-parameterization. At training time each RepVGG layer sums three branches, a 3×3 convolution, a 1×1 convolution, and an identity, each with batch normalization. For inference the branches are folded into one 3×3 convolution with a bias; the two networks compute the same function.

7.5.3.2 Fusing the Branches

Collapsing all three branches into one convolution requires three simple steps of linear algebra. First, in inference mode batch normalization is a fixed per-channel affine map—subtracting the running mean μ_o, dividing by $\sqrt{\sigma_o^2 + \epsilon}$, multiplying by γ_o, and adding β_o. Absorbing this affine map directly into a preceding convolution with kernel $\mathbf{W}$ and bias b_o (indexed by output channel o) gives an equivalent fused kernel and bias:

$$\begin{aligned} \hat{\mathbf{W}}_o &= \frac{\gamma_o}{\sqrt{\sigma_o^2 + \epsilon}} \mathbf{W}_o, \\ \hat{b}_o &= \beta_o + \frac{\gamma_o (b_o - \mu_o)}{\sqrt{\sigma_o^2 + \epsilon}}. \end{aligned} \tag{7.7}$$

Each output channel's kernel simply scales by its normalization factor while the fused bias absorbs all input-independent offsets (for instance, $\gamma_o = 2$, $\sigma_o^2 + \epsilon = 16$, $\mu_o = 1$, $\beta_o = 1$, and $b_o = 0$ give $\hat{\mathbf{W}}_o = \mathbf{W}_o/2$ and $\hat{b}_o = 1 - 0.5 = 0.5$). Since `RepVGGBlock`'s convolutions carry no biases, `fuse_bn` below implements the $b_o = 0$ case. Note that BN folding is completely general and is routinely applied by inference compilers to any `Conv + BN` pair.

Second, we express the remaining two branches as 3×3 convolutions: a 1×1 kernel is simply a 3×3 kernel zero-padded by one pixel around its center, and the identity mapping is a 3×3 convolution whose kernel $\mathbf{W}^{\text{id}}_{o,i}$ equals 1 at the center tap when $i = o$ and 0 everywhere else. Third, by linearity of convolution, summing the outputs of three parallel convolutions on the same input is identical to a single convolution whose kernel is the sum of the three folded kernels and whose bias is the sum of the three folded biases (Figure 7.19).

```
def fuse_bn(W, bn):
    """Fold inference-mode batch normalization into a bias-free conv."""
    scale = bn.weight / torch.sqrt(bn.running_var + bn.eps)
```

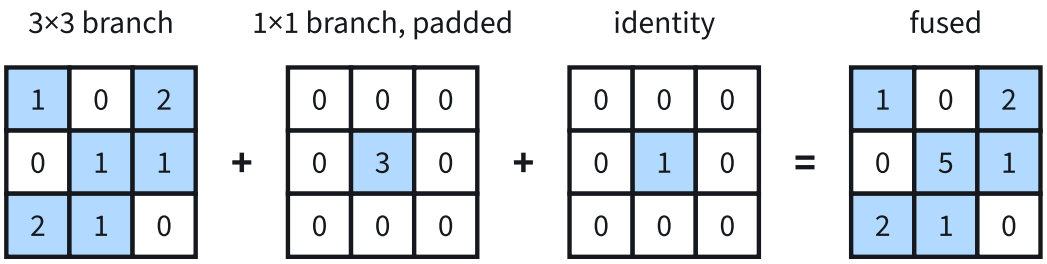


Figure 7.19: The kernels of one output channel o applied to the input channel $i = o$, after batch normalization has been folded into each branch (illustrative values; after folding, the identity's center entry is $\gamma_o/\sqrt{\sigma_o^2 + \epsilon}$ rather than 1). The padded 1×1 kernel and the identity kernel are nonzero only at the center, so the fused kernel differs from the 3×3 branch only there. For $i \neq o$ the identity kernel is zero.

```
    return W * scale.reshape(-1, 1, 1, 1), bn.bias - bn.running_mean * scale

@torch.no_grad()
def fuse_block(blk):
    c = blk.conv3.weight.shape[0]
    W3, b3 = fuse_bn(blk.conv3.weight, blk.bn3)
    W1, b1 = fuse_bn(F.pad(blk.conv1.weight, (1, 1, 1, 1)), blk.bn1)
    I = torch.zeros_like(W3)  # On the device of the block
    I[torch.arange(c), torch.arange(c), 1, 1] = 1.0
    W0, b0 = fuse_bn(I, blk.bn0)
    fused = nn.Conv2d(c, c, kernel_size=3, padding=1, device=W3.device)
    fused.weight.copy_(W3 + W1 + W0)
    fused.bias.copy_(b3 + b1 + b0)
    return fused
```

Because fusion relies on inference-mode running statistics, we test `fuse_block` rigorously: we first pass several training batches (mean 1, standard deviation 2) through `RepVGGBlock` so its running statistics depart from 0 and 1, randomize γ_o and β_o so every term in Equation 7.7 is exercised, and then compare the maximum absolute discrepancy between the multi-branch block in inference mode and the single fused convolution on fresh data.

```
blk = RepVGGBlock(8)
for _ in range(4):  # Move the BN running statistics
    blk(2 * torch.randn(32, 8, 16, 16) + 1)
with torch.no_grad():  # Nontrivial scales and shifts
    for bn in (blk.bn3, blk.bn1, blk.bn0):
        bn.weight.uniform_(0.5, 1.5)
        bn.bias.normal_()
blk.eval()
fused = fuse_block(blk)
X = 2 * torch.randn(32, 8, 16, 16) + 1
with torch.no_grad():
    error = (blk(X) - F.relu(fused(X))).abs().max()
float(error)
```

```
5.7220458984375e-06
```

The maximum discrepancy is on the order of 10^{-6} (pure single-precision floating-point roundoff), confirming exact functional equivalence. While weight count drops by only a tenth (from $10c^2$ to $9c^2$), the runtime execution simplifies dramatically: one dense convolution and one live activation tensor replace two convolutions, three normalizations, two additions, and three concurrent feature buffers.

7.5.3.3 Quantization after Fusion

Real-world deployment introduces one subtle caveat that exact floating-point algebra misses. To save energy and memory bandwidth, edge accelerators widely quantize weights and activations down to 8-bit integers, mapping each tensor's dynamic range onto just 256 uniformly spaced levels. Because RepVGG's fused kernel sums three independently batch-normalized branches whose post-folding scales can differ by orders of magnitude, the fused weights and activations often develop heavy-tailed dynamic ranges that 256 discrete bins represent poorly. The impact can be catastrophic: naive post-training 8-bit quantization drops a fused RepVGG-B0 from 75.1% down to 40.2% ImageNet top-1 accuracy, whereas a standard ResNet loses only a fraction of a point (Chu et al. 2024). Regularizing branch scales during training resolves the issue, leaving an important engineering lesson: exact algebraic equivalence in FP32 does not guarantee equivalence under low-precision quantization.

7.5.4 Summary

Efficient convnets elevate latency, memory, and energy to first-class design objectives alongside accuracy. Depthwise-separable convolutions factor spatial filtering from channel mixing, trimming layer cost to $1/c_{\mathrm{o}} + 1/k^2$ of a dense convolution, while MobileNetV2's inverted bottleneck keeps the residual highway narrow to minimize peak activation memory. On Fashion-MNIST, `MiniMobileNet` trails a parameter-matched dense network by roughly two points while performing $7.7\times$ fewer multiply-adds, yet runs only 2–$3.5\times$ faster because low-intensity depthwise layers are bottlenecked by memory bandwidth; moreover, narrowing the dense network to the same arithmetic budget beats `MiniMobileNet` in both accuracy and speed on this small task. Beyond layer factorization, compound scaling and latency-aware architecture search allocate depth, width, and resolution across stages, squeeze-and-excitation gates add global channel reweighting at negligible arithmetic cost, and structural re-parameterization trains multi-branch blocks that fold into fast plain convolutions at inference time (with care required for post-fusion quantization). Having examined efficiency by hand and by single-model search, we turn next to training recipes, systematic ResNet modernization, and population-level architecture design spaces.

7.5.5 Exercises

1. **Channel permutation between groups (ShuffleNet).** ● Grouped convolutions can be mixed without dense 1×1 weights (X. Zhang et al. 2018). Follow a grouped $k \times k$ convolution ($c_{\mathrm{i}} \to c_{\mathrm{o}}$ channels, g groups) with a fixed permutation of the c_{o} intermediate channels and a second grouped convolution ($c_{\mathrm{o}} \to c_{\mathrm{o}}$, g groups). Find the condition on g and c_{o} under which a permutation allows every output channel of the second layer to depend on all c_{i} input channels, and construct that permutation explicitly.
2. code **Width multiplier.** Add a width multiplier m to `MiniMobileNet` that scales every stage's channel count by m (Howard et al. 2017). Predict how the parameter count

and multiply-adds scale with m, and verify your prediction at $m = 0.5$ using `num_params` and `mobilenet_macs`.

3. code **Folding batch normalization with bias.** Extend `fuse_bn` to a preceding convolution that carries a nonzero bias b_o using Equation 7.7, and verify numerically that the `Conv + BN` pair in inference mode agrees with the single folded convolution to within 10^{-6}. Why can no fixed kernel and bias reproduce the output of `Conv + BN` when batch normalization runs in *training* mode?

4. code **The MobileNetV2 inverted bottleneck.** ● MobileNetV2 (Sandler et al. 2018) replaces the block of Figure 7.16 with an inverted bottleneck: a 1×1 expansion by factor $t = 6$, a depthwise 3×3 convolution, a *linear* 1×1 projection (no ReLU), and a residual shortcut whenever input and output shapes match.

    1. Implement the inverted-bottleneck block with $t = 6$ and replace the blocks of `MiniMobileNet` with it, choosing stage widths so the total parameter count stays near 542,474. Train it under the setup of Table 7.3 and compare validation accuracy and forward latency against `MiniMobileNet`.
    2. Add a ReLU after the final 1×1 projection, train both the linear and ReLU variants three times each, and check whether the linear projection outperforms the ReLU projection at this scale.
    3. For a 96×96 input, tabulate the number of elements in each block's input, output, and expanded interior tensor. Explain why an inference engine that fuses or tiles the expansion and depthwise filtering channel-by-channel never needs to materialize the full expanded tensor in RAM.

5. code **Arithmetic intensity and batch size.** ● `MiniMobileNet` performs $7.7\times$ fewer multiply-adds than `VGGSmall`, yet its GPU forward pass in Table 7.3 is only 2–$3.5\times$ faster.

    1. Define the arithmetic intensity of a layer as its multiply-adds divided by the total number of tensor elements it reads and writes (weights + input activations + output activations). Compute the arithmetic intensity of the depthwise convolution, the pointwise convolution, and an equivalent dense 3×3 convolution for every stage of `MiniMobileNet`. In which stages is the depthwise arithmetic intensity lowest, and how does this explain why EfficientNetV2 (Tan and Le 2021) replaces early depthwise stages with dense convolutions?
    2. Measure `forward_ms` for `MiniMobileNet` and `VGGSmall` across batch sizes $B \in \{1, 16, 128, 512\}$ and plot latency per image versus B. How does their speed ratio change with B, and why?

6. code **Training and quantizing RepVGG blocks.** ● Why train a three-branch `RepVGGBlock` if it fuses into a single 3×3 convolution at inference time?

    1. Extend `RepVGGBlock` and `fuse_block` to support stride-2 downsampling blocks that change channel width (keeping only the 3×3 and 1×1 branches when input and output shapes differ (X. Ding et al. 2021)). Build a small convnet from `RepVGGBlocks` and a matched plain convnet that keeps only the single 3×3 branch during training. Train both on Fashion-MNIST, compare their convergence, and verify that fusing the trained RepVGG network leaves its validation accuracy unchanged.
    2. Fold the batch normalizations of the plain network with `fuse_bn`, then quantize the weights of both deployed networks to 8-bit integers using symmetric per-

tensor scaling ($\hat{w} = s \cdot \text{round}(w/s)$ with $s = \max|w|/127$). Compare the accuracy drop of the fused RepVGG network against the plain network and relate it to the weight-distribution analysis of Chu et al. (2024).

7.6 Modern Training Recipes for Convnets

Throughout this chapter we have worked hard to hold the training procedure constant within each section so that accuracy differences could be attributed to architecture. Even under those controlled conditions, simply changing the peak learning rate shifted VGG's accuracy by 1.5 percentage points (Section 7.2). Published benchmarks are often far less controlled: a newly proposed network is trained with the latest optimization and regularization tricks, then compared against a classic baseline whose accuracy is copied from a paper written years earlier. How much of a reported breakthrough actually comes from the *training recipe* (the optimizer, learning-rate schedule, data augmentation, regularization, and training duration) rather than from the architecture itself?

ResNet-50, the deeper sibling of the ResNet-18 we built in Section 7.4, supplies a striking answer. Without altering a single layer of the 2015 architecture, Wightman et al. (2021) retrained ResNet-50 using early-2020s training practices and lifted its ImageNet top-1 accuracy from 76.1% to 80.4%—a 4.3-point jump driven purely by the training recipe. That gain rivals or exceeds the improvements claimed by many novel architectures published over the intervening years. Consequently, whenever a paper claims a two-point win over a baseline trained with an outdated recipe, we cannot tell whether the architecture or the recipe deserves the credit.

In this section, we examine what changed between 2015 and 2022, implement each core ingredient in a few lines of code, and train an identical ResNet-18 under both a classic and a modern recipe to observe the difference directly. None of these techniques is tied to convolutions: the exact same toolkit powers modern Transformer training, making this a general foundation for modern deep learning practice.

```
import random
```

7.6.1 Training Recipes from 2015 to 2022

The original 2015 ResNet recipe was already thoughtfully tuned: SGD with momentum (Section 7.5), weight decay, step-wise learning-rate decay (dividing by 10 twice), and random cropping with horizontal flips (He et al. 2016a). Over the next several years, researchers revisited every single piece of that pipeline—first individually and then in large-scale systematic combinations (He et al. 2019; Wightman et al. 2021). Table 7.4 contrasts the 2015 baseline with a representative early-2020s recipe; we unpack and implement each new component immediately below.

:A 2015 ImageNet recipe and a representative high-accuracy recipe from the early 2020s. The later column is not a universal standard; strong recipes still vary by model and compute budget.

Table 7.4: Recipe 2015 2022

Ingredient	2015 recipe	2022 recipe
Optimizer	SGD with momentum	AdamW or LAMB; tuned SGD remains competitive
Schedule	step decay (divide by 10 twice)	cosine decay with linear warmup
Augmentation	random crops, horizontal flips	RandAugment, Mixup, CutMix, random erasing
Regularization	weight decay	+ label smoothing, stochastic depth
Duration	90 epochs	300 to 600 epochs
Evaluated weights	final weights	exponential moving average of the weights

Two observations are essential when reading Table 7.4. First, its rows interact tightly rather than acting in isolation. Stronger data augmentation and regularization make the training objective harder to fit, which in turn demands far more training epochs—explaining why training schedules grew from 90 epochs to 300–600. Regularization strength and training duration must scale together: piling heavy augmentation onto a short run frequently *hurts* accuracy compared with a lighter recipe.

Second, the optimizer row reflects an expanded set of options rather than a one-size-fits-all mandate. Adaptive optimizers such as Adam (covered in full in Chapter 9) normalize each parameter's gradient by a running estimate of its magnitude so every coordinate updates at roughly the learning rate's scale. Classic implementations folded weight decay into the gradient before this rescaling, inadvertently distorting the penalty. *AdamW* (Loshchilov and Hutter 2019) decouples weight decay from the adaptive gradient step, shrinking every weight directly by a fraction equal to learning rate times decay—a clean fix that made adaptive optimizers competitive with well-tuned SGD across vision benchmarks. (LAMB is a large-batch cousin of AdamW. The 80.4% ResNet-50 above was trained with it, for 600 epochs, with a binary cross-entropy loss and without weight averaging. We will not need it here.)

7.6.2 Implementing the Ingredients

Each modern ingredient takes only a few lines of code. Below we implement label smoothing, cosine warmup schedules, Mixup, stochastic depth, and exponential weight averaging. (We omit RandAugment (Cubuk et al. 2020), which applies random sequences of standard image transforms at a tuned intensity, since in practice it is invoked as a preprocessing library call whose value lies in tuning rather than implementation.)

7.6.2.1 Label Smoothing

Under standard cross-entropy with one-hot targets, the loss on a correctly classified example keeps falling without bound as the true class's logit grows toward $+\infty$. Because no finite logits minimize the objective, the network is continually pushed toward extreme, poorly calibrated confidence. *Label smoothing* (Szegedy et al. 2016) curbs this by replacing the hard one-hot target with a softened distribution that places weight $1 - \epsilon$ on the true class y and spreads ϵ uniformly across all K classes. Writing p_k for the predicted softmax probabilities, the per-example loss becomes

$$l_\epsilon = -(1-\epsilon)\log p_y - \frac{\epsilon}{K}\sum_{k=1}^{K}\log p_k. \tag{7.8}$$

The first term is ordinary cross-entropy scaled by $1-\epsilon$, while the second penalizes pushing any class probability toward zero. The loss achieves its minimum at $p_y = 1-\epsilon+\epsilon/K$ and $p_k = \epsilon/K$ for $k \neq y$, which corresponds to a clean, finite margin between logits. Look at what happens to an already confident, correct prediction under unsmoothed versus smoothed cross-entropy:

```
logits, y = d2l.tensor([[10.0, 0.0, 0.0]]), d2l.tensor([0])
for epsilon in (0.0, 0.1):
    loss = F.cross_entropy(logits, y, label_smoothing=epsilon)
    print(f'epsilon={epsilon}: loss={float(loss):.4f}')
```

```
epsilon=0.0: loss=0.0001
epsilon=0.1: loss=0.6668
```

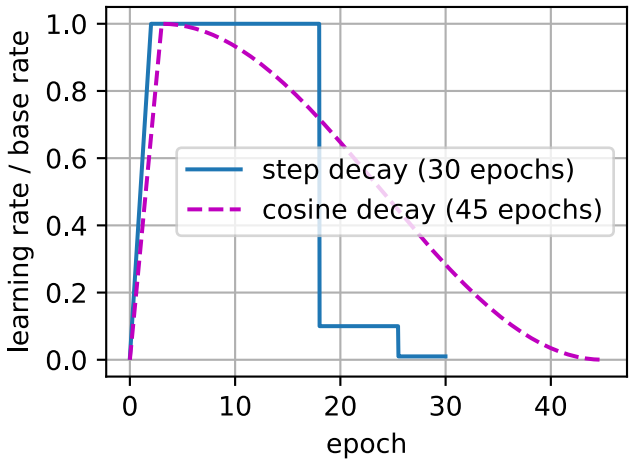
With $\epsilon = 0.1$, the loss jumps from 0.0001 to 0.67. For three classes, the minimum possible smoothed loss is roughly 0.29, attained when the winning logit leads the others by about 3.3. A margin of 10 is therefore penalized as overconfident, reversing the gradient so it pulls extreme logits back toward a moderate gap rather than pushing them further apart.

7.6.2.2 Cosine Schedules with Warmup

Traditional step schedules hold the learning rate constant for dozens of epochs and then drop it abruptly by a factor of 10—introducing the drop epochs as extra hyperparameters to tune. The modern default (Loshchilov and Hutter 2016) is already familiar from AlexNet onward (Section 7.1): a linear *warmup* from 0 to the peak rate η_0 over t_w steps, followed by a smooth half-cosine *decay* down to 0 at step T. Beyond preventing early divergence, warmup gives running batch-normalization statistics and adaptive optimizer moments time to settle before full-size steps begin, while cosine decay removes abrupt drop thresholds altogether. Below we plot `step_decay` (30 epochs, with a two-epoch warmup) alongside `d2l.cosine_warmup` (45 epochs, three-epoch warmup) at 79 minibatches per epoch, matching the exact horizons used in our experiment later in this section.

```
def step_decay(t, num_steps, warmup_steps):
    """Recipe A: linear warmup, then divide by 10 at 60% and 85% of steps."""
    warmup = min((t + 1) / warmup_steps, 1)
    return warmup * 0.1 ** sum(t >= int(f * num_steps) for f in (0.6, 0.85))

steps_per_epoch = 79
steps_a = list(range(30 * steps_per_epoch))  # Recipe A: 30 epochs
steps_b = list(range(45 * steps_per_epoch))  # Recipe B: 45 epochs
d2l.plot([[t / steps_per_epoch for t in steps_a],
          [t / steps_per_epoch for t in steps_b]],
         [[step_decay(t, len(steps_a), 2 * steps_per_epoch) for t in steps_a],
          [float(d2l.cosine_warmup(t, len(steps_b), 3 * steps_per_epoch))
           for t in steps_b]],
         xlabel='epoch', ylabel='learning rate / base rate',
```

```
        legend=['step decay (30 epochs)', 'cosine decay (45 epochs)'])
```



7.6.2.3 Mixup

Mixup (H. Zhang et al. 2018) regularizes a classifier by training on convex combinations of pairs of training examples. For each minibatch, it samples a mixing coefficient $\lambda \sim \text{Beta}(\alpha, \alpha)$ and pairs each example i with a randomly shuffled partner j, constructing

$$\begin{aligned}\tilde{\mathbf{x}} &= \lambda \mathbf{x}_i + (1-\lambda)\mathbf{x}_j,\\ \tilde{\mathbf{y}} &= \lambda \mathbf{y}_i + (1-\lambda)\mathbf{y}_j,\end{aligned} \tag{7.9}$$

where $\mathbf{y}_i$ and $\mathbf{y}_j$ are one-hot class vectors so $\tilde{\mathbf{y}}$ is a soft target distribution. Because cross-entropy is linear in its target distribution, the loss on $\tilde{\mathbf{y}}$ equals the λ-weighted sum of the cross-entropies against y_i and y_j, which is simpler and faster to compute. Encouraging predictions to interpolate linearly between training pairs discourages sharp memorization around individual points. With typical small values ($\alpha \in [0.1, 0.2]$), the Beta distribution concentrates near 0 and 1, leaving most minibatches lightly perturbed while strongly blending a few. Below we implement `mixup` using one shared λ per minibatch and save it to `d2l` for reuse in Section 7.7.

```
def mixup(X, y, alpha):
    """Mix a minibatch with a shuffled copy; return both label sets."""
    lam = random.betavariate(alpha, alpha)
    perm = torch.randperm(X.shape[0], device=X.device)
    return lam * X + (1 - lam) * X[perm], y, y[perm], lam
```

To visualize the effect clearly, we mix a minibatch of Fashion-MNIST garments with $\alpha = 2$, which concentrates λ near 0.5 (training uses a much smaller α). The top row shows the original images and the bottom row shows their superposed mixtures.

```
data = d2l.FashionMNIST(batch_size=8)
X, y = next(iter(data.train_dataloader()))
X_mix, y_a, y_b, lam = mixup(X, y, alpha=2.0)
d2l.show_images(torch.cat([X, X_mix]).squeeze(1), 2, 8)
print(f'lambda = {lam:.2f}')
```

```
lambda = 0.81
```

Because Mixup creates ghostly double exposures that never occur in real photographs, *CutMix* (Yun et al. 2019) blends spatially instead: it cuts a rectangular patch from one image, pastes it onto another, and mixes the labels in proportion to the patch's area, keeping every pixel locally realistic. Many modern vision recipes randomly alternate between Mixup and CutMix from minibatch to minibatch (see the exercises below).

7.6.2.4 Stochastic Depth

While dropout (Section 4.6) randomly zeroes out individual activations during training, *stochastic depth* (Huang et al. 2016) drops entire residual branches at random: with probability p, a block's residual branch is bypassed so the block passes its input straight through, effectively training an ensemble of shallower subnetworks. Residual connections make this safe because the identity shortcut always keeps the signal flowing. For a block computing $\mathbf{x} + f(\mathbf{x})$, stochastic depth replaces the branch with

$$\mathbf{x} + \frac{b}{1-p} f(\mathbf{x}), \qquad b \sim \text{Bernoulli}(1-p). \tag{7.10}$$

Just as in inverted dropout, scaling by $1/(1-p)$ keeps the expected branch output unchanged during training so no rescaling is needed at evaluation time. Modern implementations sample b independently for every example in the minibatch (`drop_path` below, saved to `d2l` for ConvNeXt). Subclassing `d2l.Residual` (Section 7.4), we drop the residual branch right before adding the shortcut; passing 1,000 random inputs through the block in training mode with $p = 0.5$, roughly half of all outputs should match $\text{ReLU}(\mathbf{x})$ identically.

```
def drop_path(Y, p, training):
    """Zero each example of Y with probability p, rescale the rest."""
    if not training or p == 0:
        return Y
    shape = (Y.shape[0],) + (1,) * (Y.ndim - 1)  # One draw per example
    keep = torch.rand(shape, device=Y.device) > p
    return Y * keep / (1 - p)
```

```
class StochasticResidual(d2l.Residual):
    """A residual block whose branch is dropped with probability p."""
    def __init__(self, num_channels, p):
        super().__init__(num_channels)
        self.p = p

    def forward(self, X):
        Y = self.bn2(self.conv2(F.relu(self.bn1(self.conv1(X)))))
        return F.relu(X + drop_path(Y, self.p, self.training))

blk, X = StochasticResidual(3, p=0.5), torch.randn(1000, 3, 8, 8)
Y = blk(X)  # A module is in training mode until .eval() is called
dropped = (Y == F.relu(X)).flatten(1).all(1).float().mean()
print(f'fraction of examples with a dropped branch: {float(dropped):.3f}')
```

```
fraction of examples with a dropped branch: 0.473
```

In deep architectures, the drop probability p typically ramps linearly from 0 in the earliest block to a peak of 0.1–0.5 in the deepest block. Notice that per-example masking still evaluates every branch before zeroing out dropped examples, so it regularizes without speeding up training; the original per-minibatch formulation draws a single b per block and can skip dropped branches completely to save compute.

7.6.2.5 Averaging Weights

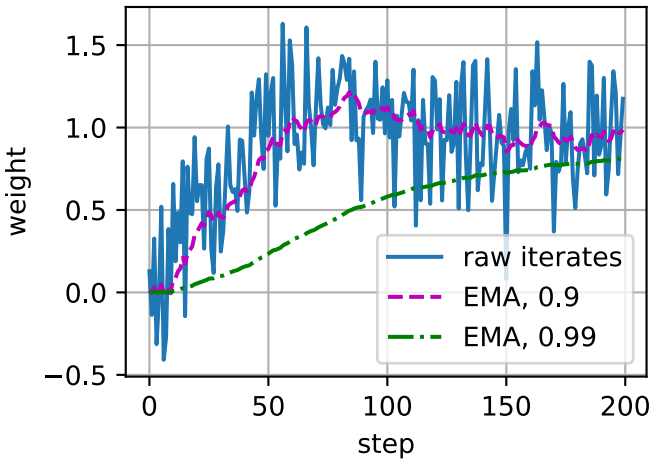
The final row of Table 7.4 delivers cleaner generalization at virtually zero extra cost. Near the end of training, minibatch noise causes parameters to bounce around the floor of a low-loss basin, so averaging recent iterates lands closer to the basin's center than any single checkpoint. Whereas *stochastic weight averaging* (SWA) (Izmailov et al. 2018) averages snapshots uniformly late in training, modern vision recipes commonly maintain an *exponential moving average* (EMA), the exact same recursion batch normalization uses for its running statistics (Section 7.3). After every optimizer step, a *shadow copy* $\bar{\theta}$ of the trainable parameters blends in the latest weights θ with decay ρ:

$$\bar{\theta} \leftarrow \rho\bar{\theta} + (1-\rho)\theta. \tag{7.11}$$

Parameters from k steps ago carry weight $(1-\rho)\rho^k$, giving an effective averaging horizon of roughly $1/(1-\rho)$ steps (100 steps for $\rho = 0.99$). Optimization continues updating θ as usual, while evaluation swaps in $\bar{\theta}$. (Because batch-normalization running statistics are buffers rather than trainable weights, an exercise below examines how best to pair them with EMA weights.) To see the tradeoff in ρ, let us track a single synthetic weight that climbs from 0 to 1 over 50 steps and then oscillates noisily around 1.

```
class EMA:
    """A shadow copy of a model's parameters, updated after each step."""
    def __init__(self, model, decay=0.99):
        self.decay = decay
        self.shadow = {k: p.detach().clone()
                       for k, p in model.named_parameters()}
```

```
    @torch.no_grad()
    def update(self, model):
        for k, p in model.named_parameters():
            self.shadow[k].lerp_(p, 1 - self.decay)  # beta*avg + (1-beta)*p

net = nn.Linear(1, 1, bias=False)
nn.init.zeros_(net.weight)  # Training starts at 0
emas = {decay: EMA(net, decay) for decay in (0.9, 0.99)}
raw, avg = [], {decay: [] for decay in emas}
for t in range(200):  # Iterates rise to 1 over 50 steps, then stay noisy
    with torch.no_grad():
        net.weight.copy_(min(t / 50, 1) + 0.3 * torch.randn(1, 1))
    raw.append(float(net.weight.detach()))
    for decay, ema in emas.items():
        ema.update(net)
        avg[decay].append(float(ema.shadow['weight']))
d2l.plot(list(range(200)), [raw, *avg.values()], xlabel='step',
         ylabel='weight', legend=['raw iterates', 'EMA, 0.9', 'EMA, 0.99'])
```



At $\rho = 0.9$ (horizon ~10 steps), the shadow weight trails the ramp by about 10 steps and then dramatically damps out the noise. At $\rho = 0.99$ (horizon ~100 steps), it is smoother still, yet lags so far behind that it still sits roughly 20% below 1 after 200 steps. That illustrates the main pitfall of EMA: whenever $1/(1-\rho)$ is large compared with the run length, the shadow copy lags behind improving weights and underperforms the final checkpoint. Very large decays like 0.999 or 0.9999 therefore belong only in long training runs.

7.6.3 Two Recipes on One Network

Now let us assemble these pieces and run a head-to-head test on a single fixed architecture: the ResNet-18 of Section 7.4, rebuilt from `d2l.Residual` with He initialization (Section 4.4). We train on a random subset of 10,000 Fashion-MNIST images resized to 96×96 (shared by both recipes within each run): using fewer training examples amplifies the role of regularization while keeping repeated experiments fast. Both recipes compute their step schedules directly from `lr` and `num_steps`.

```
class FashionMNIST10k(d2l.FashionMNIST):
    """Fashion-MNIST with a random subset of num_train training images."""
    def __init__(self, batch_size=128, resize=(96, 96), num_train=10000):
        super().__init__(batch_size, resize)
        idx = torch.randperm(len(self.train))[:num_train]
        self.train.data = self.train.data[idx]  # get_dataloader reads these
        self.train.targets = self.train.targets[idx]

class ResNet18(d2l.Classifier):
    """The ResNet-18 of the residual-networks section, from d2l.Residual."""
    def __init__(self, lr, num_steps, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        layers = [nn.LazyConv2d(64, kernel_size=7, stride=2, padding=3,
                                bias=False),
                  nn.LazyBatchNorm2d(), nn.ReLU(),
                  nn.MaxPool2d(kernel_size=3, stride=2, padding=1)]
        for i, c in enumerate((64, 128, 256, 512)):
            for j in range(2):
                down = i > 0 and j == 0
                layers.append(d2l.Residual(c, use_1x1conv=down,
                                           strides=2 if down else 1))
        layers += [nn.AdaptiveAvgPool2d((1, 1)), nn.Flatten(),
                   nn.LazyLinear(num_classes)]
        self.net = nn.Sequential(*layers)

    def weight_init(self, module):  # The Trainer applies it after a dry run
        if isinstance(module, (nn.Conv2d, nn.Linear)):
            nn.init.kaiming_normal_(module.weight)  # Variance 2/n_in
            if module.bias is not None:
                nn.init.zeros_(module.bias)
```

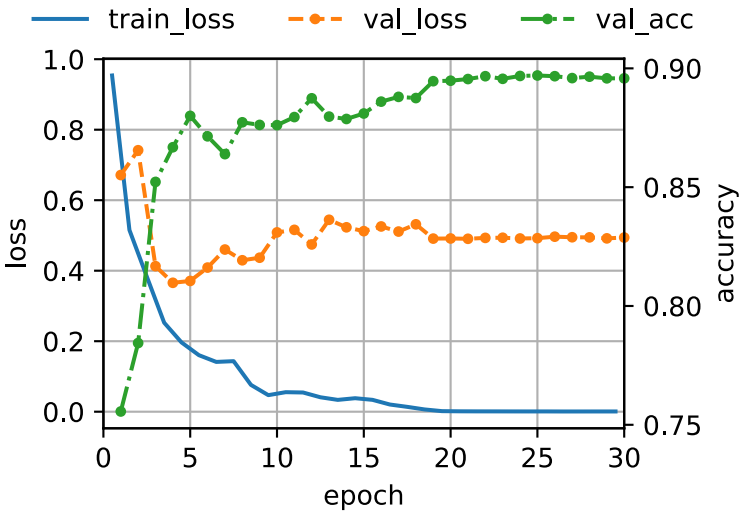
Both models share the exact same architecture and training split while changing the optimization and regularization recipe completely. *Recipe A* mirrors 2015 practice: SGD with momentum 0.9, coupled weight decay 5×10^{-4}, a peak learning rate of 0.05 with two epochs of linear warmup followed by 10-fold drops at 60% and 85% of training, plain cross-entropy loss, and 30 epochs. (The short two-epoch warmup is a minor modern stabilization, introduced for large-minibatch SGD by Goyal et al. (2017): without it, batch-normalized SGD at 0.05 diverged in epoch 1 in roughly a third of our runs.) Our *compact modern recipe* uses AdamW with weight decay 0.05, peak learning rate 0.002 with three epochs of warmup and cosine decay, label smoothing ($\epsilon = 0.1$), Mixup ($\alpha = 0.2$), and 45 epochs (reflecting the longer horizon modern regularization requires). To keep runtime manageable, it omits RandAugment, CutMix, stochastic depth, and EMA, testing a lean subset of the full menu.

Do not be misled by the 100-fold gap between the two weight-decay numbers (0.05 vs. 5×10^{-4}). The actual fractional shrinkage per step is learning rate times weight decay: at peak rate, that equals $0.002 \times 0.05 = 10^{-4}$ for AdamW versus $0.05 \times 5 \times 10^{-4} = 2.5 \times 10^{-5}$ for SGD—which momentum amplifies up to tenfold, making SGD's effective decay actually stronger. Finally, both recipes evaluate validation loss using plain cross-entropy on unmixed validation images; label smoothing and Mixup modify only the training objective.

```
class ClassicRecipe(ResNet18):
    """Recipe A: SGD with momentum, weight decay, warmup, step decay."""
    def configure_optimizers(self):
        optim = torch.optim.SGD(self.parameters(), lr=self.lr, momentum=0.9,
                                weight_decay=5e-4)
        warmup = 2 * self.num_steps // 30  # Two of the 30 epochs
        return optim, torch.optim.lr_scheduler.LambdaLR(
            optim, lambda t: step_decay(t, self.num_steps, warmup))

class ModernRecipe(ResNet18):
    """Compact modern recipe: AdamW, warmup and cosine, smoothing, Mixup."""
    def configure_optimizers(self):
        warmup = 3 * self.num_steps // 45  # Three of the 45 epochs
        optim = torch.optim.AdamW(self.parameters(), lr=self.lr,
                                  weight_decay=0.05)
        return optim, torch.optim.lr_scheduler.LambdaLR(
            optim, lambda t: d2l.cosine_warmup(t, self.num_steps, warmup))

    def training_step(self, batch):
        X, y_a, y_b, lam = mixup(*batch, alpha=0.2)
        Y_hat = self(X)
        smooth = lambda y: F.cross_entropy(Y_hat, y, label_smoothing=0.1)
        return lam * smooth(y_a) + (1 - lam) * smooth(y_b)
```

First we train Recipe A for 30 epochs and evaluate on the full 10,000-image validation split.

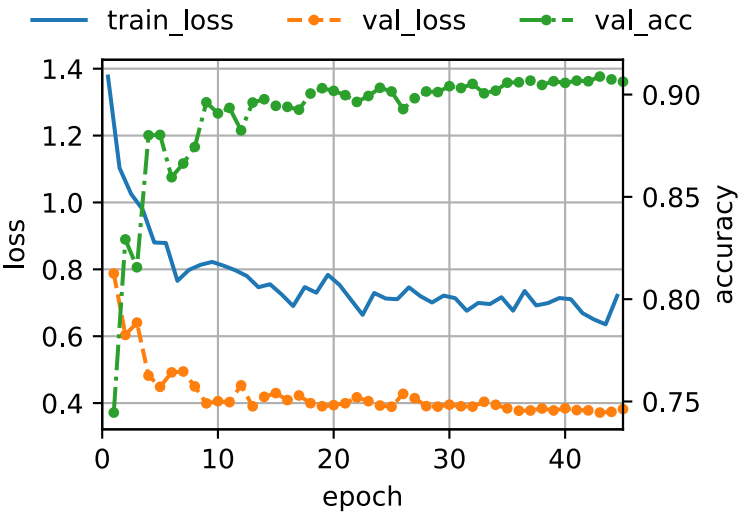
```
data = FashionMNIST10k()
steps_per_epoch = len(data.train_dataloader())  # 79 minibatches of <= 128
model = ClassicRecipe(lr=0.05, num_steps=30 * steps_per_epoch)
trainer = d2l.Trainer(max_epochs=30, num_gpus=1)
trainer.fit(model, data)
trainer.history['val_acc'][-1]
```



```
0.8958
```

Next we train the exact same architecture on the exact same training subset for 45 epochs using the compact modern recipe.

```
model = ModernRecipe(lr=2e-3, num_steps=45 * steps_per_epoch)
trainer = d2l.Trainer(max_epochs=45, num_gpus=1)
trainer.fit(model, data)
trainer.history['val_acc'][-1]
```




```
0.9063
```

Table 7.5 reports the validation accuracy range across repeated runs (each drawing a fresh 10,000-image subset, initialization, and minibatch order).

:The same ResNet-18 under both recipes (10,000 Fashion-MNIST training images at 96×96; validation accuracy, range over repeated runs).

Table 7.5: Recipe Results

Recipe	Epochs	Validation accuracy
A (2015): SGD, warmup, step decay	30	89–90.5%
Compact modern: AdamW, cosine, smoothing, Mixup	45	90.5–91%

In every paired run on the same 10,000-image subset, `ModernRecipe` beats `ClassicRecipe` by 0.5–1.5 percentage points (even while omitting RandAugment, CutMix, stochastic depth, and EMA). Three practical lessons accompany this comparison:

1. **Bundled comparisons vs. ablations:** Because optimizer, schedule, loss, augmentation, and epoch budget changed together, only a one-at-a-time ablation (Exercise 4) can isolate which ingredient contributed most—and because optimal learning rates differ $25\times$ between AdamW (0.002) and SGD (0.05), any optimizer swap must retune the learning rate.

2. **Never compare raw training losses across recipes:** Under `ClassicRecipe`, training loss plunges toward zero while validation loss rebounds to 0.4–0.5 as logits grow overconfident; under `ModernRecipe`, validation loss drops below 0.4 while training loss stays *higher* than validation loss because Mixup and label smoothing evaluate a soft, blended training objective.
3. **Reading published leaderboards:** Because training recipes alone move a fixed ResNet-50 by > 4 ImageNet points (Wightman et al. 2021), a 1-point claim over a baseline copied from an older paper confounds recipe with architecture. Even under matched recipes, test-set noise and multi-object ambiguity on ImageNet (Recht et al. 2019; Beyer et al. 2020) mean that transfer to detection or segmentation is often a sharper test of backbone quality than sub-point ImageNet top-1 deltas.

7.6.4 Summary

Between 2015 and the early 2020s, convnet training recipes evolved across every axis: adaptive optimizers with decoupled weight decay (AdamW, LAMB), cosine warmup schedules, RandAugment/Mixup/CutMix, label smoothing, stochastic depth, longer training horizons, and EMA weight averaging. One such recipe (LAMB, a cosine schedule, RandAugment with Mixup and CutMix, stochastic depth, 600 epochs, and no weight averaging) lifted an unchanged ResNet-50 from 76.1% to 80.4% on ImageNet. Each ingredient takes only a few lines of code, and a compact subset reliably improves ResNet-18 on `FashionMNIST10k`. Next, Section 7.7 holds the modern recipe fixed and modernizes the ResNet architecture itself one step at a time.

7.6.5 Exercises

1. **Binary cross-entropy and Mixup.** Wightman et al. (2021) train ImageNet classifiers with binary cross-entropy (treating each of the K classes as an independent sigmoid binary classification task) rather than K-way softmax cross-entropy. Explain why independent sigmoid outputs pair naturally with Mixup and CutMix when a blended image contains objects from two distinct classes at once.

2. code **CutMix.** CutMix (Yun et al. 2019) samples $\lambda \sim \text{Beta}(1, 1) = \text{Uniform}(0, 1)$ and pastes a random rectangular patch from a shuffled partner image covering a target area fraction $1 - \lambda$. Implement `cutmix` as a drop-in variant of `mixup`, mixing the one-hot labels using the *exact* fraction of pixels kept after clipping the rectangle to the image borders. Train `ModernRecipe` on `FashionMNIST10k` once with CutMix in place of Mixup and once alternating randomly between Mixup and CutMix each minibatch, and compare validation accuracy against Table 7.5.

3. code **Weight averaging horizon (EMA).** Integrate the `EMA` class into `ModernRecipe` by updating the shadow weights after every optimizer step and evaluating validation accuracy with the shadow parameters. Sweep the decay $\rho \in \{0.9, 0.99, 0.999\}$ in Equation 7.11. Relate the best-performing ρ to the effective horizon $1/(1-\rho)$ and the total number of steps ($45 \times 79 = 3{,}555$), and explain how batch-normalization running statistics should be handled when evaluating EMA weights.

4. code **One-at-a-time recipe ablation.** ● In Table 7.5, `ModernRecipe` beats `ClassicRecipe` by 0.5–1.5 percentage points when all five ingredients change at once.

    1. Run both `ClassicRecipe` and `ModernRecipe` five times on `FashionMNIST10k` to estimate the run-to-run standard deviation of validation accuracy.

2. Starting from `ModernRecipe`, revert one ingredient at a time to its `ClassicRecipe` counterpart: (i) remove Mixup, (ii) remove label smoothing, (iii) replace cosine warmup with `step_decay`, (iv) replace AdamW with momentum SGD (retuning the peak learning rate to 0.05), and (v) shorten training from 45 to 30 epochs. Which individual changes exceed the run-to-run noise floor?
3. Complete the 2×2 table of optimizer (SGD vs. AdamW) $\times$ schedule (`step_decay` vs. `cosine_warmup`) with Mixup and label smoothing enabled. Do the optimizer and schedule interact?

5. code **Stochastic depth in the recipe.** ● Our compact modern recipe omitted stochastic depth. Add it back to `ResNet18`:
   1. Extend `StochasticResidual` to support stride-2 downsampling blocks with 1×1 shortcuts, and build `ResNet18` with drop probabilities growing linearly from 0 in block 1 to p_{max} in block 8. What is the expected number of active residual branches per forward pass as a function of p_{max}?
   2. Sweep $p_{\text{max}} \in \{0.1, 0.2, 0.5\}$ under `ModernRecipe` on `FashionMNIST10k` (compared against $p_{\text{max}} = 0$). Why does a shallow 8-block network benefit less from large p_{max} than a 50- or 100-block network?
   3. Implement the per-minibatch variant of Huang et al. (2016), which draws a single Bernoulli variable b per block and skips evaluating the residual branch altogether when $b = 0$. At $p_{\text{max}} = 0.5$, compare the theoretical fraction of skipped branches against the measured reduction in wall-clock training time.

7.7 ConvNeXt: Modernizing a ResNet Step by Step

Every architecture in this chapter so far has been convolutional, and until 2020 convnets sat atop every major image-classification leaderboard. Then architectures built around *attention*, without a single overlapping spatial convolution, took the lead. The first vision transformer did so in late 2020 with the help of a pre-training set far larger than ImageNet. In 2021 the *Swin Transformer* (Liu et al. 2021) needed no such help: trained on ImageNet alone, it achieved 81.3% top-1 accuracy, compared with 76.1% for a similarly sized ResNet-50. Did attention itself cause that five-point leap—and if so, were convolutional networks obsolete?

A brief sketch of vision transformers suffices here (we develop them fully in Chapter 11). A *vision transformer* (Dosovitskiy et al. 2021) partitions an image into a grid of patches, embeds each patch as a feature vector, and processes the sequence through a stack of identical blocks. Each block has two sublayers: an *attention* layer that lets every patch compute a data-dependent weighted average of other patches, followed by a pointwise MLP applied independently to each patch vector. Swin restores hierarchical convnet structure by computing attention inside local 7×7 windows and downsampling between stages.

After Section 7.6, we should immediately be skeptical of crediting a five-point gap to attention alone. Swin differs from a 2015 ResNet in at least a dozen ways beyond attention: it trains with a 300-epoch modern recipe, slices images with a patch stem, allocates depth differently across stages, and adopts different activations and normalizations. To disentangle these factors, Z. Liu et al. (2022) ran a landmark controlled experiment: starting from a standard ResNet-50, they introduced Swin-inspired modernizations one at a time, measuring ImageNet top-1 accuracy after every change without ever adding attention. The resulting pure convnet, *ConvNeXt*, reaches 82.0%—edging past Swin-T using primitives we have already built in this chapter: modern training recipes (Section 7.6), depthwise convolutions and in-

verted bottlenecks (Section 7.5), and layer normalization (Section 7.3). Because each row of their ablation requires a full ImageNet run, we walk through their published roadmap step by step, implement ConvNeXt from scratch, and then conduct a compact three-step modernization roadmap of our own on Fashion-MNIST starting from a parameter-matched ResNet-18.

7.7.1 The Modernization Roadmap

Table 7.6 walks through the exact sequence of changes tested by Z. Liu et al. (2022). Each row accumulates every modification above it while keeping compute anchored near Swin-T's budget of roughly 4.5 billion multiply-adds per image (reported as 4.5 GFLOPs in the paper).

Table 7.6: The ConvNeXt modernization roadmap (Z. Liu et al. 2022). ImageNet top-1 accuracy of a ResNet-50 as design changes accumulate; each row includes all rows above it, at roughly constant cost. Swin-T, a transformer of similar cost, reaches 81.3%.

Change	What it means	Top-1
(starting point)	ResNet-50 with its 2015 recipe	76.1%
Modern training recipe	300 epochs, AdamW, Mixup, label smoothing, stochastic depth (Section 7.6)	78.8%
Stage ratio	blocks per stage $(3, 4, 6, 3) \rightarrow (3, 3, 9, 3)$	79.4%
Patchify stem	7×7 stride-2 convolution plus pooling $\rightarrow$ one 4×4 stride-4 convolution	79.5%
Depthwise convolutions	the 3×3 convolution becomes depthwise; width grows from 64 to 96 channels	80.5%
Inverted bottleneck	expand $4\times$ inside the block instead of compressing	80.6%
Large kernels	depthwise convolution moved to the front and enlarged to 7×7	80.6%
ReLU $\rightarrow$ GELU	a smooth activation, as in transformers	80.6%
Fewer activations	one activation per block instead of three	81.3%
Fewer normalizations	one normalization per block instead of three	81.4%
BN $\rightarrow$ LN	layer normalization over channels at each position	81.5%
Separate downsampling	an LN plus 2×2 stride-2 convolution between stages	82.0%

Immediately, the very first row delivers the largest single jump—without touching a single layer of the network. Echoing Section 7.6, upgrading the training recipe alone lifts an untouched ResNet-50 by 2.7 percentage points. Of the 5.9-point total gain separating 2015 ResNet-50 from ConvNeXt, 2.7 points come from training and 3.2 from architecture (in this ordering), confirming that pitting a modern transformer against a 2015-recipe ResNet confounded recipes with architectures.

7.7.1.1 Macro Design

The next two rows reshape the network's macro structure. Swin-T allocates its blocks across four stages in a $1{:}1{:}3{:}1$ ratio, concentrating depth in stage 3 where feature maps are compact and compute is cheap; shifting ResNet-50's $(3, 4, 6, 3)$ layout to $(3, 3, 9, 3)$ adds 0.6 points. Next comes the stem. Whereas ResNet opens with an overlapping 7×7 stride-2 convolution followed by max-pooling to downsample fourfold, a vision transformer chops the image into disjoint patches and projects each patch through a shared linear layer—which is algebraically identical to a non-overlapping 4×4 convolution with stride 4 (Section 6.3). Swapping in this *patchify* stem preserves accuracy (79.5%) while producing one token vector per 4×4 patch, just like a transformer.

7.7.1.2 Depthwise Convolutions and the Inverted Bottleneck

A transformer block cleanly factors computation into two stages: attention mixes information *across spatial positions*, and the MLP mixes information *across channels* at each position. As we saw in Section 7.5.1.1, a convnet can perform the exact same factorization: a *depthwise* convolution mixes across space within each channel, and 1×1 convolutions mix across channels at each pixel. Replacing the dense 3×3 layer in ResNet's bottleneck with a depthwise convolution slashes compute enough to widen stage 1 from 64 to 96 channels (matching Swin-T), jumping accuracy to 80.5%.

Once the spatial convolution is depthwise rather than dense, its cost scales linearly rather than quadratically with width, removing the rationale for compressing channels before filtering. ConvNeXt therefore adopts MobileNetV2's *inverted bottleneck* (Section 7.5), maintaining a thin residual stream and expanding $4\times$ inside the block (the same $1\times \to 4\times \to 1\times$ geometry as a transformer MLP), nudging accuracy to 80.6%.

With spatial filtering now lightweight, kernel size also becomes easy to enlarge. First, moving the depthwise convolution to the front of the block so it acts on c channels instead of $4c$ drops accuracy by 0.7 points (to 79.9%—the sole dip on the roadmap); expanding its kernel from 3×3 to 7×7 immediately recovers that ground. A 7×7 filter matches the receptive field of three stacked 3×3 layers (Section 6.3). If both cover the same window, why should the single large kernel help? The stack covers its window unevenly: far more paths lead from a central pixel to the output than from a pixel at the border, so the *effective* receptive field of a stack is roughly Gaussian and much smaller than the nominal one (Luo et al. 2016). A single 7×7 kernel can give every position in the window whatever weight it learns, in one layer. Back in VGG (Section 7.2.1), dense 7×7 kernels were prohibitively expensive ($k^2 c_{\text{i}} c_{\text{o}}$); for a depthwise kernel ($k^2 c$), growing from 3×3 to 7×7 only scales a tiny fraction of block compute by $49/9 \approx 5$. Accuracy saturates right at 7×7, matching the exact window size of Swin's local attention.

7.7.1.3 Micro Design

The final group of edits refines individual layers inside the block. Swapping ReLU for *GELU* (Hendrycks and Gimpel 2016), the smooth activation favored by transformers, leaves accuracy unchanged on its own (80.6%). What *does* move the needle is trimming the *number* of activations and normalizations. A classic ResNet bottleneck puts a normalization and an activation after all three convolutions, whereas a transformer block uses only a single activation inside its MLP and fewer normalizations. Removing two of the three GELUs, keeping only the one between the two 1×1 layers, jumps accuracy by 0.7 points, the biggest

micro-design gain on the table. (There is still no crisp theoretical proof for why removing two nonlinearities improves accuracy; it is best viewed as a clean empirical finding.) Trimming three normalizations down to one adds 0.1 points, and switching that lone normalization from batch normalization to per-position *layer normalization* across channels (Section 7.3.4.1) adds another 0.1 points (81.5%). Figure 7.20 compares the original ResNet bottleneck against the resulting ConvNeXt block (including the learnable per-channel layer scale γ described below).

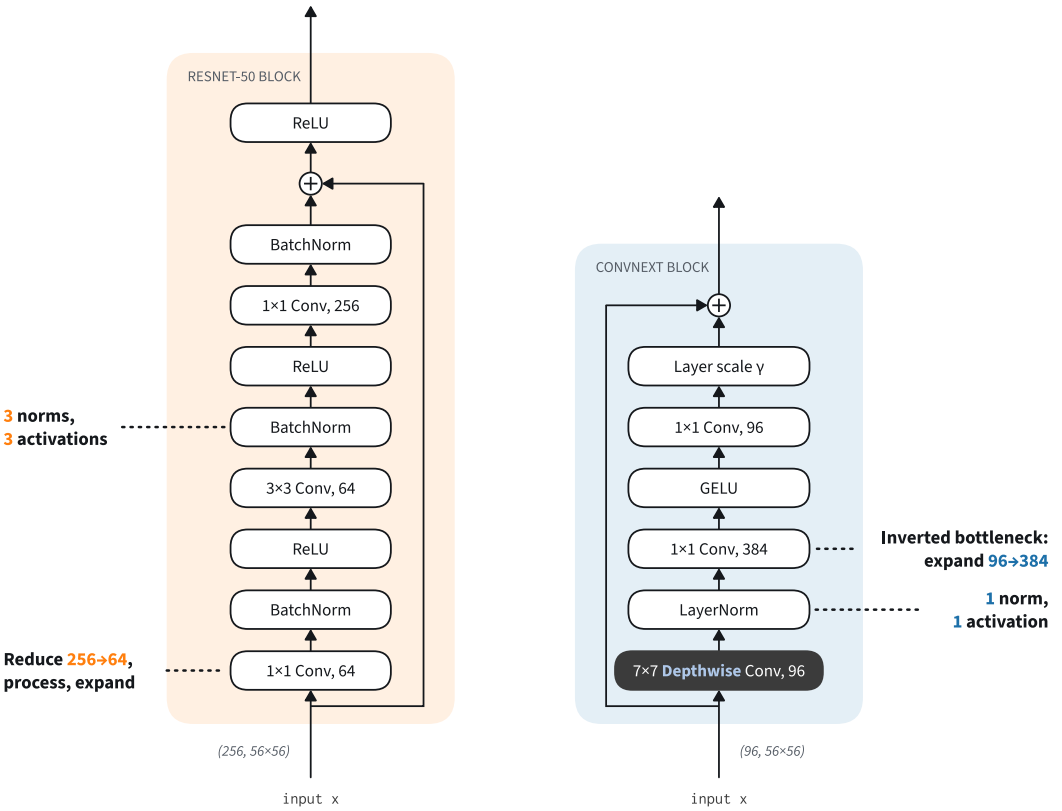


Figure 7.20: The ResNet-50 bottleneck block and the ConvNeXt block that the roadmap turns it into. Three normalizations and three activations become one of each; the compress-process-expand bottleneck becomes an inverted bottleneck led by a 7×7 depthwise convolution, and a learnable per-channel layer scale γ multiplies the branch before the addition.

Finally, whereas ResNet downsamples inside the opening block of each stage with a stride-2 convolution, Swin inserts dedicated patch-merging layers between stages. Adopting separate downsampling layers (a layer normalization followed by a 2×2 stride-2 convolution between stages, plus normalizations after the stem and before the linear head to stabilize training) completes the roadmap at 82.0%. Figure 7.21 shows the assembled ConvNeXt-T backbone, stacking $(3, 3, 9, 3)$ blocks of widths $(96, 192, 384, 768)$ in the classic stem–body–head layout (Section 7.2.3).

At matched compute, ConvNeXt-T surpasses Swin-T without a single attention layer. The takeaway is not that convolutions are inherently superior to attention, but that the 2021 accuracy gap came from training recipes and architectural design hygiene rather than at-

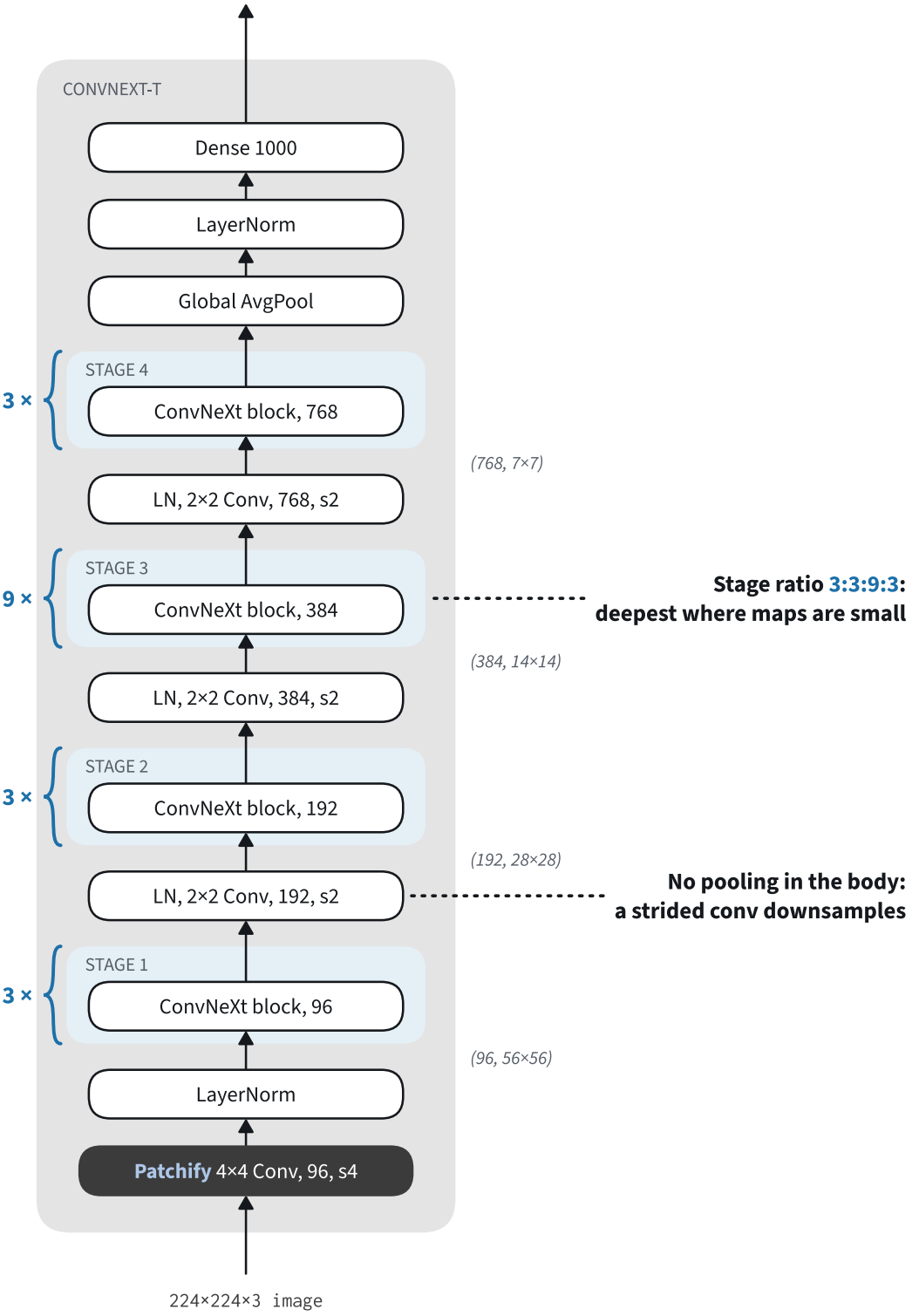


Figure 7.21: ConvNeXt-T. A patchify stem replaces convolution-plus-pooling; stages of 3, 3, 9, and 3 blocks are joined by separate LN plus 2×2 stride-2 downsampling layers; there is no pooling anywhere in the body.

tention alone—under matched tuning, both paradigms scale comparably.

7.7.2 Implementation

Because the modernization roadmap stripped away more layers than it added, the ConvNeXt block is remarkably clean to implement.

7.7.2.1 The ConvNeXt Block

Each block applies a 7×7 depthwise convolution, per-position layer normalization, a 1×1 expansion ($c \to 4c$), GELU, and a 1×1 projection ($4c \to c$), then adds the residual input. Two training stabilizers from the paper accompany the block. *Layer scale* (Touvron, Cord, Sablayrolles, et al. 2021) multiplies the residual branch by a learnable per-channel vector γ initialized to 10^{-6} (distinct from batch-norm scale), ensuring every block starts nearly as the identity so deep optimization begins smoothly. It does for a layer-normalized block what starting the last batch-norm scale of a residual branch at zero does for a ResNet (Section 7.4.2). *Stochastic depth* randomly drops entire residual branches during training via `d2l.drop_path` (Section 7.6).

Because layer normalization operates across the channel vector at each spatial location, in a channels-last tensor `(batch, height, width, channels)` it is standard `LayerNorm` over the final axis, and the two 1×1 convolutions are ordinary linear layers over that same axis. Following Z. Liu et al. (2022), we initialize all convolutional and linear weights from $\mathcal{N}(0, 0.02^2)$ with zero biases.

Images are stored channels-first, so `LayerNorm2d` permutes to channels-last for the normalization and back, and the block does the same around its two linear layers.

```
class LayerNorm2d(nn.LayerNorm):
    """LayerNorm over the channel axis of an NCHW tensor."""
    def __init__(self, num_channels, eps=1e-6):
        super().__init__(num_channels, eps=eps)

    def forward(self, X):
        return super().forward(X.permute(0, 2, 3, 1)).permute(0, 3, 1, 2)

class ConvNeXtBlock(nn.Module):
    """Depthwise 7x7, norm, 1x1 expand, GELU, 1x1 project, scaled residual."""
    def __init__(self, dim, drop_prob=0.0, layer_scale=1e-6,
                 norm=LayerNorm2d):
        super().__init__()
        self.dwconv = nn.Conv2d(dim, dim, kernel_size=7, padding=3,
                                groups=dim)
        self.norm = norm(dim)
        self.pwconv1 = nn.Linear(dim, 4 * dim)  # 1x1 conv in channels-last
        self.pwconv2 = nn.Linear(4 * dim, dim)
        self.gamma = nn.Parameter(layer_scale * torch.ones(dim))
        self.drop_prob = drop_prob

    def forward(self, X):
        Y = self.norm(self.dwconv(X)).permute(0, 2, 3, 1)  # to channels-last
        Y = self.pwconv2(F.gelu(self.pwconv1(Y)))
```

```
        Y = (self.gamma * Y).permute(0, 3, 1, 2)  # back to channels-first
        return X + d2l.drop_path(Y, self.drop_prob, self.training)
```

7.7.2.2 The Full Network

As in VGG (Section 7.2.1), the complete architecture iterates over a tuple of `(depth, channels)` stage specifications, opening with the patchify stem plus layer normalization and inserting dedicated downsampling layers between stages with no spatial pooling anywhere in the body. Stochastic-depth drop probabilities ramp linearly from 0 in the first block to `drop_path_max` in the last. For Fashion-MNIST we adopt the compact *ConvNeXt-atto* profile, with widths $(40, 80, 160, 320)$ and stage depths $(2, 2, 6, 2)$, keeping the third stage deepest just like ConvNeXt-T. Exposing `norm` as an argument and `stem` as a method makes swapping them trivial in our ablation below.

```
class ConvNeXt(d2l.Classifier):
    def __init__(self, arch=((2, 40), (2, 80), (6, 160), (2, 320)),
                 lr=2e-3, num_classes=10, drop_path_max=0.1,
                 norm=LayerNorm2d):
        super().__init__()
        self.save_hyperparameters()
        depths = [d for d, c in arch]
        rates = torch.linspace(0, drop_path_max, sum(depths)).tolist()
        layers = self.stem(arch[0][1], norm)
        c_prev, b = arch[0][1], 0
        for i, (depth, c) in enumerate(arch):
            if i > 0:  # Separate downsampling layer between stages
                layers += [norm(c_prev),
                           nn.Conv2d(c_prev, c, kernel_size=2, stride=2)]
            for _ in range(depth):
                layers.append(ConvNeXtBlock(c, rates[b], norm=norm))
                b += 1
            c_prev = c
        layers += [nn.AdaptiveAvgPool2d((1, 1)), norm(c_prev), nn.Flatten(),
                   nn.Linear(c_prev, num_classes)]
        self.net = nn.Sequential(*layers)
        self.net.apply(self.weight_init)

    def stem(self, c, norm):  # Patchify: a 4x4 convolution with stride 4
        return [nn.Conv2d(1, c, kernel_size=4, stride=4), norm(c)]

    def weight_init(self, module):
        if type(module) in (nn.Conv2d, nn.Linear):
            nn.init.normal_(module.weight, std=0.02)
            nn.init.zeros_(module.bias)
```

On 64×64 Fashion-MNIST inputs, the patchify stem produces a 16×16 grid, and the three downsampling layers step resolution down to 8×8, 4×4, and 2×2 before global average pooling. Let us check the output shape and count trainable parameters.

```
model = ConvNeXt()
assert model(torch.randn(1, 1, 64, 64)).shape == (1, 10)
d2l.num_params(model)
```

```
3376450
```

Our ConvNeXt-atto model contains exactly 3,376,450 parameters—roughly a third of ResNet-18's 11.2 million (Section 7.4). Because any misconfigured width or kernel size immediately shifts this exact number, matching the parameter count is a reliable sanity check for any reimplementation.

7.7.3 A Small Roadmap on Fashion-MNIST

Every choice in Table 7.6 was validated on 1.3 million 224×224 ImageNet photos. Do those same architectural decisions carry over to small 64×64 grayscale garments? To find out, we run a compact cumulative roadmap of our own: starting from a parameter-matched control, a ResNet-18 narrowed from base width 64 to 35 channels (3.35M parameters), we apply ConvNeXt's modifications in three cumulative steps. Every model trains for 20 epochs on full Fashion-MNIST under the compact modern recipe from Section 7.6 (AdamW with decoupled weight decay, 10% linear warmup plus cosine decay, label smoothing, and Mixup; Chapter 9 covers AdamW in detail). To isolate architecture cleanly, we set ConvNeXt's stochastic-depth rate to 0 so both families share identical regularization.

Packaging the training procedure as a `ModernRecipe` mixin lets every model in the roadmap inherit the exact same optimizer, schedule, and mixed training step while keeping validation loss as plain cross-entropy.

```
class ModernRecipe:
    """AdamW, linear warmup then cosine decay, label smoothing, Mixup."""
    def configure_optimizers(self):
        optim = torch.optim.AdamW(self.parameters(), lr=self.lr,
                                  weight_decay=0.05)
        warmup = self.num_steps // 10
        schedule = torch.optim.lr_scheduler.LambdaLR(
            optim, lambda t: d2l.cosine_warmup(t, self.num_steps, warmup))
        return optim, schedule  # The Trainer steps it after every minibatch

    def training_step(self, batch):
        X, y_a, y_b, lam = d2l.mixup(*batch, alpha=0.2)
        Y_hat = self(X)
        loss = lambda y: F.cross_entropy(Y_hat, y, label_smoothing=0.1)
        return lam * loss(y_a) + (1 - lam) * loss(y_b)

class ModernConvNeXt(ModernRecipe, ConvNeXt):
    def __init__(self, num_steps, lr=2e-3, norm=LayerNorm2d):
        super().__init__(lr=lr, drop_path_max=0.0, norm=norm)
        self.num_steps = num_steps
```

Our baseline `CompactResNet18` stacks `d2l.Residual` blocks (Section 7.4) across widths 35, 70, 140, and 280 behind ResNet's standard 7×7 stride-2 stem and max-pooling.

```
class CompactResNet18(ModernRecipe, d2l.Classifier):
    """ResNet-18 with base width 35 instead of 64, trained by the recipe."""
    def __init__(self, num_steps, lr=2e-3, num_classes=10, base=35):
        super().__init__()
        self.save_hyperparameters()
        layers = [nn.LazyConv2d(base, kernel_size=7, stride=2, padding=3,
                                bias=False),
                  nn.LazyBatchNorm2d(), nn.ReLU(),
                  nn.MaxPool2d(kernel_size=3, stride=2, padding=1)]
        for i, c in enumerate((base, 2 * base, 4 * base, 8 * base)):
            for j in range(2):
                down = i > 0 and j == 0
                layers.append(d2l.Residual(c, use_1x1conv=down,
                                           strides=2 if down else 1))
        layers += [nn.AdaptiveAvgPool2d((1, 1)), nn.Flatten(),
                   nn.LazyLinear(num_classes)]
        self.net = nn.Sequential(*layers)

    def weight_init(self, module):
        # The Trainer calls this after the first minibatch fixed the shapes
        if type(module) in (nn.Conv2d, nn.Linear):
            nn.init.kaiming_normal_(module.weight, nonlinearity='relu')
            if module.bias is not None:
                nn.init.zeros_(module.bias)
```

We bundle the paper's changes into three clean steps by architectural region: (1) replace ResNet's blocks and stages with ConvNeXt's 7×7 inverted bottlenecks, $(2, 2, 6, 2)$ stage ratio, single GELU/norm, layer scale, and separate downsampling while keeping ResNet's stem and batch normalization (covering eight rows of Table 7.6); (2) swap batch normalization for layer normalization; and (3) replace ResNet's stem with the 4×4 patchify stem to arrive at full ConvNeXt. (Although the original paper swapped the stem earlier, the final architecture is order-independent, even if individual step attributions can vary slightly.) Overriding `stem` in `ResNetStemConvNeXt` and toggling `norm` implements all four models cleanly.

```
class ResNetStemConvNeXt(ModernConvNeXt):
    """ConvNeXt with the stem of ResNet-18 instead of the patchify stem."""
    def stem(self, c, norm):  # 7x7 convolution with stride 2, max-pooling
        return [nn.Conv2d(1, c, kernel_size=7, stride=2, padding=3), norm(c),
                nn.ReLU(), nn.MaxPool2d(kernel_size=3, stride=2, padding=1)]

roadmap = {'compact ResNet-18': CompactResNet18,
           'ConvNeXt blocks and stages, BN': (
               lambda num_steps: ResNetStemConvNeXt(num_steps,
                                                    norm=nn.BatchNorm2d)),
           '+ BN -> LN': ResNetStemConvNeXt,
           '+ patchify stem = ConvNeXt': ModernConvNeXt}
```

Let us train all four models in sequence and record their parameter counts and final validation accuracies.

```
data = d2l.FashionMNIST(batch_size=128, resize=(64, 64))
num_steps = 20 * len(data.train_dataloader())
results = {}
for name, make_model in roadmap.items():
    model = make_model(num_steps)
    trainer = d2l.Trainer(max_epochs=20, num_gpus=1)
    trainer.board.display = False  # Four runs; we tabulate the results
    trainer.fit(model, data)
    results[name] = (d2l.num_params(model), trainer.history['val_acc'][-1])
```

We now iterate through `results` to display the parameter count and 20-epoch validation accuracy for each step along the `roadmap`.

```
for name, (params, accuracy) in results.items():
    print(f'{name:34s} {params:>9,} parameters, '
          f'validation accuracy {accuracy:.3f}')
```

```
compact ResNet-18                  3,346,675 parameters, validation accuracy 0.936
ConvNeXt blocks and stages, BN     3,377,770 parameters, validation accuracy 0.937
+ BN -> LN                         3,377,770 parameters, validation accuracy 0.937
+ patchify stem = ConvNeXt         3,376,450 parameters, validation accuracy 0.918
```

Table 7.7: Our roadmap on Fashion-MNIST beside the paper's. The last column is the validation accuracy after 20 epochs of the modern recipe at 64×64, rounded to the stability of repeated runs; the paper's column adds up the ImageNet top-1 gains of the rows of Table 7.6 that each step covers.

Step	Rows of the paper	Gain on ImageNet	Parameters	Validation accuracy
Compact ResNet-18	(ResNet-50 with the modern recipe)		3.35M	≈ 94%
ConvNeXt blocks and stages, BN	stage ratio to fewer normalizations, separate downsampling	+3.0	3.38M	≈ 94%
BN → LN	BN → LN	+0.1	3.38M	≈ 94%
Patchify stem (our ConvNeXt)	patchify stem	+0.1	3.38M	≈ 92%

The results in Table 7.7 pinpoint what transfers to 64×64 Fashion-MNIST and what does not. Keeping ResNet's overlapping stem, ConvNeXt's modernized blocks, stage ratio, and layer normalization tie with our parameter-matched `CompactResNet18` (~94%, within ~43–46M multiply-adds per image). The entire ~2-point drop to ~92% comes from the single step that was neutral on 224×224 ImageNet: the non-overlapping 4×4 stride-4 patchify stem, which jumps straight to a coarse 16×16 grid in one linear projection. Architectural choices tuned at ImageNet resolution do not transfer blindly to low-resolution domains, and a cumulative ablation immediately identifies which component to adapt.

7.7.4 Limits and Open Problems

While ConvNeXt showed that pure convnets match hierarchical Vision Transformers in supervised classification under matched recipes, two active frontiers remain:

1. **Masked self-supervised pretraining:** Dropping random image patches as in masked autoencoders (He et al. 2022) is natural for a sequence of patch tokens, whereas dense sliding convolutions bleed across masked holes. *ConvNeXt V2* (Woo et al. 2023) bridges this gap with sparse convolutions and Global Response Normalization (Exercise 4).
2. **Global context:** Even a 7×7 depthwise filter is local, leaving open how ultra-large depthwise kernels (Ding et al. 2022), deformable convolutions, and hybrid conv–attention stacks compare at frontier scale.

7.7.5 Summary

Starting from a standard ResNet-50, ConvNeXt applied Swin-inspired upgrades one at a time: a 300-epoch modern recipe (+2.7 points), a $1{:}1{:}3{:}1$ stage ratio, a 4×4 stride-4 patchify stem, a 7×7 depthwise convolution leading a $4\times$ inverted bottleneck, a single GELU and LayerNorm per block, layer scale, and separate 2×2 stride-2 downsampling layers. Together these changes reach 82.0% on ImageNet without attention. On 64×64 Fashion-MNIST, ConvNeXt's body and LayerNorm match a parameter-matched ResNet-18 (~94%), while the coarse 4×4 patchify stem costs ~2 points.

7.7.6 Exercises

1. **Arithmetic of the depthwise move and kernel enlargement.** In Table 7.6, just before the "Large kernels" row, each block expands $c \to 4c$ with a 1×1 convolution, filters the $4c$ channels with a 3×3 depthwise convolution, and projects $4c \to c$ with a second 1×1 convolution. The "Large kernels" step first moves the depthwise convolution to the *front* of the block (where it acts on c channels instead of $4c$) and then enlarges its kernel to 7×7.
    1. Count the multiply-adds per spatial pixel of the depthwise layer before the move (3×3 on $4c$), after the move (3×3 on c), and after enlarging to 7×7 on c. Compare the net change against what enlarging to 7×7 *in place* on $4c$ channels would have cost.
    2. What fraction of the full `ConvNeXtBlock`'s multiply-adds per pixel ($49c + 8c^2$) does the 7×7 depthwise convolution account for at $c = 96$ and at $c = 384$?
2. **Where the parameters live in ConvNeXt.**
    1. For the `ConvNeXt` model of this section ($3{,}376{,}450$ parameters), compute the exact fraction of parameters in (i) the 7×7 depthwise convolutions, (ii) the 1×1 expansion and projection layers, (iii) the normalization and layer-scale vectors inside the blocks, (iv) the downsampling layers, and (v) the stem and head.
    2. Express the parameter count of a `ConvNeXtBlock` and a two-layer `Residual` block (Section 7.4, without a 1×1 shortcut) as functions of width c. By what asymptotic factor is `ConvNeXtBlock` more parameter-efficient for large c, and which design choice accounts for that factor?
3. code **Layer scale versus stochastic depth.** Under the roadmap recipe of Table 7.7, train `ConvNeXt` with layer-scale initialization $\gamma_0 = 10^{-6}$ (default), with $\gamma_0 = 1$, and with layer scale removed altogether, and compare their training curves. Both layer scale at

10^{-6} and stochastic depth (Section 7.6) make a deep network effectively shallower at the start of training: contrast how each mechanism works, when during training it is active, and whether it alters the evaluation-time network.

4. code **Global Response Normalization (ConvNeXt V2).** ConvNeXt V2 (Woo et al. 2023) removes layer scale and inserts Global Response Normalization (GRN) right after the GELU inside each block. For a feature map $\mathbf{X}$ with C channels, let $g_c = \|\mathbf{X}_c\|_2$ be the spatial ℓ_2 norm of channel c and $\bar{g} = \frac{1}{C}\sum_{c=1}^{C} g_c$. GRN rescales each channel by its relative response with a learnable affine transform and residual:

$$\mathrm{GRN}(\mathbf{X})_c = \gamma_c \frac{g_c}{\bar{g} + \epsilon}\mathbf{X}_c + \beta_c + \mathbf{X}_c, \tag{7.12}$$

where $\epsilon = 10^{-6}$ and γ, β are initialized to zero so GRN begins as the identity. Implement GRN in a few lines of code, insert it into `ConvNeXtBlock` in place of layer scale, and compare validation accuracy under the roadmap recipe of Table 7.7.

5. code **Depthwise kernel size sweep.** ● Under the recipe of Table 7.7, train `ConvNeXt` with 3×3, 7×7, and 11×11 depthwise kernels (padding 1, 3, and 5) and compare parameter counts, multiply-adds, wall-clock time per epoch, and validation accuracy. Why does the wall-clock time per epoch grow much faster from 7×7 to 11×11 than the total multiply-add count would suggest?

6. code **2×2 architecture-by-recipe ablation.** ● The first and last rows of Table 7.7 compare `CompactResNet18` and `ModernConvNeXt` under the modern AdamW/Mixup/smoothing recipe. Complete the 2×2 ablation square by training both architectures under Recipe A of Section 7.6 (momentum SGD, step decay, no Mixup or label smoothing, retuning the peak learning rate). Does the modern training recipe boost both architectures equally?

7.8 Convolutional Network Design Spaces

Most of the networks we built in this chapter were designed by hand, fixing a small set of structural hyperparameters at every stage: how many blocks to stack, how many channels each stage emits, and which block topology to use. Where do those numbers come from, and are there general principles governing them? Looking back across our models offers surprisingly little guidance: ResNet-18 uses $(2, 2, 2, 2)$ blocks per stage, ResNet-50 uses $(3, 4, 6, 3)$, and ConvNeXt-T uses $(3, 3, 9, 3)$—borrowed from Swin and validated once. Worse, these parameters interact tightly: under a fixed compute budget, making a stage deeper forces it to be narrower, and grouped convolutions free up compute for wider channels. Exhaustive search is out of the question: with four numbers per stage across four stages, even two choices per number yield $2^{16} = 65{,}536$ distinct networks, each requiring its own training run. And even if brute force unearthed one winning network, a single point estimate cannot reveal *which* design choices mattered or how to adapt them to a new compute budget or building block.

Neural architecture search (NAS) (Zoph and Le 2016; Liu et al. 2018), which produced EfficientNet (Section 7.5), automates single-model selection, yet like manual engineering it returns one point solution rather than reusable scientific rules. Radosavovic et al. (2020) took a fundamentally different path: comparing entire *design spaces*—that is, populations of networks generated by a shared template and a small set of constraints. By sampling networks

from a design space, training them briefly, and checking whether a proposed constraint improves the *error distribution* of the entire population, they distilled simple, generalizable design rules that define the *RegNet* family. Below we implement their AnyNet template, walk through their ImageNet population evidence (500 models per space), test their strongest rule ourselves on two small Fashion-MNIST populations, and train a compact RegNet head to head against ResNet-18.

7.8.1 The AnyNet Design Space

Across this chapter, modern convnets converged on the three-part stem–body–head template introduced in Section 7.2.3. The *stem* processes raw pixels with one or a few convolutions and usually downsamples immediately; the *body* performs the bulk of the computation through a sequence of *stages*, each repeating a uniform *block* at a fixed resolution while halving spatial dimensions and widening channels at every stage boundary; and the *head* maps the final feature tensor to class logits via global average pooling and a linear layer. ResNet's four stages emit features at $1/4$, $1/8$, $1/16$, and $1/32$ of the input resolution—a multi-scale pyramid that endured all the way to ConvNeXt largely because downstream object detectors and segmentation heads tap all four scales simultaneously (Section 32.5).

7.8.1.1 Stage Parameters of Hand-Designed Networks

Table 7.8 compares four landmark architectures from this chapter side by side. Notice that while channel widths cleanly double from stage to stage in every row, stage depths follow no obvious rule.

Table 7.8: Stage depths and widths of four hand-designed networks (ImageNet versions).

Network	Blocks per stage	Output channels per stage	Block
ResNet-18 (Section 7.4)	2, 2, 2, 2	64, 128, 256, 512	two 3×3 convolutions
ResNet-50 (He et al. 2016a)	3, 4, 6, 3	256, 512, 1024, 2048	$1 \times 1, 3 \times 3, 1 \times 1$ bottleneck
ResNeXt-50 (Section 7.4.5.1)	3, 4, 6, 3	256, 512, 1024, 2048	bottleneck with 32 groups
ConvNeXt-T (Section 7.7)	3, 3, 9, 3	96, 192, 384, 768	7×7 depthwise, inverted bottleneck

The *AnyNet* design space of Radosavovic et al. (2020) locks in this macro template while leaving stage hyperparameters free (Figure 7.22). Its stem is a single stride-2 3×3 convolution emitting $c_0 = 32$ channels, followed by batch normalization and ReLU (reading RGB images on ImageNet and a single grayscale channel on Fashion-MNIST). Its body comprises four stages, where stage i stacks d_i ResNeXt blocks (Figure 7.12): a 1×1 convolution projects c_i channels to c_i/b_i, a grouped 3×3 convolution processes groups of g_i channels, and a second 1×1 convolution restores c_i channels, with batch normalization after every convolution and a residual shortcut. (Note the paper's notation: b_i is the *bottleneck ratio*, so the bottleneck width that Section 7.4.5.1 denoted b is c_i/b_i here.) The opening block of each stage halves spatial resolution with stride 2 and projects the shortcut via a stride-2 1×1 convolution. Global average pooling and a linear classifier form the head.

Each stage therefore exposes four degrees of freedom: depth d_i, output width c_i, bottleneck ratio $b_i \geq 1$, and *group width* g_i (the number of channels per group; do not confuse this with

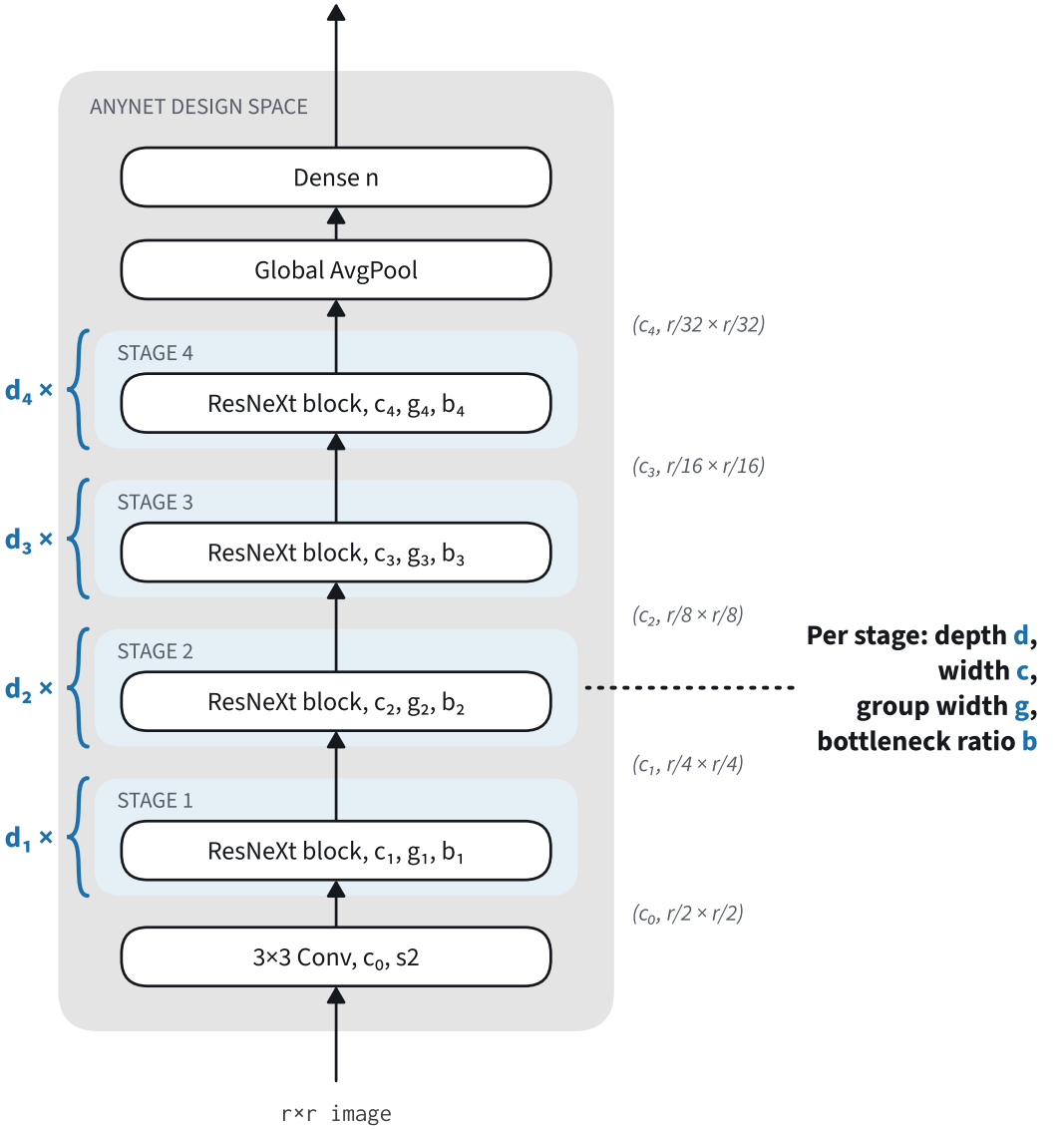


Figure 7.22: The AnyNet design space: a stem, a body of four stages, and a head. Stage i repeats a ResNeXt block d_i times; the first block of each stage halves the resolution. The annotations give the number of channels and the resolution after each part for an $r \times r$ input; the head outputs n class scores. The free choices per stage are the depth d_i, the width c_i, the group width g_i, and the bottleneck ratio b_i.

the *group count* g in Section 7.4.5.1, which remains the meaning of the `groups` argument in `d2l.ResNeXtBlock`). Across four stages, these 16 free numbers (sampling depths up to 16, widths up to 1024 in steps of 8, bottleneck ratios in $\{1, 2, 4\}$, and power-of-two group widths up to 32) span roughly $(16 \cdot 128 \cdot 3 \cdot 6)^4 \approx 2 \times 10^{18}$ candidate architectures: impossible to enumerate, yet trivial to sample randomly. On 224×224 ImageNet inputs, the stem and four stages perform five spatial halvings, ending at $224/2^5 = 7 \times 7$.

7.8.1.2 Implementing AnyNet

To explore this space in code, we build a general `AnyNet` constructor driven by stage tuples, starting with its three-layer stem.

```
class AnyNet(d2l.Classifier):
    def stem(self, num_channels):
        return nn.Sequential(
            nn.LazyConv2d(num_channels, kernel_size=3, stride=2, padding=1,
                          bias=False),
            nn.LazyBatchNorm2d(), nn.ReLU())
```

Each stage stacks `depth` instances of `d2l.ResNeXtBlock` (Section 7.4.5.1) parameterized by output width c_i, number of groups, and `bot_mul` $= 1/b_i$ (so group width equals (c_i/b_i)/groups), with stride 2 applied only to the opening block.

```
@d2l.add_to_class(AnyNet)
def stage(self, depth, num_channels, groups, bot_mul):
    blk = []
    for i in range(depth):
        if i == 0:
            blk.append(d2l.ResNeXtBlock(num_channels, groups, bot_mul,
                use_1x1conv=True, strides=2))
        else:
            blk.append(d2l.ResNeXtBlock(num_channels, groups, bot_mul))
    return nn.Sequential(*blk)
```

Assembling the stem, one stage per `(depth, num_channels, groups, bot_mul)` entry in `arch`, and the global-pooling head completes `AnyNet`. As in Section 7.4, all convolutional and linear weights use He initialization (Section 4.4.2.2), and convolutions omit biases before batch normalization.

```
@d2l.add_to_class(AnyNet)
def __init__(self, arch, stem_channels, lr=0.1, num_classes=10):
    super(AnyNet, self).__init__()
    self.save_hyperparameters()
    self.net = nn.Sequential(self.stem(stem_channels))
    for i, s in enumerate(arch):
        self.net.add_module(f'stage{i+1}', self.stage(*s))
    self.net.add_module('head', nn.Sequential(
        nn.AdaptiveAvgPool2d((1, 1)), nn.Flatten(),
        nn.LazyLinear(num_classes)))
```

```
@d2l.add_to_class(AnyNet)
def weight_init(self, module):
    # The Trainer calls this after the first minibatch fixed the shapes
    if type(module) in (nn.Linear, nn.Conv2d):
        nn.init.kaiming_normal_(module.weight, nonlinearity='relu')
        if module.bias is not None:
            nn.init.zeros_(module.bias)
```

Because every sampled model in a design space must be judged under the exact same training protocol, we attach the optimizer directly to `AnyNet`—matching the protocol used by Radosavovic et al. (2020) (and Recipe A in Section 7.6): SGD with momentum 0.9 (Section 7.5, Chapter 9), coupled weight decay (Section 2.7), and our chapter-wide `d2l.cosine_warmup` schedule (Section 7.1, warming up over the first 10% of `num_steps` to `lr` and decaying along a half-cosine to zero). Whereas the paper sets weight decay to 5×10^{-5} for 100 ImageNet epochs, for our shorter Fashion-MNIST runs we use 10^{-3}, which tuned best for `RegNetX32` (Section 7.8.3) while performing equally well for `ResNet18`.

```
@d2l.add_to_class(AnyNet)
def configure_optimizers(self):
    optim = torch.optim.SGD(self.parameters(), lr=self.lr, momentum=0.9,
                            weight_decay=1e-3)  # Decay added to the gradient
    warmup = self.num_steps // 10
    schedule = torch.optim.lr_scheduler.LambdaLR(
        optim, lambda t: d2l.cosine_warmup(t, self.num_steps, warmup))
    return optim, schedule  # The Trainer steps it after every minibatch
```

7.8.2 Comparing Design Spaces

How can we compare two massive design spaces without ever finding the single best architecture in either? Radosavovic et al. (2019) rely on four working assumptions so that a modestly sized sample of briefly trained models can faithfully represent an entire space:

1. **Good principles are abundant.** Many networks satisfying sound design rules perform well (there are many good needles in the haystack), so optimizing a *distribution* over architectures is meaningful.
2. **Short-run ranking transfers.** Validation accuracy after a brief training run correlates well with final converged accuracy (enabling 10-epoch ImageNet runs in the paper and 1-epoch Fashion-MNIST runs below).
3. **Small-scale findings extrapolate.** Rules discovered at low compute budgets continue to hold when scaled up to large models (verified at scale at the end).
4. **Design factors roughly decouple.** Individual structural constraints can be evaluated sequentially without severe hidden interactions.

None of these four assumptions is guaranteed a priori (the second and third are especially strong), and the exercises below test them directly at small scale.

Accepting these assumptions, we draw n random networks from a space, train each briefly, and record their validation errors $e_1, \ldots, e_n$. The *empirical cumulative distribution function* (CDF) gives, at every error threshold e, the fraction of sampled models achieving error at most e:

$$\hat{F}(e) = \frac{1}{n}\sum_{j=1}^{n} \mathbf{1}(e_j \leq e). \tag{7.13}$$

The higher and farther left a CDF curve rises, the richer the space is in accurate networks. When one space's CDF lies uniformly above another's, a randomly drawn model from the first space beats any error threshold with higher probability—even if the single best model in both spaces happens to coincide.

Using this lens, Radosavovic et al. (2020) sampled 500 networks per space at ~400 million multiply-adds (400 MFLOPs), trained each for 10 ImageNet epochs, and plotted the resulting CDFs in Figure 7.23, tightening the unconstrained AnyNetX$_A$ space one rule at a time. Sharing a single bottleneck ratio b across all four stages (AnyNetX$_B$) and then sharing a single group width g (AnyNetX$_C$) leaves the error distribution untouched (mean error ~49%)—a major win in its own right, eliminating 6 of the 16 free hyperparameters without accuracy loss. The dramatic breakthrough comes from constraining *stage widths*: requiring non-decreasing widths ($c_{i+1} \geq c_i$) slashes mean population error by six full points to ~43%, whereas constant or shrinking widths degrade mean error to 56% and 63%. Requiring non-decreasing depths ($d_{i+1} \geq d_i$) trims another half point while constant or shrinking depths fall behind.

Because empirical rules depend on the dataset, compute budget, and block type, it is instructive to apply the exact same population methodology to Fashion-MNIST for the single strongest rule: growing versus shrinking widths. We define two two-stage spaces using a 32-channel stem and ResNeXt blocks ($b = 1$, group width 16), drawing stage depths $d_i \in \{1, \ldots, 8\}$ and widths $c_i \in \{16, 32, \ldots, 256\}$ subject to $c_1 < c_2$ in one space and $c_1 > c_2$ in the other. Just as in the paper, we match compute budgets by tallying multiply-adds across the stem, every block, and all projection shortcuts (Section 7.1).

```
def mult_adds(arch, r, stem_channels=32, group_width=16, num_classes=10):
    """Multiply-adds of an AnyNet with k = 1 for one r x r image; arch holds
    (depth, width) for each stage."""
    r //= 2  # The stem halves the resolution
    total, c_in = r * r * 9 * stem_channels, stem_channels  # Grayscale input
    for d, c in arch:
        total += r * r * c_in * c  # First 1 x 1 convolution, before the stride
        r //= 2
        total += r * r * c_in * c  # 1 x 1 convolution of the shortcut
        total += r * r * c * c * (2 * d - 1)  # The other 1 x 1 convolutions
        total += r * r * 9 * group_width * c * d  # Grouped 3 x 3 convolutions
        c_in = c
    return total + c_in * num_classes  # Linear head
```

To draw a candidate model, we sample two depths and two distinct widths, sort the widths according to the space's constraint, and accept the draw only if its multiply-adds fall within 20% of our 112M budget per 96×96 image (matching the `RegNetX32` model we train in Section 7.8.3).

```
def sample_arch(rng, increasing, r=96, budget=112e6):
    """Depths and widths of two stages at about `budget` multiply-adds."""
```

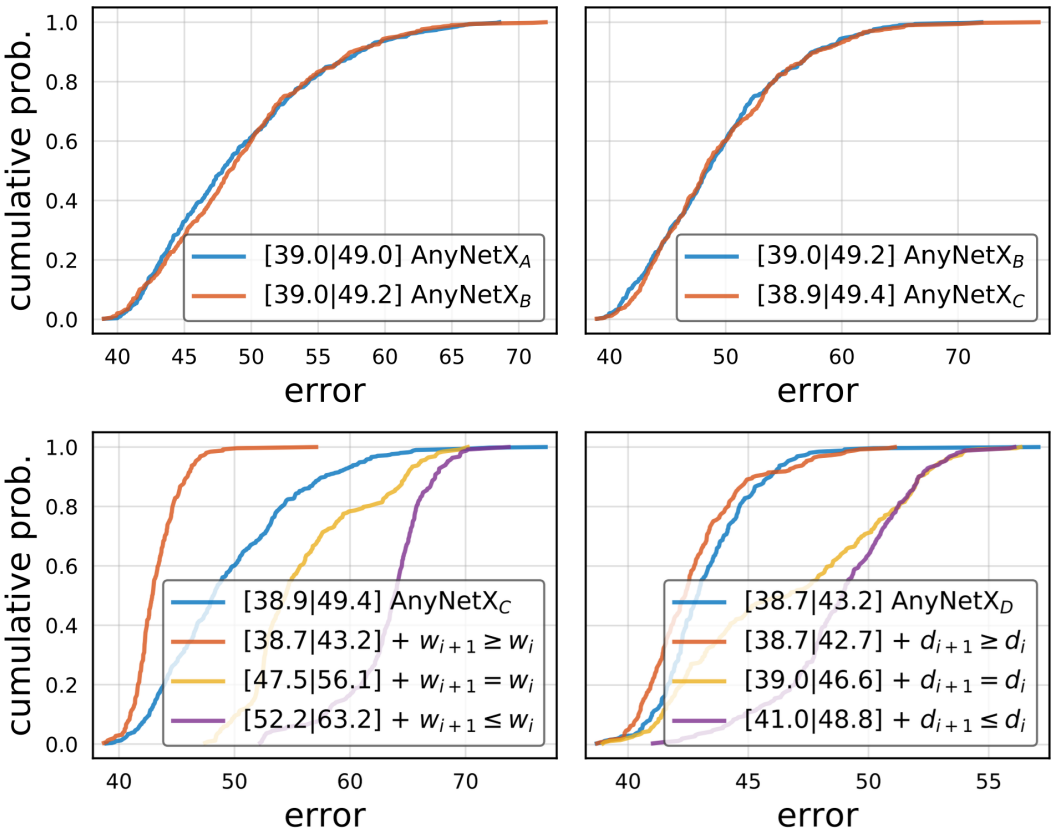


Figure 7.23: Error CDFs of design spaces, reproduced from Radosavovic et al. (2020): 500 networks sampled from each space, each trained for 10 epochs on ImageNet at about 400 million multiply-adds. Legends give the minimum and mean top-1 error in percent; w_i is the paper's symbol for the width c_i. Top: one bottleneck ratio for all stages (AnyNetX$_B$) against none (AnyNetX$_A$), then one group width as well (AnyNetX$_C$). Bottom: widths that increase, stay equal, or decrease across stages; then depths that do the same in the space with increasing widths (AnyNetX$_D$).

```
    while True:
        depths = rng.integers(1, 9, size=2).tolist()
        widths = sorted(rng.choice(np.arange(16, 257, 16), 2, replace=False)
                        .tolist(), reverse=not increasing)
        arch = tuple(zip(depths, widths))
        if abs(mult_adds(arch, r) / budget - 1) <= 0.2:
            return arch

rng = np.random.default_rng()
for increasing in (True, False):
    arch = sample_arch(rng, increasing)
    print(arch, f'{mult_adds(arch, 96) / 1e6:.0f} million multiply-adds')
```

```
((3, 32), (6, 208)) 114 million multiply-adds
((3, 112), (6, 96)) 109 million multiply-adds
```

At matched compute, models with decreasing widths spend their channels on high-resolution early maps and narrow down at the top. Below we train each sampled architecture for one epoch (one-tenth of our full runs in Section 7.8.3) using `AnyNet`'s optimizer and record its validation error.

```
def population_errors(rng, increasing, data, n, num_epochs=1):
    """Validation errors of n networks sampled from one design space."""
    errors = []
    for _ in range(n):
        arch = sample_arch(rng, increasing)
        model = AnyNet(tuple((d, c, c // 16, 1) for d, c in arch), 32)
        model.num_steps = num_epochs * len(data.train_dataloader())
        trainer = d2l.Trainer(max_epochs=num_epochs, num_gpus=1)
        trainer.board.display = False  # We plot the errors ourselves
        trainer.fit(model, data)
        errors.append(1 - float(trainer.history['val_acc'][-1]))
    return np.array(errors)
```

We draw and train 12 models per space (24 single-epoch runs in all).

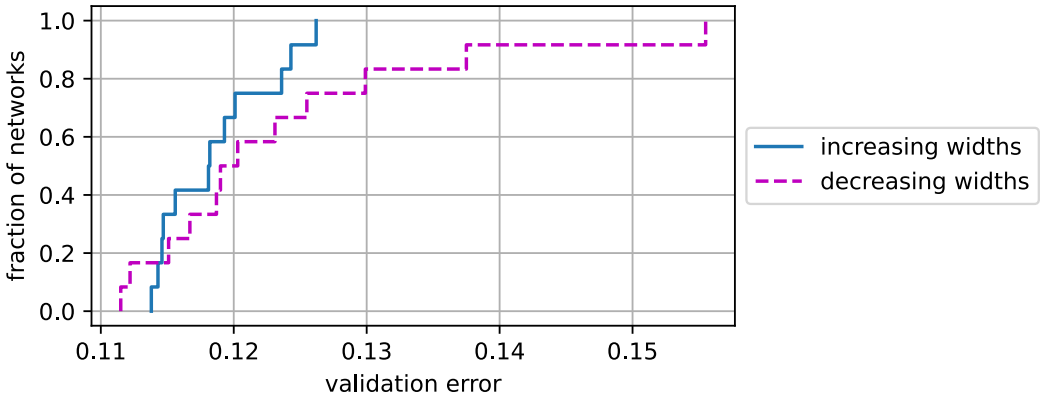
```
data = d2l.FashionMNIST(batch_size=128, resize=(96, 96))
spaces = {'increasing widths': True, 'decreasing widths': False}
errors = {name: population_errors(rng, increasing, data, 12)
          for name, increasing in spaces.items()}
```

Plotting the empirical CDF Equation 7.13 produces a step function jumping by $1/n$ at each sorted validation error.

```
def ecdf(e):
    """Corners of the step function of the empirical CDF of e."""
    e = np.sort(e)
    return np.repeat(e, 2), np.repeat(np.arange(len(e) + 1) / len(e), 2)[1:-1]

X, Y = zip(*(ecdf(e) for e in errors.values()))
d2l.plot(list(X), list(Y), xlabel='validation error',
```

```
            ylabel='fraction of networks', legend=list(errors),
            legend_loc='outside', figsize=(5, 2.5))
    for name, e in errors.items():
        print(f'{name}: mean error {e.mean():.3f}, minimum {e.min():.3f}')
```

```
increasing widths: mean error 0.119, minimum 0.114
decreasing widths: mean error 0.124, minimum 0.112
```



Here, the two CDFs largely overlap: across repeated draws, mean errors hover near 12% and differ by less than a single percentage point (usually nudging in favor of increasing widths), while the best models in both spaces land around 11%. A constraint worth twenty full points of mean error on ImageNet shrinks to under a point on Fashion-MNIST—too small for 12 samples per space to separate cleanly. That is a sobering yet illuminating finding: on an easy ten-class dataset where almost any reasonable convnet hits ~88% in a single epoch, architectural constraints have little room to separate populations. Design rules are never universal laws; their magnitude depends directly on the complexity of the task they were discovered on.

7.8.3 RegNet

7.8.3.1 Design Principles

Combining the surviving constraints from ImageNet defines the compact AnyNetX$_E$ design space:

- a single shared bottleneck ratio b across all stages;
- a single shared group width g across all stages;
- nondecreasing stage widths, $c_i \leq c_{i+1}$;
- nondecreasing stage depths, $d_i \leq d_{i+1}$.

Inspecting the top-performing networks inside AnyNetX$_E$ reveals one further striking pattern. Indexing blocks sequentially as $j = 0, 1, \ldots$ from bottom to top, optimal block widths follow a simple linear ramp,

$$u_j = u_0 + aj, \tag{7.14}$$

parameterized by an initial width u_0 and slope $a > 0$. Because every block inside a stage

shares one width, RegNet quantizes each u_j onto a geometric grid, much like the doubling widths of hand-crafted networks in Table 7.8, grouping consecutive blocks that snap to the same quantized width into a single stage to form a clean staircase. Figure 7.24 shows the staircase of one RegNet, and problem 2 asks you to work out another. Surprisingly, the authors also found that $b = 1$ (no internal bottleneck compression at all) consistently outperforms the $b = 4$ bottlenecks of ResNet-50 and ResNeXt-50. Combining quantized linear widths with $b = 1$ defines the *RegNetX* family, collapsing 16 arbitrary stage numbers down to just a few interpretable parameters per compute budget.

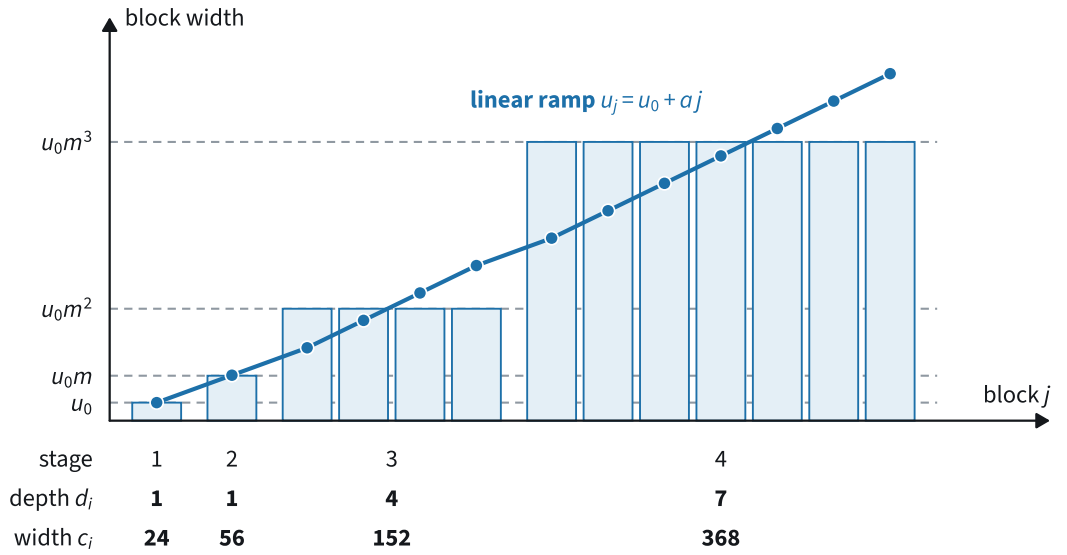


Figure 7.24: Quantized linear widths for RegNetX-200MF, a network of 13 blocks. The width of each block on the linear ramp $u_j = u_0 + aj$ (dots) snaps to the nearest level of the geometric grid $u_0, u_0m, u_0m^2, \ldots$ (dashed lines), and consecutive blocks at the same level form a stage. With $u_0 = 24$, $a = 36.44$, and $m = 2.49$ this gives four stages of depths 1, 1, 4, and 7, whose widths are 24, 56, 152, and 368 channels after rounding to multiples of 8.

7.8.3.2 RegNetX32 on Fashion-MNIST

Let us now construct a compact RegNet-style network obeying these rules and compare it directly with ResNet-18. Because upsampled 96×96 Fashion-MNIST images originate from 28×28 thumbnails, running four stages (five halvings) would shrink the final stages to tiny 6×6 and 3×3 grids; we therefore use two stages operating at 24×24 and 12×12 (problem 1 tests four stages). Behind a 32-channel stem, we hand-pick increasing depths and widths (stage 1 with $d_1 = 4$, $c_1 = 64$ and stage 2 with $d_2 = 6$, $c_2 = 160$) together with $b = 1$ and shared group width $g = 16$ (giving $64/16 = 4$ and $160/16 = 10$ groups). One stem convolution, ten three-layer blocks, and one linear classifier total 32 layers, giving `RegNetX32`.

```
class RegNetX32(AnyNet):
    def __init__(self, lr=0.1, num_classes=10, num_steps=None):
        stem_channels, group_width, bot_mul = 32, 16, 1
        depths, channels = (4, 6), (64, 160)
```

```
        super().__init__(  # The number of groups is c_i / g
            tuple((d, c, c // group_width, bot_mul)
                  for d, c in zip(depths, channels)),
            stem_channels, lr, num_classes)
        self.num_steps = num_steps  # Length of the learning-rate schedule
```

Inspecting `layer_summary` confirms that spatial resolution steps down cleanly from $96 \to 48 \to 24 \to 12$ as channel depth grows.

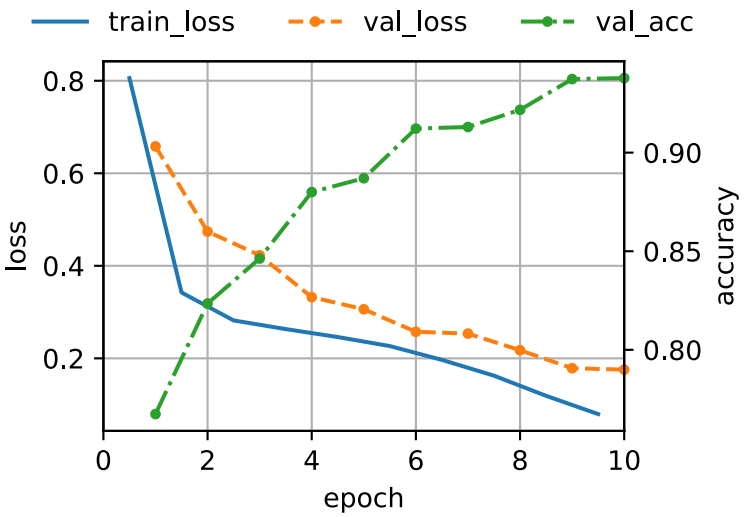
```
RegNetX32().layer_summary((1, 1, 96, 96))
```

```
Sequential              (1, 32, 48, 48)
Sequential              (1, 64, 24, 24)
Sequential              (1, 160, 12, 12)
Sequential              (1, 10)
```

Tallying arithmetic as in Section 7.1, `RegNetX32` executes 112 million multiply-adds per 96×96 image—nearly three times fewer than ResNet-18's 319 million.

Let us train `RegNetX32` on 96×96 Fashion-MNIST for 10 epochs with batch size 128 using `AnyNet`'s momentum-SGD optimizer at peak rate 0.1 (with one epoch of warmup).

```
data = d2l.FashionMNIST(batch_size=128, resize=(96, 96))
model = RegNetX32(lr=0.1, num_steps=10 * len(data.train_dataloader()))
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
trainer.fit(model, data)
```



Once the 10-epoch training run finishes, we call `d2l.num_params(model)` to report `RegNetX32`'s total parameter count alongside its final validation accuracy `trainer.history["val_acc"][-1]`.

```
print(f'{d2l.num_params(model):,} parameters, validation accuracy '
      f'{trainer.history["val_acc"][-1]:.3f}')
```

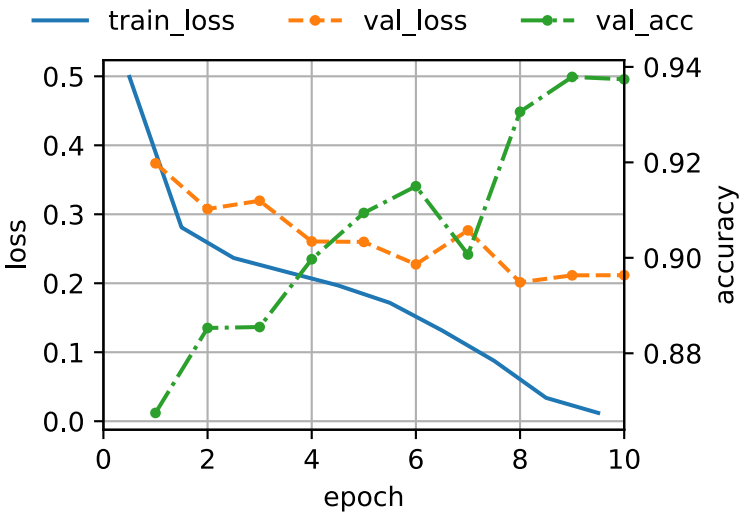
```
519,658 parameters, validation accuracy 0.938
```

To keep the comparison strictly controlled, we train ResNet-18 under the exact same `AnyNet` optimizer. Because ResNet-18 shares the same stem–body–head structure, we simply subclass `AnyNet` and override `stem` and `stage` using `d2l.Residual` (Section 7.4), passing `(depth, width, stride)` for each stage.

```
class ResNet18(AnyNet):
    def __init__(self, lr=0.1, num_classes=10, num_steps=None):
        super().__init__(((2, 64, 1), (2, 128, 2), (2, 256, 2), (2, 512, 2)),
                         64, lr, num_classes)
        self.num_steps = num_steps

    def stem(self, num_channels):
        return nn.Sequential(
            nn.LazyConv2d(num_channels, kernel_size=7, stride=2, padding=3,
                          bias=False),
            nn.LazyBatchNorm2d(), nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1))

    def stage(self, depth, num_channels, stride):
        blk = [d2l.Residual(num_channels, use_1x1conv=stride > 1,
                            strides=stride)]
        blk += [d2l.Residual(num_channels) for _ in range(depth - 1)]
        return nn.Sequential(*blk)
```

Now we train `ResNet18` on the exact same data for 10 epochs.

```
model = ResNet18(lr=0.1, num_steps=10 * len(data.train_dataloader()))
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
trainer.fit(model, data)
```



As before, we print `d2l.num_params(model)` and the final validation accuracy `trainer.history["val_acc"][-1]` so we can compare `ResNet18` directly against `RegNetX32`.

```
print(f'{d2l.num_params(model):,} parameters, validation accuracy '
      f'{trainer.history["val_acc"][-1]:.3f}')
```

```
11,175,370 parameters, validation accuracy 0.937
```

Both architectures finish near 94% validation accuracy (93.7–94.2% across runs), yet `RegNetX32` uses only 519,658 parameters compared with ResNet-18's 11,175,370—under 5% of the weights and a third of the multiply-adds (112M vs. 319M). Three nuances frame this result:

1. **Architecture–optimizer coupling:** Under plain SGD without momentum or weight decay (Section 7.4), `RegNetX32` drops to ~91% while ResNet-18 reaches ~93%; `RegNetX32` pulls even only under the momentum-and-weight-decay protocol on which the RegNet space was calibrated.
2. **Dataset ceiling:** Scaling up to four-stage RegNets (RegNetX-400MF, -800MF, -1.6GF; 4.8–8.3M parameters) plateaus at the same 93–94% ceiling on Fashion-MNIST.
3. **Arithmetic vs. GPU latency:** An epoch of `RegNetX32` takes roughly $1.5\times$ as long as one of ResNet-18, despite a third of the multiply-adds. Multiply-adds count arithmetic, not time (Section 7.5): `RegNetX32` is nearly twice as deep, and its many narrow, grouped layers write about $2.5\times$ as many activations per image, so memory traffic rather than arithmetic sets the pace. The RegNet authors found the same on ImageNet (Radosavovic et al. 2020): GPU runtime tracks activations more closely than multiply-adds, RegNetX models train and run about as fast as ResNets with equal activation counts, and at comparable multiply-adds RegNets run up to $5\times$ faster than EfficientNets, whose activations grow linearly with multiply-adds.

7.8.4 Summary

Comparing *design spaces* shifts architectural analysis from single point estimates to error distributions over populations sampled from a parameterized template (`AnyNet`). Across

500 briefly trained ImageNet samples per space, tying a single bottleneck ratio $b = 1$ and group width g across stages costs nothing while enforcing nondecreasing stage widths ($u_j = u_0 + aj$ quantized to a staircase) and depths cuts mean population error from 49% to 43%, yielding the RegNetX family. On Fashion-MNIST, `RegNetX32` matches ResNet-18's ~94% accuracy with under 5% of its parameters and a third of its multiply-adds. Across the entire chapter, a small core of structural primitives (repeatable blocks, activation normalization, and residual shortcuts) and the discipline of separating architectural gains from training-recipe gains form the foundation of modern vision backbones.

7.8.5 Exercises

1. code **Four stages at matched budget.** `RegNetX32` uses two stages on 96×96 Fashion-MNIST. Extend it to the four stages of Figure 7.22, keeping widths and depths nondecreasing across stages and the total parameter count within 10% of `RegNetX32`. Compare validation accuracy and time per epoch against `RegNetX32`, and state the spatial feature-map resolution at which each of the four stages operates.

2. **Quantized linear widths.** Radosavovic et al. (2020) parameterize the block widths of a d-block network via the linear ramp $u_j = u_0 + a\,j$ ($j = 0, \dots, d-1$) of Section 7.8.3, then snap each block to a geometric grid with ratio m:

$$s_j = \text{round}\left(\frac{\log(u_j/u_0)}{\log m}\right), \qquad \tilde{u}_j = u_0\, m^{s_j}, \tag{7.15}$$

   grouping blocks with equal $\tilde{u}_j$ into a single stage.

   1. Figure 7.24 shows the result for RegNetX-200MF. Repeat the computation for RegNetX-400MF, which has $d = 22$, $u_0 = 24$, $a = 24.48$, and $m = 2.54$ (the paper's w_0, w_a, w_m): compute the number of stages, the depth d_i of each stage, and the stage widths rounded to the nearest multiple of 8.
   2. How do the number of stages and the staircase approximation $\tilde{u}_j \approx u_j$ behave as $m \to 1$ versus as m becomes large?
   3. Which of your rounded stage widths from (a) are not divisible by a shared group width of $g = 16$, and how should they be adjusted?

3. code **RegNetY (adding Squeeze-and-Excitation).** RegNetY inserts the squeeze-and-excitation gate `SE` (Section 7.5.2.3, with reduction ratio $r = 4$) into every `ResNeXtBlock` right after its grouped 3×3 convolution (Radosavovic et al. 2020).

   1. Build `RegNetY32` with the stage configuration of `RegNetX32`. Predict the added parameter count using the $2c^2/r$ formula and verify it against `d2l.num_params`.
   2. Train `RegNetY32` on 96×96 Fashion-MNIST and compare its validation accuracy and time per epoch against `RegNetX32`. Why does `SE` add measurable wall-clock time despite adding almost zero multiply-adds?

4. code **Breaking the RegNet rules.** ● Implement three variants of `RegNetX32` at roughly matched parameter counts that each violate one AnyNetX$_E$ rule (Section 7.8.3): (i) decreasing stage depths ($d_1 > d_2$), (ii) unequal group widths across stages, and (iii) bottleneck compression $b_i > 1$ that differs across stages. Train each variant three times on Fashion-MNIST, compare against `RegNetX32`, and contrast your findings with the population CDFs in Figure 7.23. Why does testing a single hand-picked network per violation provide weaker evidence than comparing two empirical CDFs?

5. [extended] **A ConvNeXt design space.** ● Build a two-stage AnyNet design space using `ConvNeXtBlock` (Section 7.7, adding the expansion ratio $t \in \{2, 4\}$ as an argument) together with `ConvNeXt`'s stem and downsampling layers, sampling stage depths $d_i \in \{1, 2, 4\}$ and widths $c_i \in \{32, 64, 128\}$.
    1. Which of AnyNet's four per-stage degrees of freedom (d_i, c_i, g_i, b_i) have a direct counterpart in `ConvNeXtBlock`, and which are fixed by depthwise separability?
    2. Sample 16 networks from the unconstrained space, 16 with a shared expansion ratio $t_1 = t_2$, and 16 enforcing $c_1 \leq c_2$ and $d_1 \leq d_2$. Train each for 2 epochs on 96×96 Fashion-MNIST, plot the three empirical error CDFs Equation 7.13, and check which RegNet principles carry over to ConvNeXt blocks.

Chapter 8

Sequence Models

Having spent two chapters on images, we close this part of the book by turning to sequential data. Every model we have built so far expects an input of fixed shape, such as a tabular vector of attributes or a 28×28 grid of pixels, and treats each example independently. A sentence, an audio recording, a video, or a stream of sensor measurements behaves quite differently: its length varies from one example to the next, and its entries depend strongly on one another and on the order in which they arrive. Consider the opening sentence of H. G. Wells' *The Time Machine*, which serves as our running dataset throughout this chapter: "The Time Traveller (for so it will be convenient to speak of him) was expounding a recondite matter to us." Guessing the final word requires reading the nineteen words before it in order. How can a single model make that prediction when the preceding text might be arbitrarily long?

Claude Shannon framed this question as a guessing game (Shannon 1951), estimating the predictability of printed English by asking human readers to guess a passage one character at a time. A *language model* plays the same game: it assigns a probability to the next *token* (whether a character, a subword, or a whole word) given the tokens that preceded it, and can generate new text by feeding each chosen token back in as its next input.

Because the history already read grows with every step, restricting attention to a fixed window of recent tokens discards everything earlier. A *recurrent neural network* (RNN) instead maintains a fixed-size *hidden state* that summarizes the sequence read so far, updating that state with the same parameters after every token. Even where modern large language models replace recurrence with attention (Chapter 10) or structured state space models (Chapter 12), the core machinery of this chapter—subword tokenization, perplexity and bits-per-byte evaluation, and autoregressive decoding—carries over unchanged.

We begin by formulating autoregressive sequence forecasting and observing why multistep rollouts degrade much faster than one-step predictions (Section 8.1). Next, we convert raw text into subword token sequences via byte-level Byte Pair Encoding (Section 8.2) and build count-based n-gram language models alongside the perplexity and bits-per-byte metrics (Section 8.3). We then formulate and implement recurrent neural network language models from scratch and with high-level APIs (Section 8.4), stabilizing training with gradient clipping and comparing the learned representations against Kneser–Ney n-grams. Finally, Section 8.5 derives backpropagation through time to explain why vanilla RNN gradients vanish or explode, and Section 8.6 studies greedy search, beam search, and truncated sampling

methods for generating text.

8.1 Working with Sequences

When an email client sees the prefix "Thank you for your", it naturally proposes continuations such as "help" or "time". Likewise, a weather service that records the temperature every hour relies on recent readings to forecast the next one. In both situations, a good prediction depends not only on what appeared earlier, but also on the order in which it appeared: scrambling the words to "your thank for you" changes the continuation entirely, and a falling temperature curve calls for a very different forecast from a rising one. None of the models from earlier chapters can capture this structure directly, because they map a single input of fixed shape—a vector $\mathbf{x} \in \mathbb{R}^d$—to an output and treat each example as independent of the rest. A sentence, an audio recording, or a stream of sensor measurements violates both assumptions: its length varies from one instance to the next, and its entries depend strongly on one another.

In this section, we recast sequence modeling as a problem we already know how to solve: regression. We predict each entry from the entries that precede it, reducing the growing history to a fixed-size input either by keeping a window of the most recent observations or by maintaining a running summary of everything seen so far. To test these ideas in a setting where we know the exact ground truth, we fit a linear autoregressive model on a fixed window to a noisy sine wave. As we will discover, predicting a single step ahead is straightforward, whereas predicting many steps ahead by feeding the model its own predictions is considerably harder. The remainder of the chapter carries these same principles over to text.

8.1.1 Sequential Data

A *sequence* is an ordered list of feature vectors $\mathbf{x}_1, \dots, \mathbf{x}_T$, or in the scalar case, values $x_1, \dots, x_T$, where the index $t \in \mathbb{Z}^+$ denotes the *time step* and the length T may vary from one sequence to the next. Familiar examples abound: text is a sequence of characters, subwords, or words; audio (whether speech or music) is a sequence of waveform samples or spectrogram frames; and time series such as stock prices, sensor readings, and patient vital signs arrive one measurement at a time.

While we continue to assume that *entire* sequences, such as complete documents or full hospital stays, are drawn independently from a single underlying distribution over sequences, we certainly cannot assume that entries *within* a sequence are independent. The medication a patient receives on the tenth day of a hospital stay depends heavily on what transpired over the preceding nine days. Far from being a nuisance, this internal dependence is precisely what makes the past informative: if consecutive entries were independent, conditioning on earlier observations would tell us nothing about the next one. Nor do we need to assume that every position in a sequence shares the same marginal distribution; a document typically looks quite different at its closing paragraph than at its opening sentence, and a patient's condition evolves over the course of treatment toward recovery or decline.

Sequence prediction tasks arise in several distinct forms. We may want to assign a single label to an entire sequence (such as classifying the sentiment of a movie review), generate a sequence from a fixed input (such as captioning an image), or map an input sequence to an output sequence (such as translating a sentence or transcribing speech). Before tackling supervised targets of any kind, however, it is instructive to model the sequence itself

by estimating its joint probability $P(x_1, \dots, x_T)$. Estimating this distribution requires no external labels, because every entry naturally serves as the target for the entries preceding it. Moreover, a model that assigns probabilities to sequences can both generate new sequences and rank competing candidates—for instance, choosing the more plausible of two transcriptions for the same audio clip.

8.1.2 Autoregressive Models

Consider a numerical series such as a stock index or an hourly temperature record. A forecaster deciding how to act on the next measurement is primarily interested in the conditional distribution of the upcoming value given all observations seen so far:

$$P(x_t \mid x_1, \dots, x_{t-1}). \tag{8.1}$$

Because the prefix $x_1, \dots, x_{t-1}$ appears throughout this chapter, we abbreviate it as $x_{<t}$. Even when estimating the full conditional distribution is difficult, its conditional expectation $\mathbb{E}[x_t \mid x_{<t}]$ is already a useful summary—one we might attempt to fit with linear regression (Section 2.1). Any model that regresses a signal on its own past values is called an *autoregressive model*, since both the features and the target come from the same series.

There is, however, an immediate snag: the number of conditioning inputs in $x_{<t}$ grows with t, meaning that each time step presents a different number of features and cannot be fed directly into a fixed-input model. Two general strategies resolve this mismatch by compressing the history into a fixed-size representation (Figure 8.1). Both remain autoregressive, differing only in how much of the past they retain.

8.1.2.1 Fixed Windows

The first strategy conditions only on the m most recent observations, $x_{t-m}, \dots, x_{t-1}$, and discards all earlier history. For every $t > m$, each training example has exactly m features, so any fixed-input architecture—linear or deep—applies directly. This is the classical form of autoregression, with m called the *window length*. Fixed windows are simple to implement and easy to parallelize across time steps, because every prediction depends only on its own local slice of the series. Their drawback is a hard horizon: anything more than m steps in the past is completely invisible to the model.

8.1.2.2 Latent Summaries

The second strategy maintains a fixed-size running summary $\mathbf{h}_t$ of all observations $x_1, \dots, x_t$, updating it recursively as each new value arrives via $\mathbf{h}_t = g(\mathbf{h}_{t-1}, x_t)$ and predicting the next entry from that summary via $\widehat{x}_{t+1} = f(\mathbf{h}_t)$. Because $\mathbf{h}_t$ is never observed directly, such a model is called a *latent autoregressive model*. The state vector $\mathbf{h}_t$ occupies the same fixed amount of memory regardless of how long the sequence becomes, allowing the model to summarize an arbitrarily long history in bounded space. Recurrent neural networks (Section 8.4) are latent autoregressive models with a learned transition function g, as are the state space models of Chapter 12. Deciding what the hidden state should remember and how to update it reliably are the central design questions of those chapters.

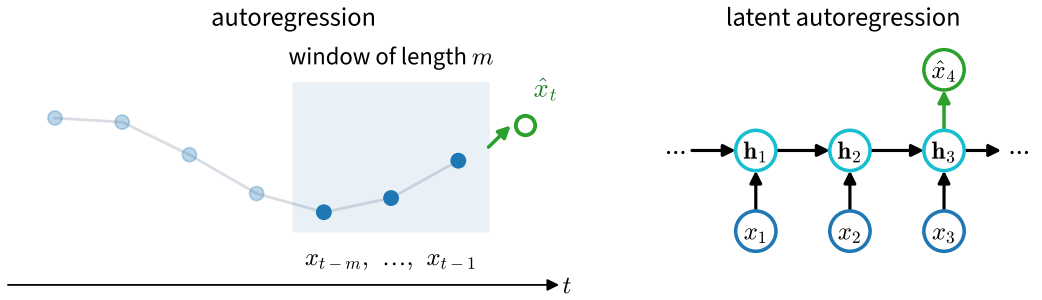


Figure 8.1: Two ways to give a sequence model a fixed-size input. (left) A fixed window conditions on the last m observations and discards the rest. (right) A latent summary carries a state $\mathbf{h}_t$ that summarizes the whole past in fixed size.

8.1.2.3 From Conditionals to Sequences

Whichever representation we choose, the one-step conditional distributions completely specify a probability model over the entire sequence. By the *chain rule of probability*, any joint distribution factors into a product of conditionals, one per time step, each conditioned on all preceding entries:

$$P(x_1, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_{<t}). \tag{8.2}$$

For $t = 1$, the prefix $x_{<1}$ is empty and the first factor is simply $P(x_1)$. Because this identity is exact for every joint distribution over sequences, modeling the next entry is mathematically equivalent to modeling the entire sequence. For a three-word phrase, for example,

$$P(\text{the}, \text{cat}, \text{sat}) = P(\text{the})\, P(\text{cat} \mid \text{the})\, P(\text{sat} \mid \text{the}, \text{cat}), \tag{8.3}$$

so if a model assigns probabilities 0.05, 0.01, and 0.2 to those three factors, the sentence receives joint probability 10^{-4}. In practice, we score a sequence by summing the logarithms of its conditionals rather than multiplying them directly, avoiding numerical underflow on long sequences. To generate a new sequence, we reverse the process: sample x_1, then sample x_2 given x_1, and continue feeding each newly drawn value into the next conditional distribution.

Notice that Equation 8.2 makes an implicit choice of ordering: we factored the sequence from left to right. In principle, any permutation of the indices yields a mathematically valid factorization. In causal domains, however, the forward direction is usually much easier to model, because predicting an effect from its causes tends to be simpler than inverting the process (Peters et al. 2017). Since forecasting the future from the past is also what we want in practice, we factor sequences left to right throughout this chapter.

When the sequence entries are discrete tokens, each conditional distribution is a classifier over the vocabulary, and estimating $P(x_1, \ldots, x_T)$ is known as *language modeling* (Section 8.3). For the remainder of this section, we stay with continuous values so that we can focus directly on the mechanics of autoregressive prediction.

8.1.3 Markov Models and Stationarity

Using a fixed window implicitly assumes that older observations can be safely ignored, while fitting a single model across all time steps assumes that the underlying predictive rule does not drift over time. Both assumptions are classical concepts in probability and time-series analysis.

8.1.3.1 The Markov Condition

When conditioning on the m most recent steps provides just as much predictive power as conditioning on the entire history, we say that the sequence satisfies a *Markov condition*: the future is conditionally independent of the distant past given the recent past. When $m = 1$, the process is a *first-order* Markov model; in general, it is an m^{th}-order Markov model. Under the first-order condition, each factor in Equation 8.2 depends solely on its immediate predecessor, and the joint probability simplifies to a product over adjacent pairs:

$$P(x_1, \dots, x_T) = P(x_1) \prod_{t=2}^{T} P(x_t \mid x_{t-1}). \tag{8.4}$$

Real-world sequences rarely obey a Markov condition exactly. In natural text, extending the context continues to yield useful information, albeit with diminishing returns for each additional token. Even so, assuming an m^{th}-order Markov condition is often a practical necessity, because it keeps the model tractable to compute and estimate from finite data; even modern large language models operate over a bounded context window rather than an infinite history. For discrete sequences, the simplest Markov model estimates $P(x_t \mid x_{t-1})$ directly by counting how often each successor follows each context—the exact principle behind the n-gram language models we build in Section 8.3.

8.1.3.2 Stationarity

Fitting a single predictor across all time steps relies on a second assumption: that the *dynamics* generating each entry from its predecessors do not change over time. Processes with time-invariant dynamics are called *time-homogeneous*—or, more loosely and far more commonly in machine learning, *stationary*. The observed values themselves may still fluctuate widely; what remains constant is the conditional distribution $P(x_t \mid x_{t-m}, \dots, x_{t-1})$. Crucially, a fixed transition rule does not force the entries to be identically distributed: the end of a document still behaves differently from its beginning because the same rule is evaluated on a different context. Stationarity is what justifies pooling examples from different parts of a single sequence, or across many sequences, by sliding a window along the series to assemble one shared training set.

8.1.4 Forecasting a Noisy Sine Wave

To see these concepts in action in a setting where we know the exact ground truth, let us fit a simple autoregressive model to synthetic data and compare one-step-ahead forecasting against multistep rollouts.

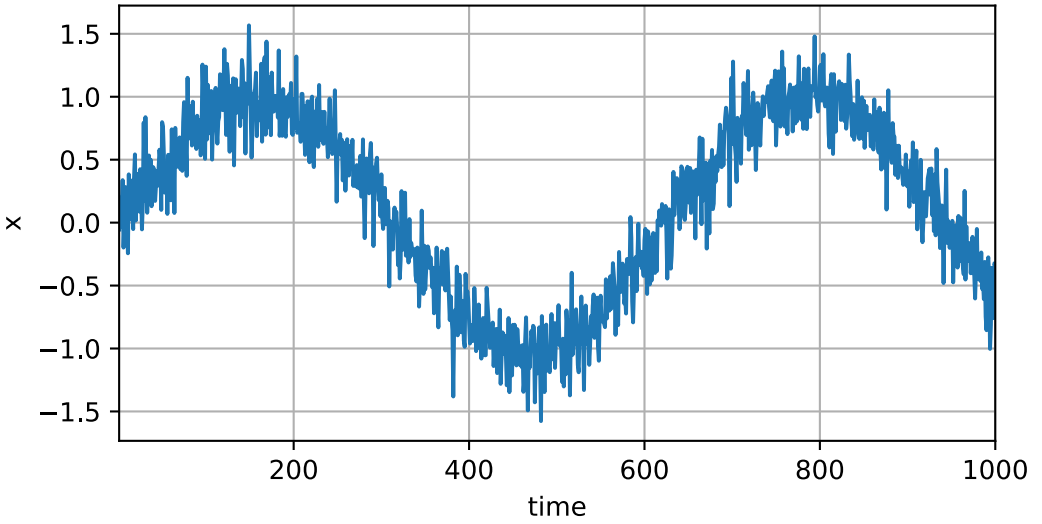
8.1.4.1 Data and Model

We generate a series of $T = 1000$ time steps from a noisy sine wave, $x_t = \sin(0.01\, t) + \epsilon_t$, where the additive noise $\epsilon_t \sim \mathcal{N}(0, 0.2^2)$ is drawn independently at every step. At this frequency, one full period spans $2\pi/0.01 \approx 628$ steps.

```
class Data(d2l.DataModule):
    def __init__(self, batch_size=16, T=1000, num_train=600, m=4):
        self.save_hyperparameters()
        self.time = d2l.arange(1, T + 1, dtype=d2l.float32)
        self.x = d2l.sin(0.01 * self.time) + d2l.randn(T) * 0.2
```

Instantiating `Data` and plotting `data.x` against `data.time` shows roughly one and a half periods of the noisy oscillation.

```
data = Data()
d2l.plot(data.time, data.x, 'time', 'x', xlim=[1, 1000], figsize=(6, 3))
```



To turn this series into supervised training examples, we apply the fixed-window strategy with window length m: each example pairs the feature vector $(x_{t-m}, \ldots, x_{t-1})$ with the target label $y = x_t$. This yields $T - m$ examples in total, since the first m entries lack sufficient history and are simply dropped (zero-padding is a common alternative). The `lag_features` method builds the entire feature matrix at once, with one column per lag, and the data loader uses the first 600 examples—just under one full period of the sine wave—for training. Figure 8.2 shows the resulting table for our setting of $m = 4$ and $T = 1000$.

```
@d2l.add_to_class(Data)
def lag_features(self):
    """Features (x_{t-m}, ..., x_{t-1}) and labels x_t for t > m."""
    features = [self.x[i : self.T-self.m+i] for i in range(self.m)]
    return d2l.stack(features, 1), d2l.reshape(self.x[self.m:], (-1, 1))
```

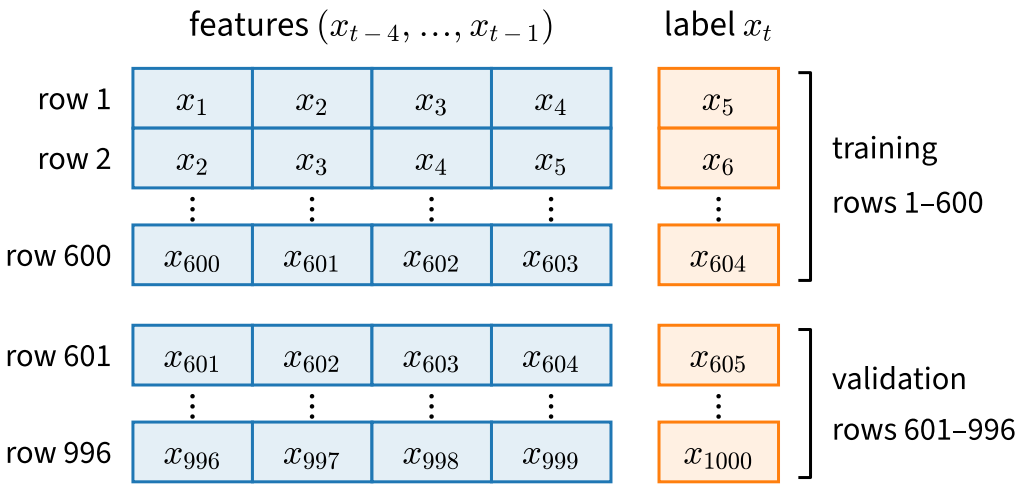
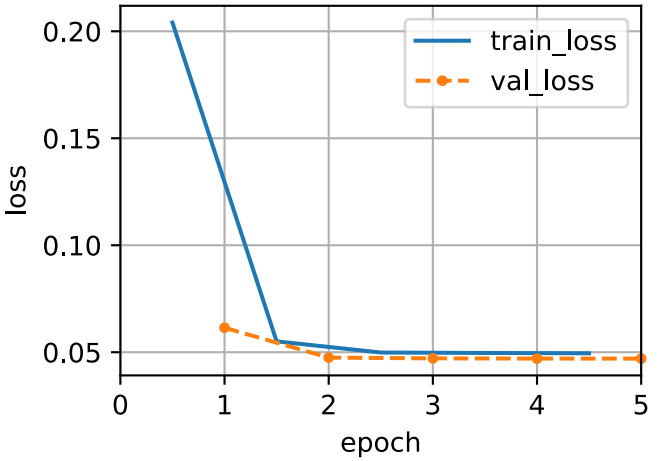


Figure 8.2: Lag features and labels for $m = 4$ and $T = 1000$. Row i pairs the four values $(x_i, \ldots, x_{i+3})$ with the label x_{i+4}, so the 996 rows end at x_{1000}. The first 600 rows are used for training and the remaining 396 for validation.

```
@d2l.add_to_class(Data)
def get_dataloader(self, train):
    i = slice(0, self.num_train) if train else slice(self.num_train, None)
    return self.get_tensorloader(self.lag_features(), train, i)
```

For our predictor, we use ordinary linear regression over the $m = 4$ lag features—about the simplest autoregressive model imaginable.

```
model = d2l.LinearRegression(lr=0.01)
trainer = d2l.Trainer(max_epochs=5)
trainer.fit(model, data)
```



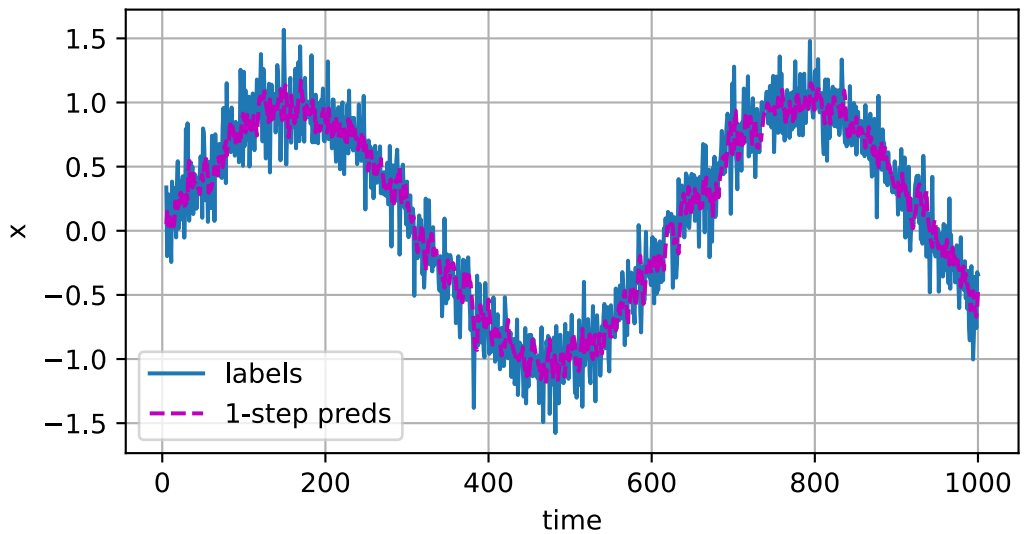
8.1.4.2 One-Step-Ahead Prediction

We begin with the easy case: predicting $\hat{x}_t$ from the *observed* preceding m values at every time step $t > m$.

```
features, labels = data.lag_features()
with torch.inference_mode():  # prediction only: record no gradients
    onestep_preds = d2l.numpy(model(features))
```

We overlay the one-step predictions `onestep_preds` on the ground-truth `labels` across all $T - m = 996$ time steps.

```
d2l.plot(data.time[data.m:], [labels, onestep_preds], 'time', 'x',
         legend=['labels', '1-step preds'], figsize=(6, 3))
```



The one-step predictions track the underlying wave closely, even beyond step 600 where the model is evaluated on data outside its training split.

8.1.4.3 Multistep Rollout

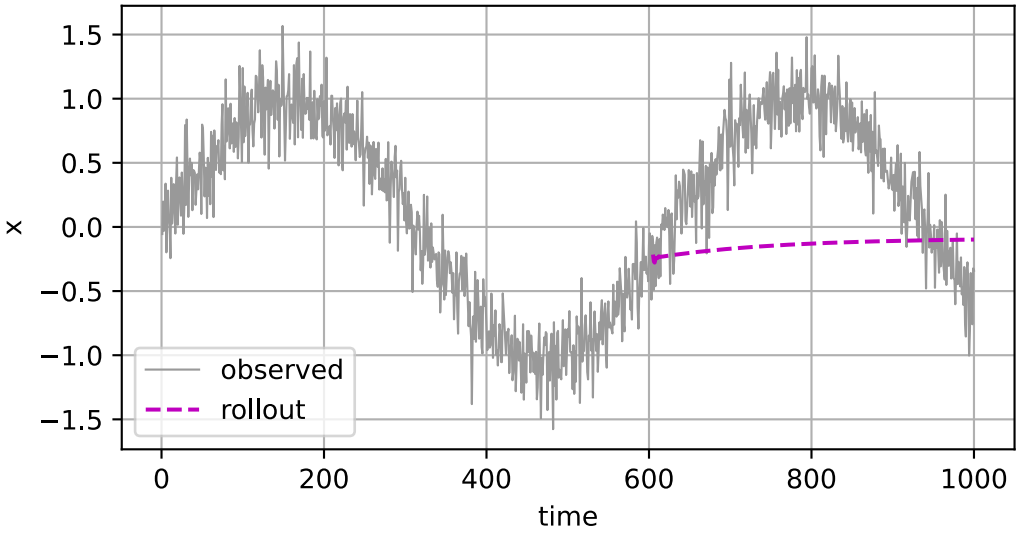
Forecasting multiple steps into the future is considerably more challenging. Suppose observations stop at time step t and we wish to forecast $\hat{x}_{t+5}$. We cannot evaluate the one-step predictor directly, because $x_{t+1}, \ldots, x_{t+4}$ have not been observed. Instead, we feed the model's own predictions back in as if they were true observations, stepping forward one time step at a time:

$$\begin{aligned}
\hat{x}_{t+1} &= f(x_{t-3}, x_{t-2}, x_{t-1}, x_t), \\
\hat{x}_{t+2} &= f(x_{t-2}, x_{t-1}, x_t, \hat{x}_{t+1}), \\
\hat{x}_{t+3} &= f(x_{t-1}, x_t, \hat{x}_{t+1}, \hat{x}_{t+2}), \\
\hat{x}_{t+4} &= f(x_t, \hat{x}_{t+1}, \hat{x}_{t+2}, \hat{x}_{t+3}), \\
\hat{x}_{t+5} &= f(\hat{x}_{t+1}, \hat{x}_{t+2}, \hat{x}_{t+3}, \hat{x}_{t+4}).
\end{aligned}$$

From the fifth step onward, the model relies exclusively on its own prior predictions. We refer to $\hat{x}_{t+k}$ as a *k-step-ahead prediction*, and to the process of chaining one-step predictions forward as a *rollout*. The code below rolls the linear model forward from $t = 604$—the last value used by the training split—through the end of the series.

```
multistep_preds = data.x.clone()
with torch.inference_mode():
    for i in range(data.num_train + data.m, data.T):
        multistep_preds[i] = model(
            multistep_preds[i-data.m : i].reshape(1, -1))
multistep_preds = d2l.numpy(multistep_preds)
```

```
start = data.num_train + data.m
d2l.plot([data.time, data.time[start:]], [data.x, multistep_preds[start:]],
         'time', 'x', figsize=(6, 3))
ax = d2l.plt.gca()
ax.lines[0].set(color='0.6', linewidth=0.8)  # The series: thin and grey
ax.legend(['observed', 'rollout']);
```



The rollout fails completely to follow the oscillation: starting near the level of the last observed window, it drifts gradually toward a constant while the true wave continues to rise and fall. How can a model with such a low one-step error break down so thoroughly? Inspecting the learned weights immediately explains why.

```
w, b = model.get_w_b()
print(f'weights on x[t-{data.m}], ..., x[t-1]:',
      d2l.numpy(d2l.reshape(w, -1)).round(3))
```

```
weights on x[t-4], ..., x[t-1]: [0.25  0.249 0.23  0.254]
```

All four weights sit close to $1/4$, meaning the model has simply learned to predict the next observation as the average of the previous four. For one-step prediction, this is a sensible strategy: the sine wave moves by at most 0.01 per step, whereas the additive noise has standard deviation 0.2. Averaging four noisy measurements cuts the noise variance by a factor of four (from 0.04 down to 0.01), while lagging the slowly varying curve by a step or two introduces negligible bias. Yet when fed its own outputs, a moving average can only perpetuate its current level—and because the weights sum to slightly less than one, that level slowly decays toward the rollout's fixed point $b/(1 - \sum_i w_i)$, where b is the bias (a ratio of two small numbers, so the exact asymptote varies from run to run). The model never learned the curvature of the sine wave because the one-step objective barely rewarded doing so, and over long horizons that omission dominates the error. Importantly, linearity itself is not the culprit. Without noise, a sine wave satisfies the exact second-order recurrence $x_t = 2\cos(0.01)\, x_{t-1} - x_{t-2}$, so just two lags with weights near 2 and -1 would reproduce the oscillation indefinitely. In the presence of noise, however, those weights would amplify the noise variance fivefold, so minimizing one-step squared error pushes the fit toward averaging instead (exercise 1).

8.1.4.4 Error Growth with the Horizon

To measure how rapidly the forecast degrades as the horizon k grows, we compute k-step-ahead predictions starting from every valid position in the series simultaneously. The

helper `k_step_pred` maintains a list of columns, one entry per starting index: it begins with the m observed lags and then appends each round of predictions computed from the most recent m columns, returning a list whose $(j-1)^{\text{th}}$ element contains the j-step-ahead forecasts.

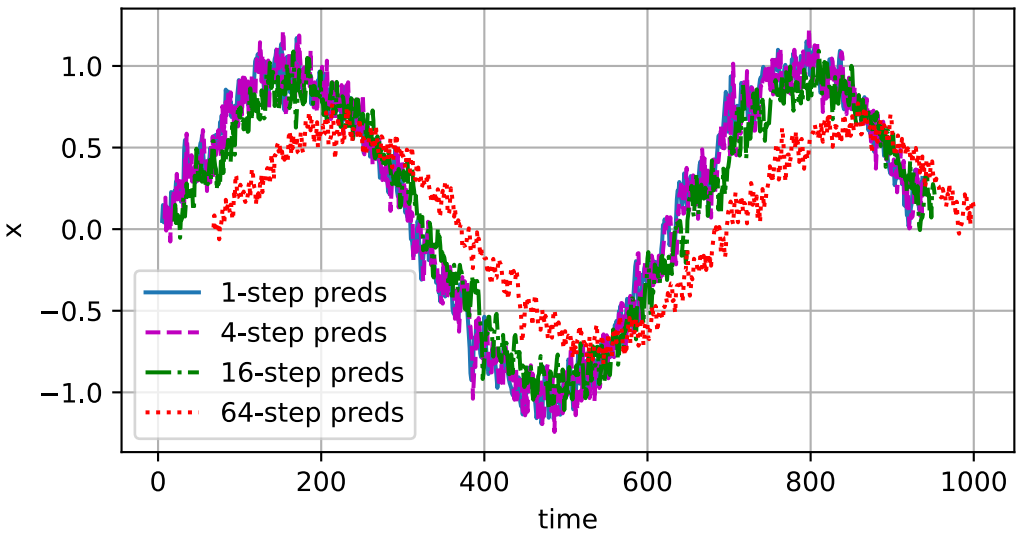
```
def k_step_pred(model, x, m, k):
    """Predictions 1, 2, ..., k steps ahead from every start position."""
    n = len(x) - m - k + 1                      # number of start positions
    columns = [x[i : i + n] for i in range(m)]   # observed lags
    for j in range(k):
        preds = model(d2l.stack(columns[j : j + m], 1))
        columns.append(d2l.reshape(preds, -1))
    return columns[m:]                          # entry j-1: j steps ahead
```

Using `k_step_pred`, we roll the trained model forward up to $k = 64$ steps ahead from every valid starting window in `data.x`.

```
steps = (1, 4, 16, 64)
with torch.inference_mode():
    preds = k_step_pred(model, data.x, data.m, steps[-1])
```

We now plot the resulting forecast curves at horizons $k \in \{1, 4, 16, 64\}$ from `preds` to see how errors accumulate with horizon length.

```
d2l.plot([data.time[data.m+k-1 : data.T-steps[-1]+k] for k in steps],
         [d2l.numpy(preds[k-1]) for k in steps], 'time', 'x',
         legend=[f'{k}-step preds' for k in steps], figsize=(6, 3))
```



The 1-step and 4-step forecasts track the series closely, whereas the 16-step forecast lags noticeably behind and the 64-step forecast looks like a copy of the wave shifted by roughly 64 steps and visibly damped in amplitude (with the degree of shrinkage governed by how

far the sum of weights falls below one). We can quantify this degradation via the mean squared error against the observed values, using the variance of the series—the error of a trivial constant forecast predicting the mean—as a baseline.

```
x = d2l.numpy(data.x)
for k in steps:
    err = d2l.numpy(preds[k-1]) - x[data.m+k-1 : data.T-steps[-1]+k]
    print(f'{k:2d} steps ahead: mean squared error {(err**2).mean():.3f}')
print(f'variance of the series: {x.var():.3f}')
```

```
 1 steps ahead: mean squared error 0.048
 4 steps ahead: mean squared error 0.049
16 steps ahead: mean squared error 0.066
64 steps ahead: mean squared error 0.246
variance of the series: 0.496
```

One step ahead, the mean squared error sits near 0.05: the irreducible noise variance $0.2^2 = 0.04$ plus the 0.01 noise variance remaining in an average of four observations. The subsequent growth with k is explained almost entirely by the phase lag. Because a k-step forecast roughly reproduces the level of the series k steps earlier, its additional error equals how much the sine wave moves over k steps. Averaged over the phase a of the wave, that squared change is

$$\frac{1}{2\pi}\int_0^{2\pi} \left(\sin(a + 0.01\,k) - \sin a\right)^2 da = 1 - \cos(0.01\,k), \tag{8.5}$$

which evaluates to 0.001 for $k = 4$, 0.013 for $k = 16$, and 0.20 for $k = 64$. Adding the baseline floor of 0.05 yields predicted errors of about 0.05, 0.05, 0.065, and 0.25—closely matching our measurements. By 64 steps ahead, the error is four to six times the one-step error and already roughly half the variance of the entire series.

Notice an important subtlety: none of this error arose from runaway accumulation. Because the averaging model damps whatever is fed back into it, its rollout decays smoothly toward a constant rather than diverging; the k-step forecast misses the target because the true wave has moved on, not because small errors compounded exponentially. More broadly, a model trained strictly on a one-step objective learns only what helps one step ahead. When rolled out over many steps, performance depends on capturing the true underlying dynamics, and a low one-step loss offers no guarantee that those dynamics have been learned.

Multistep rollouts also face a second, more famous failure mode that this damped linear model does not exhibit: because every prediction error becomes part of the input for all subsequent steps, a model that amplifies rather than damps perturbations can drift progressively further away from the data distribution. In language modeling, a few unfortunate sampled tokens can steer a generator into unnatural phrasing unlike anything seen during training, leading it off topic or into repetitive loops; learned physical simulators lose stability over long trajectories for much the same reason. Two broad strategies help mitigate this mismatch. During training, a model can be exposed to its own predictions alongside ground-truth histories so that imperfect prefixes are no longer out of distribution. At inference time, the decoding rule that converts each predicted distribution into a concrete output can be designed to suppress low-probability choices that derail a rollout—a topic we

return to for text in Section 8.6. First, however, we need to convert raw text into a representation that sequence models can process.

8.1.5 Summary

Autoregressive models cast sequence modeling as supervised prediction by estimating each entry from its predecessors, using either a fixed window of the m most recent observations (the principle behind n-grams) or a bounded-size latent state that summarizes the entire past (the principle behind recurrent networks). By the chain rule of probability, multiplying the one-step conditional distributions yields a complete probabilistic model of the sequence capable of both scoring and generating sequences. Fitting a single model across all time steps relies on stationarity—the assumption that the conditional rule generating each entry from its history does not change over time. Finally, strong one-step accuracy does not ensure strong multistep forecasting, because one-step training rewards only short-range features: on our noisy sine wave, an averaging linear model approaches the one-step noise floor yet incurs four to six times greater error 64 steps ahead, even before accounting for the compounding drift that afflicts models whose errors amplify during rollouts.

8.1.6 Exercises

1. code **Window length.** `Data` supplies `m=4` lag features and `d2l.LinearRegression` fits them.
    1. Increase `m` and retrain. How many past observations are needed before the validation error stops improving?
    2. Remove the noise term in `Data` and fit `m=2`. Compare the two weights with $2\cos(0.01)$ and -1, and roll the model out. Then restore the noise at standard deviations between 0.001 and 0.2. At what noise level do the weights give up the curvature and turn to averaging?
2. code **Nonlinear autoregressor.** Replace `d2l.LinearRegression` by a small MLP on the same four lag features and retrain, possibly for more epochs. How do the one-step and the multistep predictions change?
3. code **Error growth with the horizon.** For each k, the last entry of the list that `k_step_pred(model, data.x, data.m, k)` returns holds the k-step-ahead predictions of `data.x[data.m+k-1:]`.
    1. For $k \in \{1, 2, 4, \dots, 256\}$, plot the mean squared k-step error against k on a logarithmic axis. Where does the curve bend upward, and how does the bend relate to the damping of the 16-step and 64-step forecasts?
    2. Find the smallest k at which the mean squared k-step error exceeds the variance of the series. What does a forecast beyond this horizon offer over the constant forecast?
4. code **Dilated and averaged lags.** ● Keep four features for `d2l.LinearRegression` but let them reach further back: once as the lags $x_{t-1}, x_{t-2}, x_{t-4}, x_{t-8}$, and once as the four block means $\frac{1}{4}\sum_{i=0}^{3} x_{t-4j+i}$ for $j = 1, \dots, 4$. Compare their validation errors with those for $m = 4$ and $m = 16$. Which construction helps, and why?
5. code **Regime switching.** ● Replace the single sine in `Data` by a series that alternates between $\sin(0.01\,t)$ and $\sin(0.05\,t)$ every 200 time steps, and keep the additive noise. Both regimes then occur among the training examples and among the validation examples. The switching series violates the stationarity assumption of Section 8.1.3.2.

Does one linear model with $m = 4$ nevertheless fit both regimes, and why? Report training and validation error for each regime separately and locate the positions along the series where the fit is worst.

8.2 From Text to Tokens

Most sequences we model in this chapter are text, which introduces a challenge that the synthetic numerical series of Section 8.1 did not: neural networks operate on vectors of numbers, whereas text arrives as a raw string of characters. Before we can train any language model, we must decide how to segment a string into *tokens* (the atomic units treated as observations x_t) and how to map each token to a numerical index. This seemingly mundane preprocessing choice has far-reaching consequences: it determines the size of the model's embedding and output layers, the sequence length of every document the model reads, and even how hard tasks such as arithmetic turn out to be. Splitting *The Time Machine* into individual characters forces a model to step through roughly 170,000 time steps; splitting it into whole words requires an output class for each of about 4,600 distinct words while still breaking on any word not seen during training. For this reason, every modern language model ships with a dedicated tokenizer, and it is best to think of the tokenizer as an integral part of the model itself.

We begin by comparing character-level and word-level tokenization, which lie at opposite ends of the trade-off between vocabulary size and sequence length. We then drop down to raw bytes as a universal base alphabet and learn a subword vocabulary directly from data using *byte pair encoding* (BPE), the scheme underlying the GPT family of models (Sennrich et al. 2016; Radford et al. 2019). Our from-scratch implementation is only a few dozen lines of Python, yet it reproduces GPT-2's production tokenizer token for token.

8.2.1 Reading the Dataset

Throughout this chapter, our running corpus is H. G. Wells' The Time Machine, a novella of just over 30,000 words. Although real-world pretraining corpora run to trillions of tokens, this dataset is rich enough to exhibit the statistical properties of natural language while remaining small enough to tokenize in seconds. The `_download` method below fetches the book and reads the raw text into a single string.

```
import collections
import re
import tiktoken
```

```
class TimeMachine(d2l.DataModule):
    """The Time Machine dataset."""
    def _download(self):
        fname = d2l.download(d2l.DATA_URL + 'timemachine.txt', self.root,
                             '090b5e7e70c295757f55df93cb0a180b9691891a')
        with open(fname) as f:
            return f.read()

data = TimeMachine()
raw_text = data._download()
```

```
raw_text[:60]
```

```
'The Time Machine, by H. G. Wells [1898]\n\n\n\n\nI\n\n\nThe Time Tra'
```

Classical NLP pipelines normalized raw text aggressively—stripping out punctuation, folding all letters to lowercase, and only then splitting the cleaned string into tokens. We keep a simple normalization step here for the character- and word-level comparisons below, as well as for later sections that still use those pipelines. Still, such preprocessing is undeniably crude: it discards informative signals (capitalization distinguishes "Bill" from "bill") and bakes in brittle assumptions about which characters matter. By contrast, the byte-level tokenizer we construct later in this section requires no normalization whatsoever.

```
@d2l.add_to_class(TimeMachine)
def _preprocess(self, text):
    return re.sub('[^A-Za-z]+', ' ', text).lower()

text = data._preprocess(raw_text)
text[:60]
```

```
'the time machine by h g wells i the time traveller for so it'
```

8.2.2 Characters, Words, and Bytes

8.2.2.1 Characters and Words

Because a sequence model processes one token per time step, the two most natural choices of token are individual characters and whole words. Let us examine both, starting with characters.

```
@d2l.add_to_class(TimeMachine)
def _tokenize(self, text):
    return list(text)

tokens = data._tokenize(text)
','.join(tokens[:30])
```

```
't,h,e, ,t,i,m,e, ,m,a,c,h,i,n,e, ,b,y, ,h, ,g, ,w,e,l,l,s, '
```

Splitting on whitespace produces word tokens instead. For each scheme, we count both the number of *distinct* token types (which sets the vocabulary size) and the total number of tokens in the corpus (which sets the sequence length).

```
words = text.split()
print(f'characters: {len(set(tokens)):>6} distinct, '
      f'{len(tokens):>7} tokens in the corpus')
print(f'words:      {len(set(words)):>6} distinct, '
      f'{len(words):>7} tokens in the corpus')
```

```
characters:     27 distinct,  173428 tokens in the corpus
words:        4579 distinct,   32775 tokens in the corpus
```

Character-level tokenization of our normalized text requires only 27 symbols and can never encounter an unknown letter, yet it stretches the novella across more than 170,000 time steps, forcing any long-range dependency, such as agreement between a subject and its verb, to span dozens of steps. Word-level tokenization compresses the same text into roughly one-fifth as many tokens, each carrying far more semantic content. Even on this short novella, however, splitting into words already yields thousands of distinct types, and the vocabulary grows continually with the corpus: every proper name, typographical error, and inflection ("traveller", "travellers", "travelled") claims its own entry, most of which appear only once or twice (Section 8.3). Worse still, any word absent from the training set is *out of vocabulary* (OOV) at test time. Figure 8.3 illustrates this fundamental trade-off: enlarging the vocabulary shortens the resulting sequences, and the central design question is where along this spectrum to operate. Subword tokenizers allow us to choose that operating point flexibly, and most of the remainder of this section is devoted to building one.

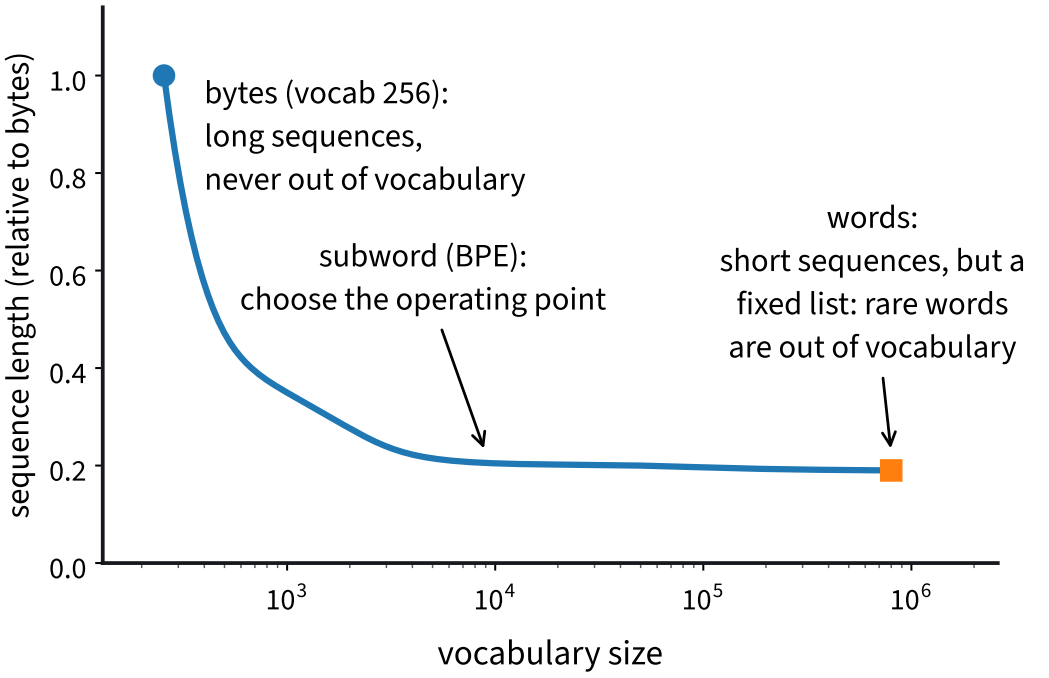


Figure 8.3: Tokenization granularities on one trade-off curve: a larger vocabulary encodes the same text in fewer tokens. Bytes never fall out of vocabulary; a fixed word list does. Byte pair encoding interpolates between the extremes.

8.2.2.2 From Tokens to Indices

Before moving to subwords, let us complete the classical pipeline, which several later sections of the book still use. Because neural networks operate on integer indices rather than strings, character and word pipelines rely on a *vocabulary* object that assigns each distinct token an index, sorted from most to least frequent, and maps unseen tokens to a reserved `<unk>` entry. Rare tokens occurring fewer than `min_freq` times can also be pruned to keep the vocabulary compact, falling back to `<unk>` as well.

```
class Vocab:
    """Vocabulary for text."""
    def __init__(self, tokens=(), min_freq=0, reserved_tokens=()):
        # Flatten a 2D list if needed
        if tokens and isinstance(tokens[0], list):
            tokens = [token for line in tokens for token in line]
        # Count token frequencies
        counter = collections.Counter(tokens)
        self.token_freqs = sorted(counter.items(), key=lambda x: x[1],
                                  reverse=True)
        # The list of unique tokens, ordered by descending frequency.
        # Reserve <unk> at index 0 so vocab[0] is the unknown token.
        self.idx_to_token = ['<unk>'] + list(reserved_tokens) + [
            token for token, freq in self.token_freqs
            if freq >= min_freq and token not in reserved_tokens]
        self.token_to_idx = {token: idx
                             for idx, token in enumerate(self.idx_to_token)}

    def __len__(self):
        return len(self.idx_to_token)

    def __getitem__(self, tokens):
        if not isinstance(tokens, (list, tuple)):
            return self.token_to_idx.get(tokens, self.unk)
        return [self.__getitem__(token) for token in tokens]

    def to_tokens(self, indices):
        if isinstance(indices, (list, tuple)) or getattr(indices, 'ndim', 0):
            return [self.idx_to_token[int(index)] for index in indices]
        return self.idx_to_token[int(indices)]

    @property
    def unk(self):  # Index for the unknown token
        return self.token_to_idx['<unk>']
```

To test `Vocab`, we build a vocabulary over our character `tokens`, map the first ten characters to integer `indices`, and invert them with `vocab.to_tokens`.

```
vocab = Vocab(tokens)
indices = vocab[tokens[:10]]
print('indices:', indices)
print('tokens:', vocab.to_tokens(indices))
```

```
indices: [3, 9, 2, 1, 3, 5, 13, 2, 1, 13]
tokens: ['t', 'h', 'e', ' ', 't', 'i', 'm', 'e', ' ', 'm']
```

The `build` convenience method packages this entire classical pipeline (preprocessing, tokenizing, and indexing) into a single call that returns the corpus of integer indices alongside its vocabulary.

```
@d2l.add_to_class(TimeMachine)
def build(self, raw_text, vocab=None):
    tokens = self._tokenize(self._preprocess(raw_text))
    if vocab is None: vocab = Vocab(tokens)
    corpus = [vocab[token] for token in tokens]
    return corpus, vocab

corpus, vocab = data.build(raw_text)
len(corpus), len(vocab)
```

```
(173428, 28)
```

8.2.2.3 Text as Bytes

To interpolate smoothly between characters and words, we first need a base alphabet capable of representing every possible string. Characters themselves are surprisingly ill-suited for this role: Unicode defines roughly 150,000 characters across scripts, diacritics, mathematical symbols, and emoji, so a character-level vocabulary ceases to be small as soon as we step beyond plain English. Modern tokenizers drop one level lower. Every string stored on a computer is ultimately represented as a sequence of *bytes* under a character encoding, almost universally UTF-8. Because a single byte takes one of only $2^8 = 256$ values, a byte-level base vocabulary contains just 256 symbols and can *never* encounter an out-of-vocabulary input: any string, in any language, complete with punctuation, code, or typos, decomposes cleanly into bytes.

UTF-8 achieves this universality through a variable-length encoding that uses one byte for standard ASCII characters and up to four bytes for less common scripts and symbols.

```
s = 'Hello, naïve café: 你好 🙂'
print(len(s), 'characters,', len(s.encode('utf-8')), 'bytes')
for ch in 'a é 你 🙂'.split():
    print(repr(ch), '->', list(ch.encode('utf-8')))
n_bytes = len(raw_text.encode('utf-8'))
print(f'The novella: {n_bytes} bytes, {n_bytes / len(words):.1f} per word')
```

```
23 characters, 32 bytes
'a' -> [97]
'é' -> [195, 169]
'你' -> [228, 189, 160]
'🙂' -> [240, 159, 154, 128]
The novella: 178979 bytes, 5.5 per word
```

The price of this universal alphabet is sequence length. Our novella expands to nearly 180,000 bytes (more than five tokens per word), and because accented letters such as “é” require two bytes, Chinese characters three, and emoji four, raw byte sequences stretch non-English text even further. Everything that follows preserves the universality of bytes while compressing sequences back down to manageable lengths.

8.2.3 Byte Pair Encoding

8.2.3.1 The Merge Rule

Byte pair encoding (BPE) began as a compression scheme: replace the most frequent pair of adjacent bytes by a new symbol, and repeat (Gage 1994). Sennrich et al. (2016) applied the same rule to the characters inside words to obtain subword units for machine translation, and GPT-2 returned it to raw bytes (Radford et al. 2019). This byte-level variant is the one we build. It begins with the 256 single-byte symbols and grows its vocabulary greedily: at each iteration, it counts every adjacent *pair* of tokens in the training corpus, merges the most frequent pair into a single new token, and repeats until the vocabulary reaches its target size. Because each merge shortens the corpus by exactly the number of occurrences it replaces, BPE acts as a greedy compression algorithm whose learned vocabulary forms a dictionary of the corpus's most common recurring fragments.

Tracing the algorithm by hand makes its behavior transparent. Consider a toy corpus containing five distinct words across eight occurrences:

```
hug pug pun bun hugs hug hug pug
```

The raw corpus is 32 bytes long. The most frequent adjacent pair is (`u`, `g`), appearing six times (inside "hug", "pug", and "hugs"), so the first merge introduces the new token `ug`. After substituting `ug` everywhere, the most frequent adjacent pair becomes (`h`, `ug`), and the process continues:

step	most frequent pair	count	new token	corpus length
0	(start from bytes)			32
1	`u`, `g`	6	`ug`	26
2	`h`, `ug`	4 (tie)	`hug`	22
3	`hug`, `' '`	3 (tie)	`'hug '`	19

Note that steps 2 and 3 involve ties: (`ug`, `' '`) also appears four times at step 2, while (`' '`, `p`) and (`' '`, `hug`) also appear three times at step 3. Following standard convention, we break ties in favor of the pair that occurs earliest in the corpus.

Two important properties emerge directly from this trace. First, merges compose hierarchically: because `hug` is formed from `h` and `ug`, the learned vocabulary defines a *merge tree* over the byte alphabet (Figure 8.4), allowing any unseen text to be segmented by replaying the learned merges. The word "hugs", for instance, splits cleanly into `hug` and `s`, while the rare word "bun" remains three raw bytes—any substring untouched by learned merges simply falls back to individual bytes. Second, notice that step 3 fused a word with its *trailing space*. While a trailing space is benign on its own, nothing prevents subsequent merges from attaching the beginning of the following word as well. On a real corpus, unconstrained BPE readily creates multi-word tokens, a flaw we remedy shortly.

8.2.3.2 Implementation

We implement the BPE algorithm in three concise pieces. First, the `BPETokenizer` class holds the tokenizer's state: token ids 0 through 255 represent raw bytes, and each learned merge receives the next available integer id, so ids double as merge *ranks* (a smaller id indicates an earlier merge). The dictionary `merges` maps each merged pair of ids to its new id, while `vocab` maps every id to its corresponding byte string. The `pattern` and `specials` arguments, along

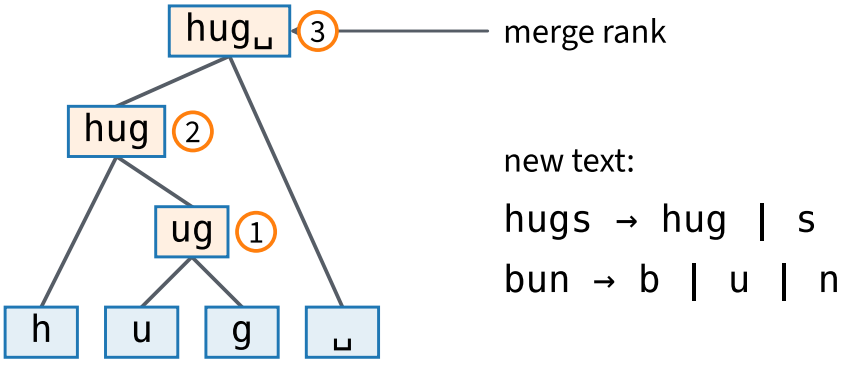


Figure 8.4: The first three merges learned on the toy corpus form a tree over the byte alphabet. New text reuses the tree: “hugs” becomes two tokens, while unseen fragments fall back to bytes.

with `byte_ids`, come into play later in this section and can be ignored for the moment. We also import the `regex` package, an extension of Python’s standard `re` module that supports Unicode character classes (such as `\p{L}` for any letter) needed for pre-tokenization.

```
import regex

class BPETokenizer:
    """Byte-level BPE tokenizer trained by iterated most-frequent-pair
    merges.

    Token ids 0..255 are raw bytes; learned merges get ids
    256..vocab_size-1; special tokens sit above those and are never
    produced by BPE itself. `pattern` is an optional pre-tokenization
    regex: text is split into chunks and merges never cross chunk
    boundaries.
    """

    GPT2_PATTERN = (r"'s|'t|'re|'ve|'m|'ll|'d| ?\p{L}+| ?\p{N}+"
                    r"| ?[^\s\p{L}\p{N}]+|\s+(?!\S)|\s+")

    def __init__(self, vocab_size=1024, pattern=None,
                 specials=('<pad>', '<bos>', '<eos>')):
        self.vocab_size, self.pattern = vocab_size, pattern
        self.merges = {}                                # (id, id) -> new id
        self.vocab = {i: bytes([i]) for i in range(256)}  # id -> bytes
        self.byte_ids = list(range(256))                # byte value -> id
        self.specials = {s: vocab_size + i for i, s in enumerate(specials)}

    def __len__(self):
        return self.vocab_size + len(self.specials)
```

```
    @property
    def pad(self):
        return self.specials['<pad>']

    @property
    def bos(self):
        return self.specials['<bos>']

    @property
    def eos(self):
        return self.specials['<eos>']

    def _chunks(self, text):
        return [text] if self.pattern is None else regex.findall(
            self.pattern, text)
```

Both training and encoding depend on a single core operation: replacing every occurrence of a target pair of adjacent ids with the id of their merged token. The helper `_merge` scans left to right, writing `new_id` and advancing by two whenever `pair` matches, and copying one id otherwise. Consequently, on the string `aaa`, the pair (`a`, `a`) is replaced once on the left rather than overlappingly.

```
@d2l.add_to_class(BPETokenizer)
def _merge(self, ids, pair, new_id):
    """Replace every occurrence of pair in ids by new_id."""
    out, i = [], 0
    while i < len(ids):
        if i + 1 < len(ids) and (ids[i], ids[i + 1]) == pair:
            out.append(new_id)  # Merge the pair and skip past it
            i += 2
        else:
            out.append(ids[i])
            i += 1
    return out
```

Training repeats the same loop we traced by hand: count all adjacent pairs, merge the most frequent pair everywhere, and record the resulting rule. Just as in the trace, ties are broken in favor of whichever pair was seen first—the exact behavior of Python's `max`. One practical optimization is that identical text chunks are stored only once, with their pair counts weighted by the chunk's frequency in the corpus. Before pre-tokenization, the entire corpus is a single chunk with weight 1; once pre-tokenization breaks text into words, a common word is processed once instead of thousands of times. Production BPE trainers go further by updating pair counts incrementally around each merge; here we simply recount after each step, trading speed for clarity.

```
@d2l.add_to_class(BPETokenizer)
def train(self, text):
    """Learn vocab_size - 256 merges, most frequent pair first."""
    chunk_freq = collections.Counter(self._chunks(text))
    seqs = [list(chunk.encode('utf-8')) for chunk in chunk_freq]
```

```
    for new_id in range(256, self.vocab_size):
        pairs = collections.Counter()
        for seq, w in zip(seqs, chunk_freq.values()):
            for pair in zip(seq, seq[1:]):
                pairs[pair] += w  # A chunk that occurs w times counts w times
        if not pairs:
            break  # nothing left to merge
        pair = max(pairs, key=pairs.get)
        self.merges[pair] = new_id
        self.vocab[new_id] = self.vocab[pair[0]] + self.vocab[pair[1]]
        seqs = [self._merge(seq, pair, new_id) for seq in seqs]
```

Training `BPETokenizer` for three merges (`vocab_size=256 + 3`) on our toy corpus and inspecting `toy_tok.merges` reproduces the three merge steps from our hand trace above:

```
toy_tok = BPETokenizer(vocab_size=256 + 3)
toy_tok.train('hug pug pun bun hugs hug hug pug')
for (a, b), new_id in toy_tok.merges.items():
    print(f'{toy_tok.vocab[a]} + {toy_tok.vocab[b]} -> '
          f'{toy_tok.vocab[new_id]}')
```

```
b'u' + b'g' -> b'ug'
b'h' + b'ug' -> b'hug'
b'hug' + b' ' -> b'hug '
```

8.2.3.3 Encoding and Decoding

Training produces a merge table. To *encode* new text, we start from its raw UTF-8 bytes and repeatedly merge the adjacent pair with the lowest merge rank until no adjacent pair appears in the table. Notice that encoding consults only the merge ranks—never corpus counts—so it is a pure function of the learned table: **the tokenizer is completely defined by its merge ranks**. Applying the lowest-ranked merge first faithfully retraces training, because each training merge was learned on text where all earlier merges had already been applied (a property formalized in the exercises). Decoding is simpler still: every token id maps directly to a byte sequence, so we concatenate the bytes and decode them as UTF-8, using `errors='replace'` so that any partial or malformed byte sequence emitted by a generative model degrades gracefully instead of raising an exception.

```
@d2l.add_to_class(BPETokenizer)
def _encode_chunk(self, text_bytes):
    ids = [self.byte_ids[b] for b in text_bytes]
    while len(ids) > 1:
        # The lowest-rank merge applicable anywhere in this chunk
        pair = min(zip(ids, ids[1:]),
                   key=lambda p: self.merges.get(p, float('inf')))
        if pair not in self.merges:
            break
        ids = self._merge(ids, pair, self.merges[pair])
    return ids
```

```
@d2l.add_to_class(BPETokenizer)
def encode(self, text, allow_special=False):
    if allow_special and self.specials:
        pat = '(' + '|'.join(regex.escape(s) for s in self.specials) + ')'
        ids = []
        for part in regex.split(pat, text):
            if part in self.specials:
                ids.append(self.specials[part])
            elif part:
                ids.extend(self.encode(part))
        return ids
    return [i for chunk in self._chunks(text)
            for i in self._encode_chunk(chunk.encode('utf-8'))]

@d2l.add_to_class(BPETokenizer)
def decode(self, ids):
    specials = {i: s.encode('utf-8') for s, i in self.specials.items()}
    data = b''.join(self.vocab[i] if i in self.vocab else specials[i]
                    for i in ids)
    return data.decode('utf-8', errors='replace')
```

On the toy tokenizer, encoding matches our hand trace and decoding recovers the original string cleanly.

```
print([toy_tok.vocab[i] for i in toy_tok.encode('hugs')])
print([toy_tok.vocab[i] for i in toy_tok.encode('bun')])
toy_tok.decode(toy_tok.encode('hug pug hugs'))
```

```
[b'hug', b's']
[b'b', b'u', b'n']
```

```
'hug pug hugs'
```

8.2.3.4 Training on The Time Machine

To see what BPE learns from real prose, we train it on *The Time Machine*. Because token lists and generated passages often exceed the width of a printed page, we first define a small printing helper that wraps long lines neatly for use here and throughout the rest of the book.

```
import textwrap

def print_wrapped(*args, width=76):
    """Print like `print`, folding each line to the width of a page."""
    for line in ' '.join(str(a) for a in args).split('\n'):
        print(textwrap.fill(line, width, subsequent_indent='    '))
```

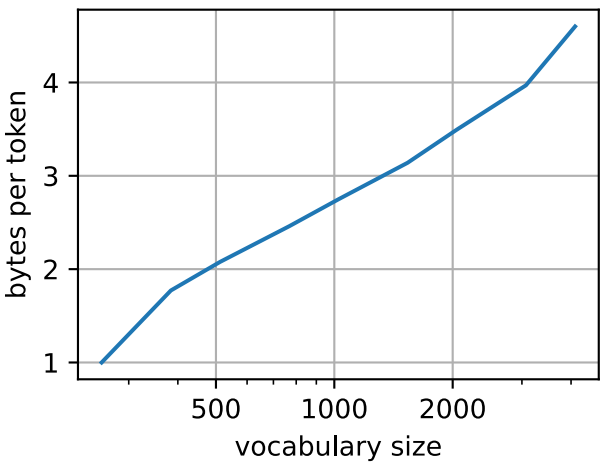
We now train a 1,024-token vocabulary directly on the raw, unnormalized text of the novella. The earliest merges capture common letter pairs, whereas the final merges assemble whole words from Wells' prose. Notice, however, that one token in the encoded sample below is the two-word composite `"Time Traveller "`, spaces and all—an artifact we examine next.

```
tok = BPETokenizer(vocab_size=1024)
tok.train(raw_text)
merged_ids = list(tok.merges.values())
print('first:', [tok.vocab[i] for i in merged_ids[:8]])
print('last: ', [tok.vocab[i] for i in merged_ids[-8:]])
ids = tok.encode('The Time Traveller smiled. Are you sure?')
print_wrapped(len(ids), 'tokens:', [tok.vocab[i] for i in ids])
```

```
first: [b'e ', b'th', b'd ', b'in', b't ', b's ', b'an', b'er']
last:  [b'appe', b'vo', b'ering ', b'too ', b'door', b'acro', b'clo', b'after ']
13 tokens: [b'The ', b'Time Traveller ', b's', b'mi', b'le', b'd', b'. A',
    b'r', b'e ', b'you ', b'su', b're', b'?']
```

How much shorter does a sequence become as we enlarge the vocabulary? Because merges are numbered in the exact order they are learned, any smaller BPE vocabulary is a *prefix* of a larger one trained on the same corpus: retaining only merges with id strictly below $|\mathcal{V}|$ reproduces the exact tokenizer we would have obtained by stopping training at size $|\mathcal{V}|$. A single training run therefore lets us sweep vocabulary sizes from 256 up to 4,096. Below, we measure compression in bytes per token on the first 10,000 characters of the novella (exercise 5 evaluates held-out text).

```
big = BPETokenizer(vocab_size=4096)
big.train(raw_text)
excerpt = raw_text[:10000]
sizes = [256, 384, 512, 768, 1024, 1536, 2048, 3072, 4096]
ratios = []
for v in sizes:
    sub = BPETokenizer(vocab_size=v)  # the first v - 256 merges of big
    sub.merges = {p: i for p, i in big.merges.items() if i < v}
    sub.vocab = {i: b for i, b in big.vocab.items() if i < v}
    ratios.append(len(excerpt.encode('utf-8')) / len(sub.encode(excerpt)))
d2l.plot(sizes, [ratios], 'vocabulary size', 'bytes per token',
         xscale='log')
```



Each doubling of the vocabulary buys roughly a constant gain in compression, so bytes

per token grows approximately logarithmically with vocabulary size. This scaling follows directly from the greedy merge rule: early merges such as `"e "`, `"th"`, and `"d "` collapse thousands of occurrences apiece, whereas the 4,000th merge collapses only a handful. We should be careful, however, not to extrapolate this curve blindly to production vocabularies of 50,000 to 200,000 tokens. The upper end of our plot is flattered because we evaluated on the same text used to train the tokenizer, where a 4,096-token vocabulary has memorized words and phrases specific to *The Time Machine*. On general English prose, production vocabularies typically achieve four to five bytes per token—roughly one token per word—as we verify below, devoting their extra capacity to rarer words, source code, and (in multilingual vocabularies such as `o200k_base`) other languages.

8.2.3.5 Pre-Tokenization

The composite `"Time Traveller "` is hardly an isolated case. Alongside true word fragments, our unconstrained tokenizer accumulates dozens of tokens that straddle word boundaries:

```
crossers = [v.decode('utf-8', 'replace') for i, v in tok.vocab.items()
            if i >= 256 and b' ' in v.strip(b' ')]
print_wrapped(len(crossers), 'tokens cross a word boundary, e.g.',
              crossers[:8])
```

```
78 tokens cross a word boundary, e.g. [', and ', 'of the ', '. I ', '. Th',
    '. A', '. I', 'in the ', 'ed to ']
```

Although multi-word merges like `"of the "` help compress the training text, they are brittle composites: they consume vocabulary slots on fixed phrases while fragmenting the statistical evidence for their constituent words (since `"of"` now appears scattered across many distinct tokens). Every production BPE tokenizer therefore inserts a *pre-tokenization* stage before merging. A regular expression first splits the input string into chunks, and BPE is trained and applied strictly *within* each chunk so that merges never cross chunk boundaries. GPT-2's pattern (Radford et al. 2019), stored as `BPETokenizer.GPT2_PATTERN`, consists of five branches:

```
's|'t|'re|'ve|'m|'ll|'d        common English contractions, split off
| ?\p{L}+                      a run of letters, with its leading space
| ?\p{N}+                      a run of digits, with its leading space
| ?[^\s\p{L}\p{N}]+            a run of punctuation or other symbols
|\s+(?!\S)|\s+                 remaining whitespace
```

This pattern encodes three deliberate conventions: each word retains its *leading space* (`" the"` rather than `"the "`) so that spacing travels with the word itself; letters, digits, and punctuation are separated into distinct chunks so that numbers and symbols never fuse onto words; and common English contractions are split off as standalone chunks.

```
regex.findall(BPETokenizer.GPT2_PATTERN,
              "The traveller's clock struck 12,345 times.")
```

```
['The',
 ' traveller',
 "'s",
 ' clock',
 ' struck',
```

```
 ' 12',
 ',',
 '345',
 ' times',
 '.']
```

Enabling pre-tokenization requires only passing `pattern` to `BPETokenizer`; Figure 8.5 illustrates the full pipeline from raw text to token ids. Compared with the unconstrained run, every boundary-crossing composite vanishes, freeing those vocabulary slots for complete words paired with their leading spaces. The earliest merges now attach frequent letters to leading spaces (`" t"`, `" a"`) or join common letter pairs (`"he"`, `"in"`).

```
bpe = BPETokenizer(vocab_size=1024, pattern=BPETokenizer.GPT2_PATTERN)
bpe.train(raw_text)
crossers = [v for i, v in bpe.vocab.items()
            if i >= 256 and b' ' in v.strip(b' ')]
print('boundary-crossing tokens now:', len(crossers))
gained = sorted(set(bpe.vocab.values()) - set(tok.vocab.values()),
                key=lambda v: (-len(v), v))
print_wrapped('longest tokens gained instead:',
              [v.decode('utf-8', 'replace') for v in gained[:6]])
print('first merges:', [bpe.vocab[i] for i in list(bpe.merges.values())[:6]])
```

```
boundary-crossing tokens now: 0
longest tokens gained instead: [' Psychologist', 'sychologist', '
    Traveller', ' Morlocks', ' darkness', ' Machine']
first merges: [b' t', b'he', b' a', b'in', b' the', b' s']
```

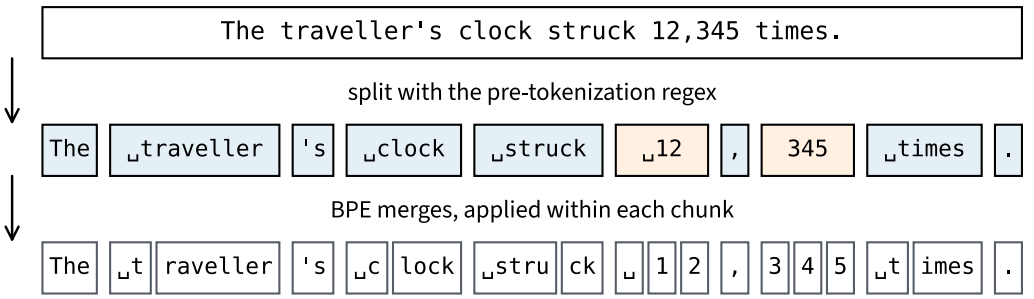


Figure 8.5: Pre-tokenization in action on a real sentence. The regex first cuts the text into chunks: words with their leading spaces, digit runs, and punctuation. BPE merges are then applied within each chunk, and can never cross a chunk boundary.

8.2.4 Special Tokens

Training and serving pipelines also require a small set of control tokens that carry meaning for the *infrastructure* rather than the text itself: <pad> pads shorter sequences to a common batch length, <bos> marks the start of a sequence, and <eos> lets a model signal that generation has finished (the decoders of Section 8.6 watch for it as a stopping cue). Our `BPETokenizer` places these special ids *above* the BPE range—a deliberate safety invariant ensuring that ordinary text can never accidentally encode to a special token. Otherwise, any document or

user message containing the literal characters "<eos>" could inject a control signal directly into the model's input stream, a classic vector for *prompt injection* in which untrusted user text is interpreted as a system instruction. Accordingly, our encode method treats special token strings as ordinary bytes unless the caller explicitly passes allow_special=True, mirroring tiktoken's allowed_special guard (which raises an error by default).

```
print('as plain text:  ', bpe.encode('<bos>the time machine'))
print('as control code:', bpe.encode('<bos>the time machine',
                                     allow_special=True))
print('pad/bos/eos ids:', bpe.pad, bpe.bos, bpe.eos, '| len:', len(bpe))
```

```
as plain text:   [60, 98, 111, 115, 62, 341, 513, 680]
as control code: [1025, 341, 513, 680]
pad/bos/eos ids: 1024 1025 1026 | len: 1027
```

8.2.5 Tokenizers in the Wild

8.2.5.1 The Rank Table of a Production Tokenizer

Most widely deployed language models, including the GPT family and Llama from version 3 onward, rely on byte-level BPE tokenizers built along these exact lines, simply trained on vastly larger corpora. GPT-2's tokenizer (Radford et al. 2019) is stored compactly as a lookup table mapping each token's byte sequence to its merge rank, paired with the pre-tokenization regex and special-token registry; there is no neural network inside it. (Other subword families, such as WordPiece or Unigram, additionally store text normalization rules or token probabilities used to pick among competing segmentations.) OpenAI's tiktoken library ships GPT-2's rank table directly. Below, we inspect entries right across the boundary between single bytes and learned merges—noting that the leading underscore on _mergeable_ranks indicates an internal attribute whose name may change across library releases.

```
enc = tiktoken.get_encoding('gpt2')
print(list(enc._mergeable_ranks.items())[254:260])
print(enc._special_tokens)
```

```
[(b'\xa0', 254), (b'\xad', 255), (b' t', 256), (b' a', 257), (b'he', 258), (b'in', 259)]
{'<|endoftext|>': 50256}
```

Ranks 0 through 255 correspond to the individual bytes in a permuted order (GPT-2 lists printable ASCII characters first), followed from rank 256 onward by the learned merges in training order—the exact same layout as our BPETokenizer. Remarkably, GPT-2's first four merges learned from 40 gigabytes of web text (" t", " a", "he", and "in") are identical to the first four merges bpe learned from a single nineteenth-century novella, reflecting the stability of the most frequent English character patterns.

8.2.5.2 Verifying Our Implementation

Because a BPE tokenizer is completely specified by its rank table, we can load GPT-2's published ranks straight into our own BPETokenizer encoder. Each rank becomes a token id, and the 256 single-byte entries define the byte permutation. Since the published table records

only the final bytes of each token rather than the pair of parents that formed it, we reconstruct each merge pair by replaying all lower-ranked merges on the token's bytes until exactly two pieces remain. The reconstruction details below can be skimmed; the key point is the verification cell that follows.

```
@d2l.add_to_class(BPETokenizer)
@classmethod
def from_tiktoken(cls, name):
    """Load a published bytes->rank table (e.g. 'gpt2') into our encoder."""
    import tiktoken  # lazy: d2l itself does not require tiktoken
    enc = tiktoken.get_encoding(name)
    ranks = enc._mergeable_ranks
    tok = cls(vocab_size=len(ranks), pattern=enc._pat_str, specials=())
    tok.specials = dict(enc._special_tokens)
    tok.vocab = {rank: b for b, rank in ranks.items()}
    tok.byte_ids = [ranks[bytes([b])] for b in range(256)]
    for token, rank in ranks.items():
        if len(token) > 1:  # recover which pair merged into this token
            parts = [bytes([b]) for b in token]
            while len(parts) > 2:
                a, b = min(zip(parts, parts[1:]),
                           key=lambda p: ranks.get(p[0] + p[1],
                                                   float('inf')))
                i = list(zip(parts, parts[1:])).index((a, b))
                parts[i:i + 2] = [a + b]
            tok.merges[ranks[parts[0]], ranks[parts[1]]] = rank
    return tok
```

On a test paragraph mixing prose, numbers, accented characters, currency symbols, and an emoji, our encoder matches `tiktoken` token for token—an assertion checked automatically whenever this book is built.

```
ours = BPETokenizer.from_tiktoken('gpt2')
para = ('The Time Traveller (for so it will be convenient to speak of '
        'him) was expounding a recondite matter to us; his pale grey '
        'eyes shone and twinkled. Prices rose 3.5% in 1895, and naïve '
        'café patrons paid £12,345 more!  ')
assert ours.encode(para) == enc.encode(para)
print(len(ours.encode(para)), 'tokens, identical to tiktoken')
print([ours.vocab[i].decode('utf-8', 'replace')
       for i in ours.encode(para)[:8]])
```

```
60 tokens, identical to tiktoken
['The', ' Time', ' Trave', 'ller', ' (', 'for', ' so', ' it']
```

How well do large production vocabularies compress *The Time Machine*? Below, we compare our 1,024-token tokenizer against GPT-2's 50,257-token vocabulary and `o200k_base`, the roughly 200,000-token vocabulary used by GPT-4o.

```
rows = [('ours (1k, in-domain)', len(bpe.encode(raw_text)))]
for name in ('gpt2', 'o200k_base'):
```

```
    e = tiktoken.get_encoding(name)
    rows.append((f'{name} ({len(e._mergeable_ranks) // 1000}k)',
                 len(e.encode(raw_text))))
for name, n in rows:
    print(f'{name:>22}: {n:>6} tokens, {n_bytes / n:.2f} bytes/token')
```

```
 ours (1k, in-domain):  65741 tokens, 2.72 bytes/token
          gpt2 (50k):  44073 tokens, 4.06 bytes/token
   o200k_base (199k):  42624 tokens, 4.20 bytes/token
```

Even though our vocabulary is 50 to 200 times smaller, it achieves roughly two-thirds of their compression ratio because it was trained specifically on this novella. Like any statistical model, a tokenizer is a compression model of its training distribution and performs best on text resembling the corpus it was built from.

8.2.5.3 Fertility, Digits, and Glitch Tokens

The average number of tokens a tokenizer spends per word in a given language is known as its *fertility*. Because GPT-2's tokenizer was trained primarily on English web text, translating the very same sentence into three languages yields starkly different token counts:

```
sentences = {
    'English': 'The time traveller returned after many years.',
    'German': 'Der Zeitreisende kehrte nach vielen Jahren zurück.',
    'Greek': 'Ο ταξιδιώτης του χρόνου επέστρεψε μετά από πολλά χρόνια.'}
for lang, s in sentences.items():
    gpt2, o200k = [len(tiktoken.get_encoding(e).encode(s)) / len(s.split())
                   for e in ('gpt2', 'o200k_base')]
    print(f'{lang:>8}: {gpt2:.2f} (gpt2), {o200k:.2f} (o200k_base) '
          'tokens per word')
```

```
English: 1.14 (gpt2), 1.14 (o200k_base) tokens per word
 German: 2.86 (gpt2), 1.57 (o200k_base) tokens per word
  Greek: 7.33 (gpt2), 1.89 (o200k_base) tokens per word
```

Under GPT-2's tokenizer, a Greek word consumes roughly seven tokens (each Greek letter takes two UTF-8 bytes and very few Greek merges were learned), compared with roughly one token per English word. Consequently, a 1,024-token context window holds about 900 English words but only about 140 Greek words, making inference both slower and more expensive for non-English users. By contrast, the multilingual `o200k_base` vocabulary brings Greek fertility down to roughly two tokens per word.

Numbers reveal how subtle tokenization details reshape what a model must learn. Whereas GPT-2 splits digit sequences wherever BPE happened to discover frequent fragments, newer tokenizers cap digit chunks at three characters during pre-tokenization so that segmentation is more uniform:

```
for name in ('gpt2', 'o200k_base'):
    e = tiktoken.get_encoding(name)
    print(f'{name}: 12345678 ->',
          [e.decode([t]) for t in e.encode('12345678')])
```

```
gpt2: 12345678 -> ['123', '45', '678']
o200k_base: 12345678 -> ['123', '456', '78']
```

Even so, neither segmentation aligns with the columns of positional arithmetic, which groups digits from right to left into ones, tens, and hundreds. Under a left-to-right three-digit cap, `"12345678"` splits into `"123"`, `"456"`, and `"78"`, forcing a language model to perform carries across misaligned boundaries. This mismatch is a well-known contributor to arithmetic errors in language models, and helps explain why inserting standard thousands separators (`"12,345,678"`) can make numerical tasks easier.

Finally, a vocabulary entry is only as reliable as the model's exposure to it during training. Early GPT-2 and GPT-3 vocabularies included tokens such as `" SolidGoldMagikarp"`—a Reddit username that appeared frequently enough in the *tokenizer's* training corpus to earn a dedicated merge, yet was largely absent from the *language model's* pretraining data. Such a *glitch token* can still be produced from user prompts even though its embedding vector received almost no gradient updates; prompted to repeat it, early models often responded with evasions or unrelated words.

```
enc.encode(' SolidGoldMagikarp')  # one token for an 18-character string
```

```
[43453]
```

8.2.6 Summary

A tokenizer bridges raw text and the integer sequences consumed by neural networks. Its vocabulary size governs a core trade-off between sequence length and output-layer size, while fixed word vocabularies additionally break on unseen words. Byte-level BPE begins with the 256 single-byte symbols, eliminating out-of-vocabulary inputs altogether, and greedily merges the most frequent adjacent token pairs until the target vocabulary size is reached. Because encoding replays learned merges strictly in rank order, the merge table alone defines the tokenizer. In practice, a pre-tokenization regex prevents merges from crossing word boundaries, and special control tokens are assigned ids above the BPE range so that ordinary text cannot synthesize them. GPT-2's production tokenizer follows this exact construction at larger scale, and our implementation reproduces it token for token; its varying fertility across languages, its digit-splitting quirks, and its glitch tokens are all direct consequences of its training data and merge rules. In the next section, we turn to modeling the token sequences that a tokenizer produces.

8.2.7 Exercises

1. **Encoding by rank.** Suppose a `BPETokenizer` with `vocab_size=258` has learned exactly two merges: `(b, c)` with id 256, and then `(a, b)` with id 257. The *rank rule* of `_encode_chunk` always applies the lowest-ranked applicable merge. The *leftmost rule* always merges the leftmost adjacent pair that appears in the merge table.
    1. Segment the string "abc" under each of the two rules.
    2. Give the segmentation that every occurrence of "abc" in the training corpus had at the end of training, and say which rule agrees with it.
2. code **Special-token round trip.** Section 8.2.4 places special tokens above the BPE id range so that ordinary text can never produce them. Construct a `BPETokenizer` whose

`specials` add a document separator `<sep>` to the three defaults, and train it on `raw_text`. Let `x` be two short documents joined by `<sep>`.

1. Encode `x` once with and once without `allow_special=True`, and check that both id sequences decode to `x`.
2. Re-encode the decoded text once with `allow_special=True` and once without it. Which of the two id sequences from the previous sub-problem does each re-encoding reproduce? State what this implies for a pipeline that decodes a model's output and later feeds the text back in as input.
3. `tiktoken` raises an error when a special string appears in text that is not allowed to contain it, whereas `encode` treats the string as plain bytes. Give one situation in which each policy is preferable.

Adapted from Andrej Karpathy, minbpe, Step 4.

3. code **Digit chunking.** Tokenize the addition prompts "1234+5678=", "1234+567=" and "567+678=" and their sums "6912", "1801" and "1245" with the `gpt2` and `o200k_base` encodings of `tiktoken`, writing out the token boundaries of each number.

   1. Under the three-digit cap of `o200k_base`, label each digit token with the place values (ones, tens, hundreds, thousands) it covers. For each prompt, state whether the tokens at matching positions of the two summands and of the sum cover the same place values. Explain what goes wrong for a model that adds token by token when they do not.
   2. Propose a pre-tokenization rule that aligns tokens with place value, and state one drawback of your rule.

4. **Glitch tokens.** Section 8.2.5.3 attributes a glitch token to a string that was frequent in the tokenizer's training corpus but rare in the language model's.

   1. Why does the merge rule of `BPETokenizer.train` turn such a string into a single token rather than several fragments, and what role does pre-tokenization play in fixing the token's extent?
   2. Propose a test that flags glitch tokens given the trained embedding table and the model's training corpus, and state what each of the two inputs contributes.

5. code **Pre-tokenization ablation.** ● Split `raw_text` into its first 90% and its last 10%. On the first part, train two 4,096-token tokenizers that differ only in the pattern: one without a pattern, as `big` was trained, and one with `BPETokenizer.GPT2_PATTERN`.

   1. List the 20 longest tokens learned by each tokenizer, and mark those that span more than one word.
   2. Measure which of the two compresses the first part better, and explain the outcome in terms of the merge rule.
   3. Measure bytes per token of both tokenizers on the last 10%. Which tokenizer loses the larger fraction of its compression on this held-out text, and which compresses it better? Relate both answers to the tokens you listed.

8.3 Language Models

A smartphone keyboard suggesting the next word and a speech recognizer deciding whether a speaker said "to recognize speech" or "to wreck a nice beach" both depend on the same

capability: assigning a numerical score that reflects how plausible a passage of text is. Because those two phrases sound nearly identical, acoustics alone cannot tell them apart; what resolves the ambiguity is knowing that the first is idiomatic English while the second is outlandish. A *language model* provides this judgment: given a sequence of tokens $x_1, \ldots, x_T$ drawn from a vocabulary $\mathcal{V}$, such as the BPE tokens constructed in Section 8.2, it estimates the joint probability $P(x_1, \ldots, x_T)$.

At first glance, assigning a probability to every possible sequence appears hopeless: even with our modest 1,024-token vocabulary, there are already $1024^{10} \approx 10^{30}$ sequences of length ten. Fortunately, the chain rule of probability (Section 8.1) converts this intractable joint distribution without approximation into a product of next-token conditionals, each conditioned on the preceding prefix $x_{<t} = x_1, \ldots, x_{t-1}$:

$$P(x_1, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_{<t}). \tag{8.6}$$

A language model is therefore fundamentally a next-token predictor. The same conditional distributions let us both *score* an existing sequence (by multiplying them, or adding their logarithms) and *generate* new text (by sampling x_1, then sampling x_2 given x_1, and continuing step by step). Remarkably, a wide range of tasks can be framed directly as text continuation: completing the prompt "The novella The Time Machine was written by" answers a factual question, which is precisely the paradigm behind modern large language models (Section 11.2).

We begin with the simplest possible estimator for these conditional probabilities, counting occurrences in a corpus, and investigate why it breaks down. To evaluate improvements objectively, we introduce perplexity, compare two classical smoothing methods, and then define bits per byte so that models with different tokenizers can be compared fairly. Finally, we build the minibatch data loader that feeds tokenized sequences into the neural language models of the upcoming sections.

```
import collections
import random
import re
```

8.3.1 n-gram Language Models

8.3.1.1 Markov Models and Counting

Because each factor in Equation 8.6 conditions on the entire prefix $x_{<t}$, the number of distinct contexts grows without bound as t increases. Invoking the Markov condition (Section 8.1.3.1) truncates the context to the $n-1$ most recent tokens:

$$P(x_t \mid x_{<t}) \approx P(x_t \mid x_{t-n+1}, \ldots, x_{t-1}). \tag{8.7}$$

Although this assumption is undeniably crude (a sentence hardly forgets its subject after two words), it reduces the set of possible contexts to a finite number. A model with no context ($n = 1$) is called a *unigram* model, one that conditions on a single preceding token ($n = 2$) a *bigram* model, and one that conditions on two preceding tokens ($n = 3$) a *trigram* model.

Under maximum likelihood estimation, each conditional probability is simply a ratio of corpus counts. Writing $N(\cdot)$ for the number of times a token tuple appears in the training corpus, we obtain

$$\hat{P}(x_t \mid x_{t-n+1}, \ldots, x_{t-1}) = \frac{N(x_{t-n+1}, \ldots, x_t)}{N(x_{t-n+1}, \ldots, x_{t-1})}, \tag{8.8}$$

the empirical fraction of times the context $(x_{t-n+1}, \ldots, x_{t-1})$ was followed by x_t. An n-gram language model is nothing more than a lookup table of these conditional relative frequencies.

Let us estimate unigram, bigram, and trigram tables on *The Time Machine*. While the neural language models later in this chapter operate on BPE tokens, here we tokenize the novella into words so that generated passages are easy to inspect by eye. Reusing the simple lowercase-letter normalization from Section 8.2, we split on whitespace and hold out the final 10% of the text for evaluation. The vocabulary consists of all words occurring in the training split; any held-out word absent from training is mapped to the unknown token `<unk>`.

```
fname = d2l.download(d2l.DATA_URL + 'timemachine.txt',
                     sha1_hash='090b5e7e70c295757f55df93cb0a180b9691891a')
raw_text = open(fname).read()
text = re.sub('[^A-Za-z]+', ' ', raw_text).lower()
split = text.rindex(' ', 0, int(0.9 * len(text)))
words_train = text[:split].split()
known = set(words_train)  # A held-out word never seen in training is <unk>
words_val = [w if w in known else '<unk>' for w in text[split:].split()]
len(words_train), len(words_val)
```

```
(29402, 3373)
```

The `NGramLM` class below stores, for every observed $(n-1)$-token context, a counter of the tokens that followed it in training—everything needed to evaluate Equation 8.8 (the `alpha` attribute will be used shortly when we introduce smoothing). Its `sample` method extends a given prefix by repeatedly drawing the next token in proportion to its count given the most recent $n-1$ tokens.

```
class NGramLM:
    """An n-gram language model estimated from counts."""
    def __init__(self, tokens, n, vocab_size, alpha=0.01):
        self.n, self.vocab_size, self.alpha = n, vocab_size, alpha
        cont = collections.defaultdict(collections.Counter)
        for gram in zip(*(tokens[i:] for i in range(n))):
            cont[gram[:-1]][gram[-1]] += 1
        self.cont = dict(cont)
        self.total = {ctx: sum(c.values()) for ctx, c in self.cont.items()}

    def sample(self, prefix, num_tokens, rng):
        """Extend prefix by sampling from the unsmoothed count model."""
        out = list(prefix)
        for _ in range(num_tokens):
```

```
            ctx = tuple(out[len(out) - self.n + 1:])
            counts = self.cont[ctx]
            out.append(rng.choices(list(counts), counts.values())[0])
        return out

vocab_size = len(known) + 1  # The training words and <unk>
models = {n: NGramLM(words_train, n, vocab_size) for n in (1, 2, 3)}
```

To inspect what these tables actually capture, let us query the trigram model on the context `"the time"`, which appears frequently throughout a novella about time travel.

```
trigram = models[3]
total = trigram.total[('the', 'time')]
for word, count in trigram.cont[('the', 'time')].most_common(3):
    print(f'P({word} | the time) = {count}/{total} = {count / total:.2f}')
single = sum(len(c) == 1 for c in trigram.cont.values()) / len(trigram.cont)
print(f'word pairs followed by a single word in training: {single:.0%}')
```

```
P(traveller | the time) = 47/87 = 0.54
P(machine | the time) = 27/87 = 0.31
P(i | the time) = 6/87 = 0.07
word pairs followed by a single word in training: 84%
```

In the training split, `"the time"` occurs 87 times and is followed by `"traveller"` roughly half the time and by `"machine"` roughly a third of the time—showing that the table readily picks up the central phrases of the novel. Yet frequent contexts like this are the rare exception: for more than four out of five observed word pairs, the training table records only a single continuation.

8.3.1.2 Sampling from n-gram Models

Generating text directly applies Equation 8.6: draw the next token from the conditional distribution given the prefix so far, append it, and repeat. Because fluent-looking output can easily mask verbatim memorization, we pair our samples with a diagnostic function, `longest_copy`, that measures the longest contiguous span of generated words appearing verbatim in the training set.

```
def longest_copy(words, train_words):
    """Length of the longest run of words that also occurs in train_words."""
    corpus, best = ' ' + ' '.join(train_words) + ' ', 0
    for i in range(len(words)):
        while (i + best < len(words) and
               ' ' + ' '.join(words[i:i + best + 1]) + ' ' in corpus):
            best += 1
    return best
```

We seed each model with the prefix `"the time"` and generate 25 words, leaving the random number generator unseeded so that each execution produces fresh samples.

```
rng = random.Random()
for n, model in models.items():
    sample = model.sample(['the', 'time'], 25, rng)
    copied = longest_copy(sample, words_train)
    d2l.print_wrapped(f'{n}-gram (longest copied run: {copied} words): '
                      + ' '.join(sample))
```

```
1-gram (longest copied run: 2 words): the time me and a had i laboratory
    apologize i all seemed potentialities so reached such i assured reached
    little is following of head one however the
2-gram (longest copied run: 3 words): the time and when we began to trouble
    myself but the under world of our friend of flowers brilliant arch of
    the sun belt sandals or other
3-gram (longest copied run: 13 words): the time traveller not a mere block
    but highly decorated with deep framed panels on either hand to pick
    flowers to stick a fork into meat again
```

The unigram model emits words at their corpus frequencies with no grammatical coherence; the bigram model strings together locally plausible pairs that wander aimlessly, since each word depends only on its immediate predecessor; and the trigram sample already sounds remarkably like H. G. Wells.

On a 30,000-word corpus, however, the trigram sounds like Wells largely because much of it *is* Wells. The longest copied run exposes this behavior immediately, rising from two or three words for the unigram, to a handful for the bigram, and typically seven to a dozen consecutive words for the trigram. Whenever the sampler lands on a word pair that occurred with only one successor in training, it is forced onto a single deterministic track, replaying entire clauses of the novella verbatim. Rather than generalizing, the model is stitching together memorized fragments. (When estimated over billions of words, n-gram tables do capture rich local English syntax, which is why they powered speech recognition and statistical machine translation for decades.)

8.3.1.3 Count Sparsity

This memorization is the visible symptom of a deeper statistical bottleneck. The number of *possible* n-grams is $|\mathcal{V}|^n$, growing exponentially with n, whereas the number of n-grams actually present in a corpus grows only linearly with corpus length. We can measure the resulting sparsity directly by checking what fraction of held-out n-grams never appeared even once in training.

```
for n in (1, 2, 3):
    seen = set(zip(*(words_train[i:] for i in range(n))))
    grams = list(zip(*(words_val[i:] for i in range(n))))
    rate = sum(gram not in seen for gram in grams) / len(grams)
    print(f'unseen held-out {n}-grams: {rate:.1%} '
          f'({len(seen)} distinct in training)')
```

```
unseen held-out 1-grams: 8.4% (4315 distinct in training)
unseen held-out 2-grams: 51.5% (18535 distinct in training)
unseen held-out 3-grams: 87.0% (26982 distinct in training)
```

Roughly one in twelve held-out words never appeared in training, half of all held-out word

pairs are unseen, and nearly nine out of ten word triples have count zero. Under the maximum likelihood estimate Equation 8.8, every unseen n-gram receives probability zero—and a single zero factor in Equation 8.6 assigns probability zero to an entire valid English passage.

8.3.1.4 Word Frequency and Zipf's Law

Why do count tables become sparse so rapidly? The underlying cause is the strongly skewed distribution of word frequencies in natural language. Using the `Vocab` class from Section 8.2, which ranks tokens by descending frequency, let us inspect the ten most common words in *The Time Machine*.

```
words = text.split()
vocab = d2l.Vocab(words)
vocab.token_freqs[:10]
```

```
[('the', 2261),
 ('i', 1267),
 ('and', 1245),
 ('of', 1155),
 ('a', 816),
 ('to', 695),
 ('was', 552),
 ('in', 541),
 ('that', 443),
 ('my', 440)]
```

These ten most frequent words reveal nothing about time travel: they are articles, pronouns, and prepositions (often called *stop words*), and almost any Victorian novel would produce a nearly identical list. What is striking is how steeply frequencies drop off: the tenth most common word is already less than one-fifth as frequent as the first. Plotted on log–log axes, frequency versus rank closely follows a straight line beyond the top few ranks, which is the signature of a power law. Specifically, the frequency n_i of the i^{th} most common word obeys *Zipf's law*:

$$n_i \propto \frac{1}{i^s}, \quad \text{equivalently} \quad \log n_i = -s \log i + c, \tag{8.9}$$

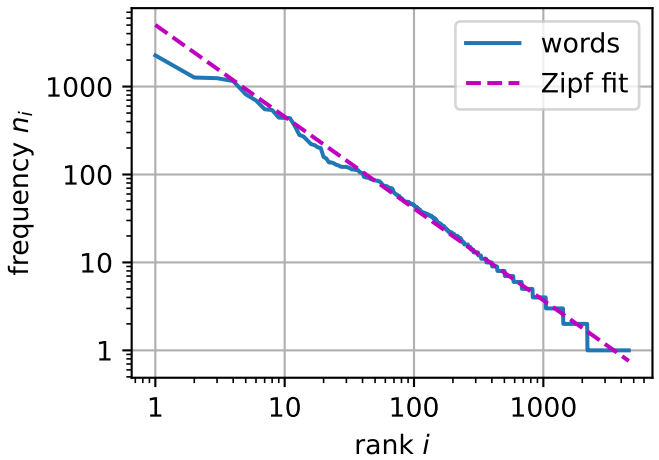
for an exponent s and constant c. The helper `zipf_fit` below estimates s and c by fitting a straight line in log–log space over ranks 10 through 1,000.

```
def zipf_fit(freqs, lo=10, hi=1000):
    """Fit log n_i = -s log i + c over ranks lo..hi; return (s, c)."""
    ranks = np.arange(lo, hi + 1)
    slope, c = np.polyfit(np.log(ranks), np.log(freqs[lo - 1:hi]), 1)
    return -slope, c
```

Applying `zipf_fit` to the word frequencies `freqs` from `vocab.token_freqs`, we print the fitted exponent s and overlay the power-law line on the empirical counts.

```
freqs = [freq for token, freq in vocab.token_freqs]
s, c = zipf_fit(freqs)
print(f'Zipf exponent s = {s:.2f}')
ranks = np.arange(1, len(freqs) + 1)
d2l.plot(ranks, [freqs, np.exp(c) * ranks ** -s], xlabel='rank $i$',
         ylabel='frequency $n_i$', xscale='log', yscale='log',
         legend=['words', 'Zipf fit'])
```

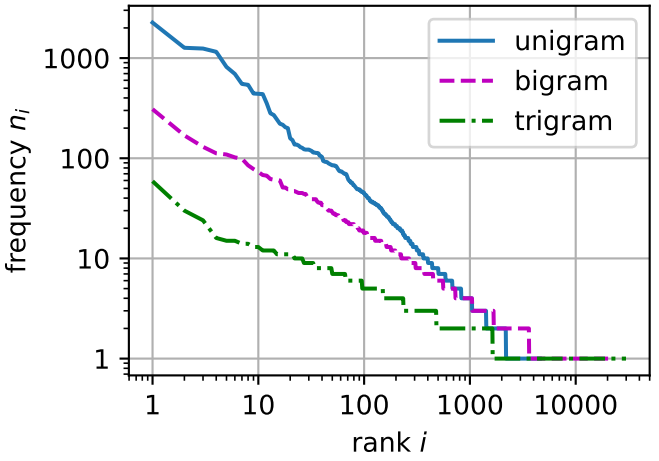
```
Zipf exponent s = 1.04
```



The fitted exponent sits close to 1, matching Zipf's classic observation across natural languages. Power-law distributions are heavy-tailed: although any individual rare word appears infrequently, there are so many distinct rare words that together they account for a substantial share of any text. To see whether bigrams and trigrams behave similarly, we compute their frequency curves, fit their Zipf exponents, measure the fraction of distinct n-grams that appear only once, and plot all three curves together.

```
counts = [sorted(collections.Counter(zip(*(words[i:] for i in range(n))))
                 .values(), reverse=True) for n in (1, 2, 3)]
for n, f in zip((1, 2, 3), counts):
    print(f'{n}-grams: s = {zipf_fit(f)[0]:.2f}, '
          f'{np.mean(np.array(f) == 1):.0%} of distinct {n}-grams occur once')
d2l.plot([np.arange(1, len(f) + 1) for f in counts], counts,
         xlabel='rank $i$', ylabel='frequency $n_i$', xscale='log',
         yscale='log', legend=['unigram', 'bigram', 'trigram'])
```

```
1-grams: s = 1.04, 52% of distinct 1-grams occur once
2-grams: s = 0.69, 82% of distinct 2-grams occur once
3-grams: s = 0.47, 95% of distinct 3-grams occur once
```



Word pairs and triples also follow power laws, albeit with smaller exponents (roughly 0.7 and 0.5). Crucially for count-based models, the tail grows dramatically heavier as n increases: while half of all distinct words occur only once, more than nine in ten distinct trigrams are singletons whose probability estimates rest on a lone observation. It is this heavy tail that produces the deluge of unseen trigrams on held-out text: any model that estimates each context independently requires exponentially more data as the context lengthens.

8.3.2 Perplexity and Smoothing

8.3.2.1 Perplexity

Inspection of the count tables revealed severe sparsity that our sample generations had concealed. Before attempting to repair these tables, we need a principled quantitative metric so we can tell whether a modification truly improves generalization. Evaluating the joint probability of held-out text is an obvious starting point, but raw sequence probabilities shrink exponentially with document length: every model assigns a vastly smaller probability to Tolstoy's *War and Peace* than to Saint-Exupéry's *The Little Prince*, regardless of model quality. What we need is a per-token average.

Information theory gives us exactly that. In Section 3.1.2.3, we saw that the cross-entropy loss of a probabilistic classifier equals the *surprisal* of the true label—the negative logarithm of the probability assigned to it. Since a language model is a next-token classifier at every position, we evaluate it by its average surprisal across all T tokens of the sequence:

$$\frac{1}{T}\sum_{t=1}^{T} -\log P(x_t \mid x_{<t}), \tag{8.10}$$

where P is the model's predictive distribution and x_t is the actual token at step t. Because this average normalizes by length, documents of different lengths become comparable. In natural language processing, it is customary to report the exponential of Equation 8.10, known as *perplexity* (Jelinek et al. 1977):

$$\text{PPL} = \exp\left(-\frac{1}{T}\sum_{t=1}^{T}\log P(x_t \mid x_{<t})\right). \tag{8.11}$$

Equivalently, perplexity is the geometric mean of the inverse probability assigned to each observed token. A model that assigns probability $1/4$ to every target token achieves a perplexity of 4, matching the uncertainty of a uniform choice among four equally likely candidates. Perplexity is therefore naturally interpreted as the model's *effective branching factor* per token. Three limiting cases anchor the scale:

- A model that always predicts the true token with probability 1 achieves the minimum possible perplexity of 1.
- A model that assigns probability 0 to even a single observed token incurs infinite perplexity.
- A baseline model predicting the uniform distribution over the vocabulary achieves perplexity $|\mathcal{V}|$, setting an upper bound that any useful model must beat.

Our unsmoothed maximum likelihood n-grams immediately run into the second case: because they assign zero probability to every unseen n-gram, a single novel trigram in the held-out text sends perplexity to infinity. To evaluate and compare count models meaningfully, we must first *smooth* the counts to eliminate zeros.

8.3.2.2 Laplace Smoothing

The classical remedy is *Laplace smoothing* (also called add-α smoothing), which adds a positive constant $\alpha > 0$ to every count and $\alpha|\mathcal{V}|$ to the denominator so that the resulting distribution remains normalized:

$$\hat{P}(x_t \mid x_{t-n+1}, \ldots, x_{t-1}) = \frac{N(x_{t-n+1}, \ldots, x_t) + \alpha}{N(x_{t-n+1}, \ldots, x_{t-1}) + \alpha|\mathcal{V}|}. \tag{8.12}$$

Every unseen n-gram now receives a small positive probability, and any completely unseen context defaults to the uniform distribution $1/|\mathcal{V}|$. The idea predates language modeling by two centuries: Laplace introduced it with $\alpha = 1$ to estimate the probability that the sun will rise tomorrow given that it has risen every day so far.

Let us first add a Laplace-smoothed `prob` method to `NGramLM` that implements Equation 8.12 using the model's stored smoothing constant `alpha`.

```
@d2l.add_to_class(NGramLM)
def prob(self, context, token):
    """Laplace-smoothed P(token | last n-1 tokens of context)."""
    ctx = tuple(context[len(context) - self.n + 1:])
    num = self.cont.get(ctx, {}).get(token, 0) + self.alpha
    return num / (self.total.get(ctx, 0) + self.alpha * self.vocab_size)
```

With `prob` in place, `eval_nll` sums the negative log-likelihood across a token sequence so we can compute held-out perplexity on `words_val` for each n-gram order.

```
def eval_nll(model, tokens):
    """Total negative log-likelihood of tokens, in nats."""
    return -sum(math.log(model.prob(tokens[max(0, t - model.n + 1):t],
                                    tokens[t]))
                for t in range(len(tokens)))

for n, model in models.items():
    ppl = math.exp(eval_nll(model, words_val) / len(words_val))
    print(f'{n}-gram held-out perplexity, Laplace smoothing: {ppl:7.1f}')
```

```
1-gram held-out perplexity, Laplace smoothing:   840.6
2-gram held-out perplexity, Laplace smoothing:   753.6
3-gram held-out perplexity, Laplace smoothing:  2481.4
```

Under Laplace smoothing, the trigram model, whose generated samples looked most convincing, actually suffers the *worst* held-out perplexity by a wide margin, while the bigram barely edges past the unigram. Count sparsity explains both outcomes: when 87% of held-out trigrams are unseen, the vast majority of trigram predictions collapse into nearly uniform smoothed guesses. Nor can tuning α save it: even the optimal α from exercise 2 leaves trigram perplexity above 2,400. Fluent samples turned out to be a misleading guide to generalization.

These numbers expose the two core flaws of Laplace smoothing. First, it distributes probability mass equally across all unseen successors, ignoring how plausible each word is in general. Second, it fails to fall back on shorter contexts: an unseen trigram receives the same flat estimate regardless of whether its final two words form a frequent bigram or an implausible pair.

8.3.2.3 Kneser–Ney Smoothing

Kneser–Ney smoothing (Kneser and Ney 1995), in the interpolated formulation of Chen and Goodman (1999), addresses both flaws and reigned as the strongest count-based language model for two decades. Denote by u a context of $n-1$ tokens, by w the next token, and by u' the shorter context obtained by dropping the oldest token of u. The method combines three interlocking ideas.

The first is *absolute discounting*. Empirical counts systematically overestimate the true frequencies of rare events: an n-gram observed once in training will, on average, appear fewer than once in a fresh sample of the same size because of selection bias among the huge pool of rare n-grams. Subtracting a fixed discount $0 < D < 1$ from every nonzero count corrects this upward bias, most strongly for small counts where the distortion is largest, and frees a total probability mass of $D\,S(u)/N(u)$ in context u, where $S(u)$ is the number of distinct token types observed immediately after u in training.

The second idea is *interpolation*: rather than spreading the discounted mass uniformly, we allocate it to the next-lower-order estimate $P_{\text{KN}}(w \mid u')$, which is defined recursively in the exact same way down to the uniform floor $1/|\mathcal{V}|$:

$$P_{\text{KN}}(w \mid u) = \frac{\max(N(u,w) - D, 0)}{N(u)} + \frac{D\,S(u)}{N(u)}\,P_{\text{KN}}(w \mid u'). \tag{8.13}$$

Under this recursion, a frequently seen context relies primarily on its own empirical counts,

whereas a context observed with many diverse successors shifts more weight to the shorter context—reflecting the intuition that a context with many past continuations is also more likely to admit novel ones. If a context u was never observed in training at all, all probability mass passes directly down: $P_{\text{KN}}(w \mid u) = P_{\text{KN}}(w \mid u')$.

The third idea changes *what* the lower orders count. Because the lower-order distribution is invoked primarily to score words that have not appeared in the longer context, it should measure how readily a word enters *new* contexts rather than its raw corpus frequency. In our training text, `"traveller"` appears 49 times, yet follows only two distinct words (`"time"` 48 times and `"real"` once), whereas `"said"` appears 76 times after 48 different predecessors. Raw unigram frequency would mistakenly predict `"traveller"` almost as eagerly as `"said"` in an unfamiliar context, whereas counting distinct predecessors recognizes that `"traveller"` rarely appears without `"time"`. (The classic NLP example is `"Francisco"`, which is common in news text yet almost exclusively preceded by `"San"`.) Kneser–Ney therefore replaces $N(u, w)$ at every order below the highest with the *continuation count*, the number of distinct tokens that precede (u, w) in training, with $N(u)$ and $S(u)$ computed analogously from those continuation counts.

Better still, the discount parameter D has a clean closed-form estimate derived by Ney et al. (1994): $D = c_1/(c_1 + 2c_2)$ at each order, where c_1 and c_2 are the numbers of distinct n-grams at that order occurring exactly once and twice, respectively (using continuation counts below the highest order).

The function `kneser_ney` below implements Equation 8.13 in full: it tallies the highest-order n-grams, derives continuation counts for every lower order from the distinct tuples one order above, computes $N(u)$, $S(u)$, and D, and evaluates the recursion across a token sequence, returning the log-probability (in nats) of each token given its available prefix.

```
def kneser_ney(train, tokens, vocab_size, n=3):
    """Interpolated Kneser-Ney log-probabilities (nats) of each of tokens."""
    counts = {n: collections.Counter(zip(*(train[i:] for i in range(n))))}
    for k in range(n - 1, 0, -1):  # Lower orders: continuation counts
        counts[k] = collections.Counter(g[1:] for g in counts[k + 1])
    stats = {}
    for k, c in counts.items():
        c1, c2 = (sum(x == r for x in c.values()) for r in (1, 2))
        N, S = collections.Counter(), collections.Counter()
        for g, x in c.items():
            N[g[:-1]] += x
            S[g[:-1]] += 1
        stats[k] = c, N, S, c1 / max(c1 + 2 * c2, 1)
    logp = []
    for i in range(len(tokens)):
        p = 1 / vocab_size  # Below the unigram: uniform
        for k in range(1, min(n, i + 1) + 1):
            c, N, S, D = stats[k]
            g = tuple(tokens[i - k + 1:i + 1])
            if N[g[:-1]]:  # An unseen context passes p down unchanged
                p = (max(c[g] - D, 0) + D * S[g[:-1]] * p) / N[g[:-1]]
        logp.append(math.log(p))
    return logp
```

Before scoring the full validation split, let us watch Kneser–Ney at work on our familiar con-

text `"the time"`. Below, we print the trigram discount D and compare the probabilities that Kneser–Ney and Laplace smoothing assign to three candidate continuations: `"traveller"`, which followed `"the time"` in 47 of its 87 training occurrences, and two words that never followed it in training, `"was"` and `"filby"`.

```
c3 = collections.Counter(zip(words_train, words_train[1:], words_train[2:]))
c1, c2 = (sum(x == r for x in c3.values()) for r in (1, 2))
print(f'trigram discount D = {c1 / (c1 + 2 * c2):.2f}')
for w in ('traveller', 'was', 'filby'):
    lp = kneser_ney(words_train, ['the', 'time', w], vocab_size)[-1]
    print(f'P({w} | the time): Kneser-Ney {math.exp(lp):.2g}, '
          f'Laplace {models[3].prob(["the", "time"], w):.2g}')
```

```
trigram discount D = 0.93
P(traveller | the time): Kneser-Ney 0.53, Laplace 0.36
P(was | the time): Kneser-Ney 0.00037, Laplace 7.7e-05
P(filby | the time): Kneser-Ney 2.3e-05, Laplace 7.7e-05
```

The trigram discount is roughly 0.9, subtracting nearly an entire count from every observed trigram—a vivid reflection of how noisy single-count observations are. For a common trigram with count 47, subtracting 0.9 barely moves the needle, keeping `"traveller"` close to its empirical frequency of $47/87$, whereas Laplace smoothing squanders nearly a third of that probability mass across the vocabulary. The contrast on unseen continuations is even sharper: Laplace smoothing treats `"was"` and `"filby"` identically, whereas Kneser–Ney recognizes that `"was"` follows a wide variety of contexts while `"filby"` appears in very few, separating their probabilities by more than an order of magnitude. Let us measure how much this gains on the held-out text.

```
for n in (1, 2, 3):
    kn = kneser_ney(words_train, words_val, vocab_size, n)
    print(f'{n}-gram held-out perplexity, Kneser-Ney: '
          f'{math.exp(-sum(kn) / len(kn)):7.1f}')
```

```
1-gram held-out perplexity, Kneser-Ney:   592.8
2-gram held-out perplexity, Kneser-Ney:   312.3
3-gram held-out perplexity, Kneser-Ney:   300.1
```

Kneser–Ney completely reverses the ranking. Even its unigram beats Laplace's, its bigram cuts perplexity roughly in half, and its trigram outperforms all other count models because unseen trigrams back off smoothly to well-calibrated bigram and unigram continuation estimates rather than collapsing to a uniform guess. Notice, though, that the trigram improves only modestly over the bigram: because most held-out trigrams remain unseen, the trigram model falls back to the bigram for the majority of positions. Achieving low perplexity on *unseen* text is the defining test of a language model, and perplexity will serve as our primary benchmark for the neural models that follow.

Still, Kneser–Ney shares a fundamental limitation with every count-based method: while smoothing redistributes probability mass away from zeros, it cannot infer which unseen combinations are semantically plausible. Because a count table treats each token as an atomic, unrelated symbol, it has no way of knowing that `"cat"` and `"feline"` share syntactic and semantic roles, or that `"I want to eat, grandma"` is benign whereas `"I want to eat grandma"` is alarming. Sharing statistical strength across related words and contexts demands learned

continuous representations—the exact motivation for turning to neural networks in Section 8.4.

8.3.2.4 Bits per Byte

Perplexity suffers from one crucial limitation: it is tied to the choice of tokenization. Because a character model chooses among 27 symbols at each step while a word model chooses among thousands, their raw perplexities are on completely different scales and cannot be compared directly. Yet tokenization is an internal design choice (Section 8.2), and a fair evaluation metric must be invariant to it. Fortunately, the remedy is simple: replace the denominator. Instead of dividing total surprisal by the number of tokens, we divide by the number of *bytes* in the evaluated text, a quantity determined solely by the text itself. Measuring surprisal in bits (base-2 logarithms) gives *bits per byte* (bpb):

$$\text{bpb} = \frac{1}{B}\sum_{t=1}^{T} -\log_2 P(x_t \mid x_{<t}), \tag{8.14}$$

where B is the length of the evaluated text in UTF-8 bytes. Bits per byte has an immediate operational meaning as a lossless compression rate: paired with an arithmetic coder, a model achieving bpb bits per byte compresses the text to $\text{bpb}/8$ of its original byte size.

To compare perplexity and bits per byte directly, we fit trigram models, first with Laplace smoothing and then with Kneser–Ney, on the same normalized training text under three tokenizations: characters, words, and a 1,024-token BPE vocabulary (trained on the normalized training split using the tokenizer from Section 8.2). One subtlety is required to keep the comparison honest: whenever the word model predicts <unk>, it has not yet specified *which* out-of-vocabulary word occurred. To encode the held-out text losslessly, the word model must still spell each unknown word out, so we add $\log_2 27$ bits per letter plus the trailing space—the cost of coding each character uniformly from the 27 normalized symbols.

```
bpe = d2l.BPETokenizer(vocab_size=1024, pattern=d2l.BPETokenizer.GPT2_PATTERN)
bpe.train(text[:split])
streams = {'character': (list(text[:split]), list(text[split:]),
                         len(set(text))),
           'BPE (1,024)': (bpe.encode(text[:split]), bpe.encode(text[split:]),
                           bpe.vocab_size),
           'word': (words_train, words_val, vocab_size)}
n_bytes = len(text[split:].encode('utf-8'))
spell = sum((len(w) + 1) * math.log(27)  # Nats to spell out the <unk> words
            for w in text[split:].split() if w not in known)
print(f'{"tokenization":>12} {"|V|":>5} {"tokens":>7} {"ppl":>7} '
      f'{"bpb":>5} {"KN bpb":>6}')
for name, (train, val, m) in streams.items():
    nll = eval_nll(NGramLM(train, 3, m), val)
    nll_kn = -sum(kneser_ney(train, val, m))
    extra = spell if name == 'word' else 0
    print(f'{name:>12} {m:>5} {len(val):>7} {math.exp(nll / len(val)):>7.1f} '
          f'{(nll + extra) / math.log(2) / n_bytes:>5.2f} '
          f'{(nll_kn + extra) / math.log(2) / n_bytes:>6.2f}')
```

tokenization	|V|	tokens	ppl	bpb	KN bpb
character	27	17345	6.4	2.68	2.67
BPE (1,024)	1024	5509	281.6	2.58	1.90
word	4316	3373	2481.4	2.81	2.22

Measured by raw perplexity, the three Laplace-smoothed models appear separated by more than two orders of magnitude; measured by bits per byte, they lie within a third of a bit of one another, and the character model, seemingly far ahead in perplexity, no longer wins. For a fixed Markov order n, coarser tokens span longer stretches of text: two words of context reach back roughly ten characters, whereas two characters scarcely cover a syllable. The word model would lead were it not penalized by out-of-vocabulary words, whose spelling cost adds roughly 0.6 bits per byte and drops it to last place under Laplace smoothing. Byte-level BPE suffers no OOV penalty at all, achieving the strongest compression and providing yet another reason modern language models adopt subword tokenization.

In the final column, Kneser–Ney trims more than half a bit per byte from both the BPE and word trigrams while leaving the character trigram virtually unchanged, since with only 27 symbols very few character trigrams are unseen and smoothing has little work to do. At roughly 1.9 bits per byte, the Kneser–Ney BPE trigram is our strongest count-based baseline.

Bits per byte is therefore the standard metric when comparing modern language models built on different vocabularies (for instance, a 50,000-token model against a 200,000-token model). Three benchmarks help calibrate the scale: Shannon's classic human-prediction experiments estimated the entropy of printed English at roughly 1 bit per character (Shannon 1951); modern neural language models compress diverse web text to well below 1 bit per byte; and the count-based models of this section sit between roughly 1.9 and 2.8 bits per byte.

8.3.3 Partitioning Sequences

The neural language models in the upcoming sections ingest minibatches of fixed-length BPE token subsequences produced by the tokenizer from Section 8.2. Let us assemble that data pipeline now.

Picture the tokenized corpus as a single contiguous stream of token ids $x_1, x_2, x_3, \ldots$. An input window of length T starting at index t contains $x_t, \ldots, x_{t+T-1}$, and its paired target is the same window shifted one token to the right, $x_{t+1}, \ldots, x_{t+T}$. As the model steps through the input window, it predicts the next token at every position, so each window supplies T supervised next-token targets. Every starting offset in the corpus except the final T tokens defines a valid window, with adjacent windows overlapping in all but one token (Figure 8.6). During training, we sample minibatches of these windows at random to provide the shuffling that stochastic gradient descent requires.

The `TimeMachine` constructor below puts the pipeline together: it downloads the novella, trains the BPE tokenizer from Section 8.2, encodes the text into integer ids, and stacks all overlapping windows of length `num_steps` into aligned input (`X`) and target (`Y`) arrays. The trained tokenizer is stored on both `data.tokenizer` and `data.vocab` so that we can decode ids back to text and query `len(data.vocab)` uniformly across tokenization modes.

```
@d2l.add_to_class(d2l.TimeMachine)
def __init__(self, batch_size, num_steps, num_train=50000, num_val=5000,
             tokenization='bpe', vocab_size=1024):
```

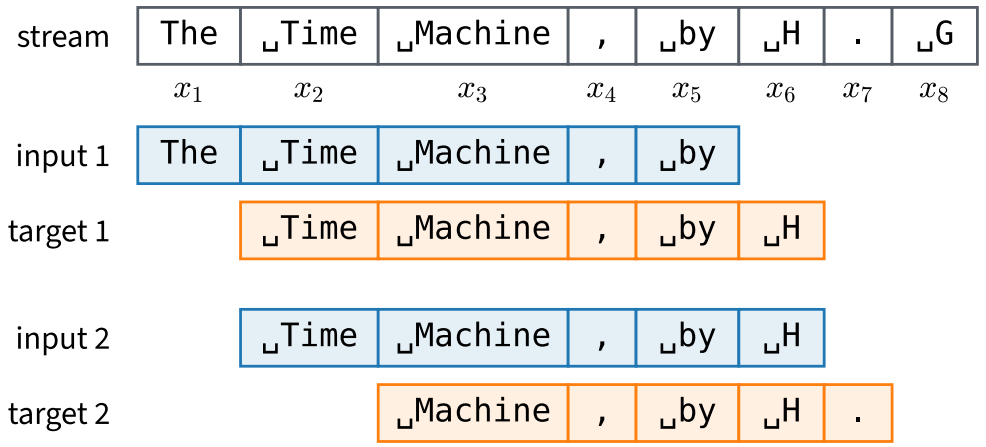


Figure 8.6: The first two input windows of length $T = 5$ over the opening BPE tokens of *The Time Machine* (␣ marks a leading space). Each target is its input shifted forward by one token, and consecutive windows overlap in all but one token; a minibatch gathers windows drawn at random.

```
        super(d2l.TimeMachine, self).__init__()
        self.save_hyperparameters()
        raw_text = self._download()
        if tokenization == 'bpe':
            self.tokenizer = d2l.BPETokenizer(
                vocab_size, pattern=d2l.BPETokenizer.GPT2_PATTERN)
            self.tokenizer.train(raw_text)
            corpus, self.vocab = self.tokenizer.encode(raw_text), self.tokenizer
        else:  # tokenization='char': characters, via TimeMachine.build
            corpus, self.vocab = self.build(raw_text)
        windows = [corpus[i:i+num_steps+1] for i in range(len(corpus)-num_steps)]
        # Drop the windows that straddle the split, so that no validation token
        # is a training target
        windows = windows[:num_train] + windows[num_train+num_steps:]
        array = d2l.tensor(windows)
        self.X, self.Y = array[:,:-1], array[:,1:]
```

We reserve the first `num_train` windows for training and `num_val` subsequent windows for validation. Because adjacent windows overlap heavily, the windows immediately following the training split would contain tokens that already appeared as training targets; dropping a buffer of `num_steps` boundary windows prevents any data leakage between training and validation. During validation, minibatches are iterated sequentially without shuffling.

```
@d2l.add_to_class(d2l.TimeMachine)
def get_dataloader(self, train):
    idx = slice(0, self.num_train) if train else slice(
        self.num_train, self.num_train + self.num_val)
    return self.get_tensorloader([self.X, self.Y], train, idx)
```

Each minibatch consists of two integer tensors of shape (`batch_size`, `num_steps`). Decoding a sample row confirms that `X` contains Wells' prose and that `Y` is shifted forward by exactly one token.

```
data = d2l.TimeMachine(batch_size=2, num_steps=10)
for X, Y in data.train_dataloader():
    print('X:', X, '\nY:', Y)
    break
print('X[0]:', repr(data.tokenizer.decode([int(i) for i in X[0]])))
print('Y[0]:', repr(data.tokenizer.decode([int(i) for i in Y[0]])))
```

```
X: tensor([[ 302,  115,  280,  408,  277,  272,  360,  270,  432,  260],
        [ 353,  258, 1012,  480,  342,   46,  533,  260,   10,   84]])
Y: tensor([[ 115,  280,  408,  277,  272,  360,  270,  432,  260,  335],
        [ 258, 1012,  480,  342,   46,  533,  260,   10,   84,  355]])
X[0]: 'sts cast and filed into the'
Y[0]: 's cast and filed into the re'
```

A window of BPE tokens supplies far more context than any count table can exploit—our word tables already ran out of data at just two words of history. In the next section, we replace lookup tables with a neural network that compresses an arbitrarily long prefix into a learned hidden state.

8.3.4 Summary

A language model assigns probabilities to token sequences, and the chain rule of probability reduces this task to next-token prediction, enabling the same model to both score and generate text. Under a Markov assumption, n-gram models estimate next-token probabilities directly from corpus counts; although simple to construct and capable of locally fluent samples, they suffer from exponentially growing context spaces and Zipfian heavy tails, leaving most observed n-grams as singletons and causing the models to memorize rather than generalize. Perplexity, the exponential of average per-token cross-entropy, measures a model's effective branching factor and exposes why Laplace-smoothed trigrams perform poorly on held-out text. Interpolated Kneser–Ney smoothing discounts observed counts and redistributes the freed mass to lower-order continuation counts, yielding the strongest count-based models, yet it still cannot generalize across semantically related tokens. Because perplexity depends on vocabulary granularity, bits per byte provides a tokenization-invariant compression rate for fair cross-model comparisons. Finally, to feed neural language models in the sections ahead, we convert the BPE-tokenized corpus into overlapping fixed-length input–target windows with a clean gap preventing leakage into the validation split.

8.3.5 Exercises

1. **Dense versus sparse storage.** Consider a 5-gram model over a vocabulary of $|\mathcal{V}| = 50{,}000$ word tokens.
    1. How many entries does a dense table of the conditional probabilities Equation 8.8 contain?
    2. Suppose instead that only the 5-grams observed in a trillion-token corpus are stored, at 16 bytes per entry. How much memory is needed at most, and what

fraction of all possible 5-grams is covered?
3. The word vocabulary of Section 8.3.1 has $|\mathcal{V}| = 4{,}316$ entries, and the code there prints the number of distinct word trigrams in the training text. Which fraction of all possible trigrams occurs in training? Which fraction of the held-out trigrams does? Is the coverage computed in (b) therefore a good estimate of how often a query on new text finds its 5-gram in the table?

2. code **Smoothing constant.** Sweep α in Equation 8.12 over the 19 values $\alpha = 10^{k/4}$ for $k = -16, -15, \dots, 2$, for the unigram, bigram, and trigram word models, and plot held-out perplexity against α on a logarithmic axis.
   1. How does the best α change with n, and why?
   2. Rank the three models, each at its own best α, by held-out perplexity. How does this ranking differ from the one printed in Section 8.3.2 with the default $\alpha = 0.01$ of `NGramLM`?

3. **Perplexity under skew.** A training corpus over ten symbols consists of one symbol 91% of the time and each of the other nine 1% of the time.
   1. Compute the perplexity of the unigram model estimated from this corpus on a long test text drawn from the same distribution. How does it compare with the vocabulary size of ten?
   2. The word-level unigram model of Section 8.3.2 chooses among thousands of word types. Read off its held-out perplexity as an effective number of equally likely choices, and explain with Zipf's law Equation 8.9 why this number is far below $|\mathcal{V}|$.

   Adapted from Daniel Jurafsky and James H. Martin, Speech and Language Processing, 3rd ed. draft, Exercise 3.12.

4. **Overlapping windows.** `d2l.TimeMachine` builds one row of `num_steps + 1` consecutive tokens for every possible start position in the corpus. The first `num_steps` tokens of a row form a row of `X`, and the last `num_steps` tokens form the matching row of `Y`. The first `num_train` rows go to training, the next `num_steps` rows are dropped, and the `num_val` rows after them go to validation (Section 8.3.3). Take `num_steps=32`, `num_train=10000`, and `num_val=5000`.
   1. Counting input and target tokens together, which corpus positions do the training rows touch, and which do the validation rows touch? How many positions would both touch if no rows were dropped, and why would that inflate the validation score?
   2. How many times per epoch does a token in the interior of the training range serve as a target?
   3. How many windows does a partition of the entire corpus of about 66,000 tokens into non-overlapping 32-token windows yield? Compare this partition with the 10,000 overlapping training rows in the number of windows and in the number of distinct corpus positions covered.

5. code **Normalization across lengths.** ● `NGramLM` has no end-of-sequence token. Train a bigram `NGramLM` with $\alpha = 1$ and `vocab_size=3` on a short string over a three-symbol vocabulary. Use `prob` and the chain rule Equation 8.6 to sum the probabilities of all sequences of length 2, and separately those of all sequences of length 3.
   1. What does each sum equal, and what do the two sums say about the model as a

distribution over sequences whose length is not fixed in advance?

2. Add an end-of-sequence token to the vocabulary, so that `vocab_size=4`, and append it to the training string. Treat a sequence as complete once it emits this token. Show that the probabilities of all complete sequences now sum to exactly one. Which property of the model rules out probability mass on sequences that never end?

Adapted from Daniel Jurafsky and James H. Martin, Speech and Language Processing, 3rd ed. draft, Exercise 3.5.

6. code **Seen and unseen trigrams.** ● Split the held-out positions from the third word on by whether the word trigram ending at that position occurred in training. Take the bigram and the trigram word models, each at its best α from the sweep of problem 2, and compute the average negative log-likelihood of each model on each group. On which group is the trigram the better model, and how does Kneser–Ney Equation 8.13 exploit this?

8.4 Recurrent Neural Networks

The count-based language models of Section 8.3 predict each token from a short window of immediate predecessors. A trigram model conditions only on the two preceding tokens; anything earlier in the document has zero influence on its prediction. Simply widening the window quickly hits a wall: a count table spanning 20 BPE tokens would have to track $1024^{20} \approx 10^{60}$ contexts, and even a neural network reading a fixed window of m concatenated token embeddings (Bengio et al. 2003) remains blind to any token $m+1$ steps back while growing linearly in parameter count with m.

In Section 8.1, we introduced a fundamentally different approach: a *latent* autoregressive model that compresses the entire history into a fixed-size *hidden state* vector $\mathbf{h}_{t-1}$ and predicts the next token from that summary,

$$P(x_t \mid x_{<t}) \approx P(x_t \mid \mathbf{h}_{t-1}), \qquad \mathbf{h}_t = g(\mathbf{h}_{t-1}, x_t). \tag{8.15}$$

Under this recurrence, the computational cost per step stays constant no matter how long the prefix becomes. A *recurrent neural network* (RNN) parameterizes the transition function g with a neural network layer whose weights are shared across all time steps (Elman 1990). (Note that a *hidden state* carried across time steps differs from a feedforward *hidden layer* sitting between input and output within a single step.) In this section, we formulate the RNN language model, tally its parameters and compute, and implement it both from scratch and with high-level framework layers on *The Time Machine*.

8.4.1 Recurrent Neural Networks with Hidden States

An RNN is easiest to understand by starting from a single-hidden-layer MLP (Section 4.1) and adding one recurrent connection. Given a minibatch of b examples with d features each, $\mathbf{X} \in \mathbb{R}^{b \times d}$ (we use b for batch size throughout this chapter because n denotes n-gram order), an ordinary hidden layer with activation function ϕ computes

$$\mathbf{H} = \phi(\mathbf{X}\mathbf{W}_{\text{xh}} + \mathbf{b}_{\text{h}}), \tag{8.16}$$

with $\mathbf{W}_{\text{xh}} \in \mathbb{R}^{d \times h}$ and $\mathbf{b}_{\text{h}} \in \mathbb{R}^{1 \times h}$, followed by an output projection $\mathbf{O} = \mathbf{H}\mathbf{W}_{\text{hq}} + \mathbf{b}_{\text{q}}$ into q logits.

Now suppose we receive a sequence of minibatch inputs $\mathbf{X}_t \in \mathbb{R}^{b \times d}$ indexed by time step t. To let the hidden representation remember earlier steps, we feed the previous hidden state $\mathbf{H}_{t-1} \in \mathbb{R}^{b \times h}$ back into the layer via a recurrent weight matrix $\mathbf{W}_{\text{hh}} \in \mathbb{R}^{h \times h}$:

$$\begin{aligned} \mathbf{H}_t &= \phi(\mathbf{X}_t \mathbf{W}_{\text{xh}} + \mathbf{H}_{t-1} \mathbf{W}_{\text{hh}} + \mathbf{b}_{\text{h}}), \\ \mathbf{O}_t &= \mathbf{H}_t \mathbf{W}_{\text{hq}} + \mathbf{b}_{\text{q}}. \end{aligned} \tag{8.17}$$

Because the same five parameter tensors ($\mathbf{W}_{\text{xh}}$, $\mathbf{W}_{\text{hh}}$, $\mathbf{b}_{\text{h}}$ in the recurrent layer and $\mathbf{W}_{\text{hq}}$, $\mathbf{b}_{\text{q}}$ in the output layer) are reused at every step (starting from $\mathbf{H}_0 = \mathbf{0}$), the parameter count is completely independent of sequence length (Figure 8.7). Viewed in unrolled form, an RNN running over T steps is a T-layer feedforward network whose layers all share the same weights.

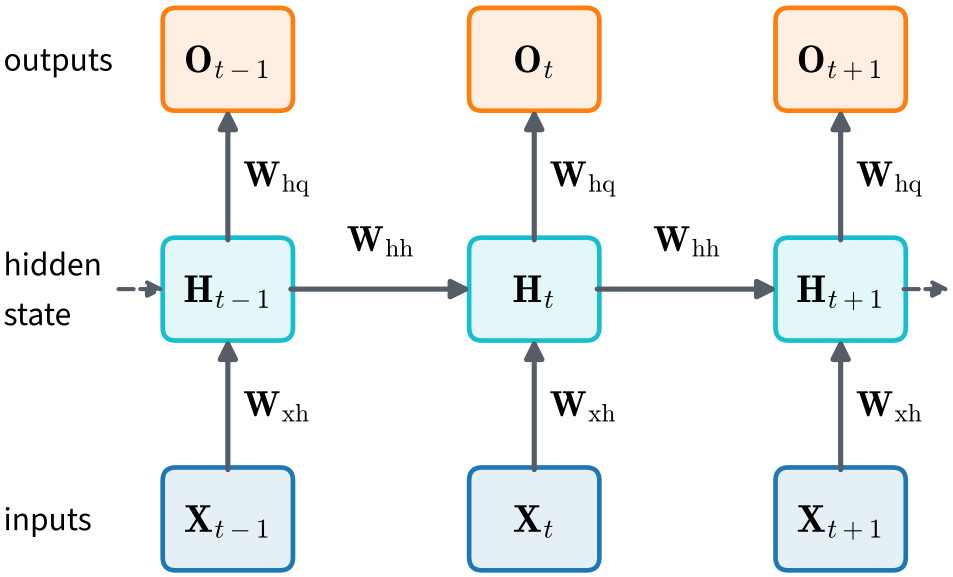


Figure 8.7: An RNN with a hidden state, unrolled across three adjacent time steps. The same weights are reused at every step.

Notice that the two matrix products in Equation 8.17 equal a single product of the horizontally concatenated inputs with the vertically stacked weights:

$$[\mathbf{X}_t, \mathbf{H}_{t-1}] \begin{bmatrix} \mathbf{W}_{\text{xh}} \\ \mathbf{W}_{\text{hh}} \end{bmatrix} = \mathbf{X}_t \mathbf{W}_{\text{xh}} + \mathbf{H}_{t-1} \mathbf{W}_{\text{hh}}.$$

Libraries nevertheless keep the two products separate because during training all inputs $\mathbf{X}_1, \ldots, \mathbf{X}_T$ are known up front, allowing $\mathbf{X}_t \mathbf{W}_{\text{xh}}$ across all T steps to be batched outside the sequential time loop (exercise 1).

Furthermore, a recurrent state can solve tasks that are impossible for *any* fixed-window model, such as reporting at every step whether the running count of ones in a bit stream is even or odd (parity): flipping a bit older than any fixed window m reverses the true parity while leaving the last m bits unchanged, whereas a two-unit recurrent state tracks parity over arbitrary lengths.

8.4.1.1 Constant Memory per Step

Because computing $\mathbf{H}_t$ requires only $\mathbf{H}_{t-1}$ and $\mathbf{X}_t$, both memory and computation per step remain constant regardless of sequence length: with $h = 128$ `float32` hidden units, the state of a sequence occupies just 512 bytes whether the model has read ten tokens or a million. That constant footprint is also an inherent *lossy* bottleneck—every detail remembered from the prefix must pass through those 512 bytes, unlike *attention* (Chapter 10), which retains representations of every past token at a cost that grows with sequence length.

8.4.1.2 RNN Language Models and Teacher Forcing

An RNN language model consumes one token id per step and predicts a distribution over the next token across all $q = |\mathcal{V}|$ vocabulary entries (Figure 8.8). At step t, the b token ids in the minibatch look up their corresponding rows in a learned embedding table $\mathbf{E} \in \mathbb{R}^{|\mathcal{V}| \times d}$ to form $\mathbf{X}_t \in \mathbb{R}^{b \times d}$ (mathematically identical to multiplying a $b \times |\mathcal{V}|$ one-hot matrix by $\mathbf{E}$, without materializing or multiplying by zeros). The recurrent layer updates $\mathbf{H}_t$, and $\mathrm{softmax}(\mathbf{O}_t)$ defines $P(x_{t+1} \mid x_{\le t})$ in the chain-rule factorization $P(x_1, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_{<t})$ (Section 8.3).

Where do the parameters and FLOPs sit? For our BPE vocabulary of $|\mathcal{V}| = 1{,}027$ ids (1,024 BPE tokens plus three special tokens), embedding width $d = 64$, and hidden size $h = 128$:

part	shape	parameters
embedding table $\mathbf{E}$	$1{,}027 \times 64$	65,728
input weight $\mathbf{W}_{\text{xh}}$	64×128	8,192
recurrent weight $\mathbf{W}_{\text{hh}}$	128×128	16,384
hidden bias $\mathbf{b}_{\text{h}}$	128	128
output weight $\mathbf{W}_{\text{hq}}$	$128 \times 1{,}027$	131,456
output bias $\mathbf{b}_{\text{q}}$	$1{,}027$	1,027
total		222,915

The recurrent layer accounts for just 24,704 parameters (11% of the model) and $24{,}576$ multiply-adds per token, whereas the vocabulary-sized embedding and output layers hold 89% of the parameters and the output projection alone costs $128 \cdot 1{,}027 = 131{,}456$ multiply-adds.

During training, the input at every step is the *ground-truth* token from the corpus regardless of what the model predicted at the previous step—a regime known as *teacher forcing*. Because all ground-truth inputs in a window are known up front, the entire window's cross-entropy loss (the logarithm of perplexity) is evaluated in one forward pass. During generation, however, each sampled or greedy token is fed back in as the next input, exposing the rollout to its own errors just like our multistep sine-wave forecast in Section 8.1.

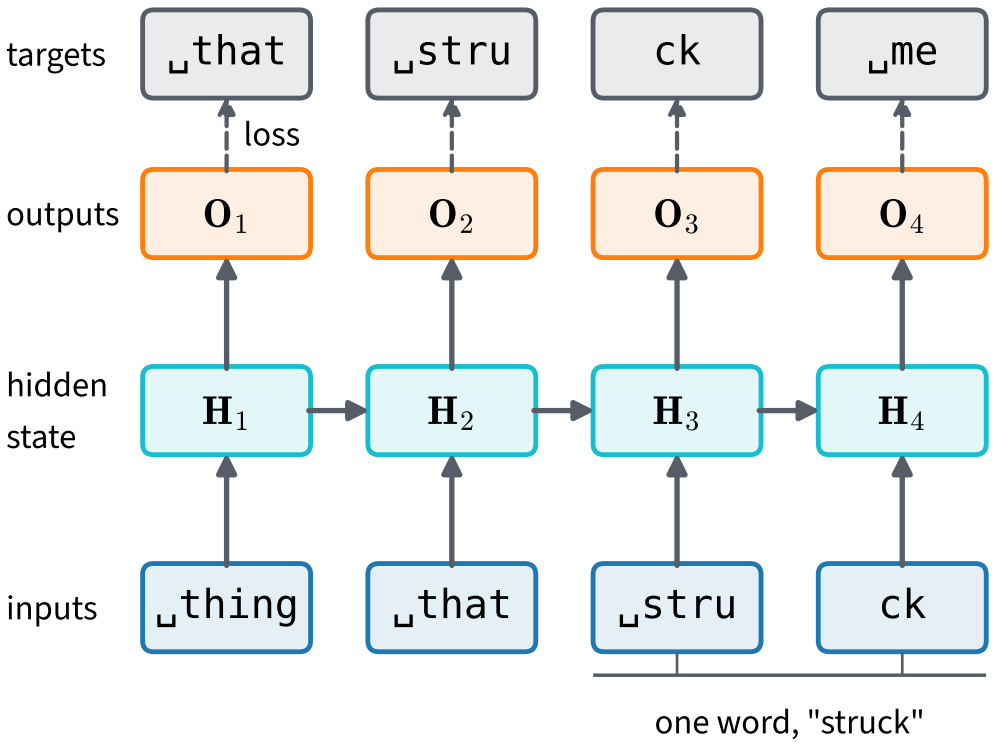


Figure 8.8: An RNN language model trained by teacher forcing over BPE tokens of *The Time Machine*. The target sequence is the input shifted forward by one token, so each step is trained to predict the next token.

8.4.2 An RNN Language Model from Scratch

We now implement the three components of this language model (the recurrent cell, the embedding lookup, and the output layer) from scratch.

8.4.2.1 The Recurrent Cell

The `RNNScratch` class stores the parameters of Equation 8.17 with $\phi = \tanh$. We draw $\mathbf{W}_{\text{xh}}$ and $\mathbf{W}_{\text{hh}}$ from $\mathcal{N}(0, \sigma^2)$ with $\sigma = 0.01$ and initialize $\mathbf{b}_{\text{h}}$ to zero. At this scale, the spectral radius of $\mathbf{W}_{\text{hh}}$ is roughly $\sigma\sqrt{h} \approx 0.11$, so an untrained cell decays its past state rapidly and leaves training to grow recurrent weights where needed.

```
class RNNScratch(d2l.Module):
    """The RNN model implemented from scratch."""
    def __init__(self, num_inputs, num_hiddens, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.W_xh = nn.Parameter(
            d2l.randn(num_inputs, num_hiddens) * sigma)
        self.W_hh = nn.Parameter(
            d2l.randn(num_hiddens, num_hiddens) * sigma)
        self.b_h = nn.Parameter(d2l.zeros(num_hiddens))
```

The forward pass loops over the leading time axis of `inputs` of shape (`num_steps`, `batch_size`, `num_inputs`), initializing the hidden state to zeros when none is supplied and returning the list of all per-step hidden states alongside the final state.

```
@d2l.add_to_class(RNNScratch)
def forward(self, inputs, state=None):
    if state is None:
        # Initial state with shape: (batch_size, num_hiddens)
        state = d2l.zeros((inputs.shape[1], self.num_hiddens),
                          device=inputs.device)
    outputs = []
    for X in inputs:  # Shape of inputs: (num_steps, batch_size, num_inputs)
        state = d2l.tanh(d2l.matmul(X, self.W_xh) +
                         d2l.matmul(state, self.W_hh) + self.b_h)
        outputs.append(state)
    return outputs, state
```

To verify the recurrent cell's dimensions, we first run `RNNScratch` forward on a dummy mini-batch `X` of shape `(num_steps, batch_size, num_inputs)`.

```
batch_size, num_inputs, num_hiddens, num_steps = 2, 16, 32, 100
rnn = RNNScratch(num_inputs, num_hiddens)
X = d2l.ones((num_steps, batch_size, num_inputs))
outputs, state = rnn(X)
```

We now define a `check_shape` helper (reused throughout later chapters) to confirm that stacking `outputs` produces `(num_steps, batch_size, num_hiddens)` and `state` has shape `(batch_size, num_hiddens)`.

```
def check_shape(a, shape):
    """Check the shape of a tensor."""
    assert a.shape == shape, \
            f'tensor\'s shape {a.shape} != expected shape {shape}'

check_shape(d2l.stack(outputs, 0), (num_steps, batch_size, num_hiddens))
check_shape(state, (batch_size, num_hiddens))
```

8.4.2.2 The Embedding and Output Layers

`RNNLMScratch` wraps a recurrent cell `rnn` with an embedding table `E` and an output projection (`W_hq`, `b_q`), plus optional dropout rates on the embeddings and hidden states. Embedding rows are initialized from $\mathcal{N}(0, 1)$ (matching unit-scale one-hot inputs) while output weights use scale $\sigma = 0.01$ with zero bias, so an untrained model produces near-zero logits and an initial perplexity close to $|\mathcal{V}|$. Subclassing `d2l.Classifier` provides mean cross-entropy loss, and overriding `validation_step` records that loss directly so its exponential gives validation perplexity.

```
class RNNLMScratch(d2l.Classifier):
    """The RNN-based language model implemented from scratch."""
    def __init__(self, rnn, vocab_size, lr=0.01, dropout=0, emb_dropout=0):
        super().__init__()
        self.save_hyperparameters()
        self.sigma = getattr(rnn, 'sigma', 0.01)  # Scale of the output weights
        self.init_params()

    def init_params(self):
        self.E = nn.Parameter(
            d2l.randn(self.vocab_size, self.rnn.num_inputs))
        self.W_hq = nn.Parameter(
            d2l.randn(self.rnn.num_hiddens, self.vocab_size) * self.sigma)
        self.b_q = nn.Parameter(d2l.zeros(self.vocab_size))

    def validation_step(self, batch):
        return self.loss(self(*batch[:-1]), batch[-1])
```

Because the data loader yields minibatches of shape (`batch_size`, `num_steps`), `embedding` transposes the token ids before indexing `E`, returning a (`num_steps`, `batch_size`, `num_inputs`) tensor whose leading axis is time.

```
@d2l.add_to_class(RNNLMScratch)
def embedding(self, X):
    # Output shape: (num_steps, batch_size, num_inputs)
    return self.E[X.T]
```

`output_layer` stacks the per-step states into shape (`batch_size`, `num_steps`, `num_hiddens`) and applies the shared output projection in a single batched matrix product, while `forward` chains embedding lookup, dropout, the recurrent cell, and the output projection.

```
@d2l.add_to_class(RNNLMScratch)
def output_layer(self, rnn_outputs):
    # A list of per-step states, or a tensor whose first axis is time
    H = d2l.stack(list(rnn_outputs), 1)  # (batch_size, num_steps, num_hiddens)
    return d2l.matmul(H, self.W_hq) + self.b_q

@d2l.add_to_class(RNNLMScratch)
def forward(self, X, state=None):
    embs = F.dropout(self.embedding(X), self.emb_dropout, self.training)
    rnn_outputs, _ = self.rnn(embs, state)
    H = d2l.stack(list(rnn_outputs), 0)  # (num_steps, batch_size, num_hiddens)
    return self.output_layer(F.dropout(H, self.dropout, self.training))
```

Passing a dummy minibatch of token indices of shape `(batch_size, num_steps)` through `RNNLMScratch` confirms that the returned logits tensor `outputs` has shape `(batch_size, num_steps, vocab_size)`.

```
model = RNNLMScratch(rnn, vocab_size=1024)
outputs = model(d2l.ones((batch_size, num_steps), dtype=d2l.int64))
check_shape(outputs, (batch_size, num_steps, 1024))
```

8.4.3 Gradient Clipping

Because an unrolled RNN applies the same transition Jacobian across T steps, gradients can shrink or grow exponentially with T (Section 8.5). A sudden gradient spike moves the parameters by Euclidean distance $\eta\|\mathbf{g}\|$, overshooting the local region where the first-order Taylor approximation holds and undoing dozens of updates in a single step. Rather than shrinking η across the board, *gradient clipping* caps the global gradient norm (computed jointly over all model parameters) at a threshold θ by projecting $\mathbf{g}$ onto the ball of radius θ:

$$\mathbf{g} \leftarrow \min\left(1, \frac{\theta}{\|\mathbf{g}\|}\right)\mathbf{g}. \tag{8.18}$$

A gradient longer than θ is scaled back to length θ without changing its direction, and a shorter one is left alone (Figure 8.9).

Whenever `gradient_clip_val` is positive, `d2l.Trainer` (Section 2.4) applies Equation 8.18 right after backpropagation and before the optimizer step.

The method below rescales each parameter's gradient in place.

```
@d2l.add_to_class(d2l.Trainer)
def clip_gradients(self, grad_clip_val, model):
    params = [p for p in model.parameters() if p.requires_grad]
    norm = torch.sqrt(sum(torch.sum(p.grad ** 2) for p in params))
    scale = torch.clamp(grad_clip_val / norm, max=1.0)  # no `if` on the host
    for param in params:
        param.grad.mul_(scale)
```

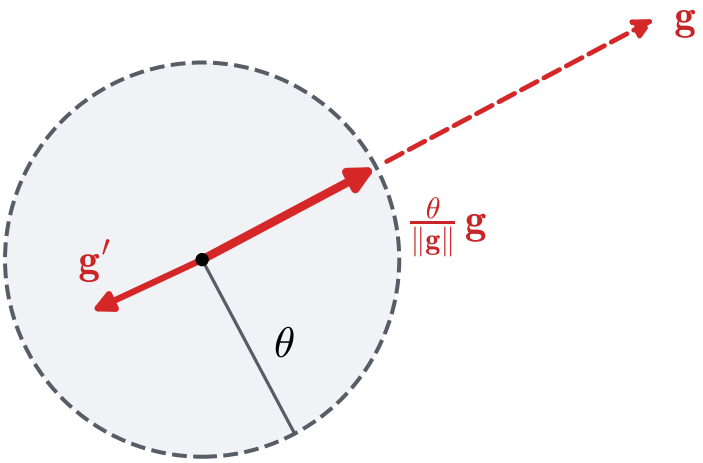


Figure 8.9: Gradient clipping as a projection onto the ball of radius θ. The long gradient $\mathbf{g}$ is rescaled to length θ along its own direction, whereas the gradient $\mathbf{g}'$ inside the ball is left unchanged.

8.4.4 Training and Evaluation

`d2l.TimeMachine` (Section 8.3) encodes the 65,741 BPE tokens of the novella and yields mini-batches of $T = 32$ overlapping windows—50,000 for training and 5,000 for validation, separated by a 32-window gap so no validation token appears as a training target. Because each window starts from $\mathbf{H}_0 = \mathbf{0}$, backpropagation is *truncated* to 32 steps (Section 8.5).

```
data = d2l.TimeMachine(batch_size=1024, num_steps=32)
```

8.4.4.1 A Plain Model and the Role of Clipping

We first instantiate our 222,915-parameter baseline model ($d = 64$, $h = 128$, no dropout) with SGD learning rate $\eta = 4$.

```
rnn = RNNScratch(num_inputs=64, num_hiddens=128)
model = RNNLMScratch(rnn, vocab_size=len(data.vocab), lr=4)
```

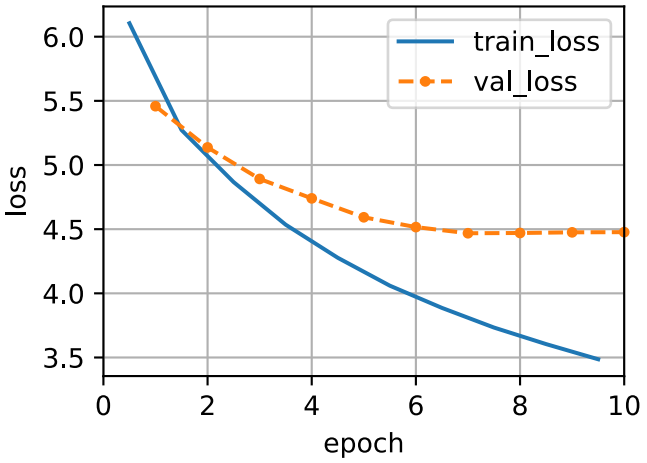
Before training begins, we snapshot a NumPy copy of the random initial embedding matrix `model.E` in `E_init` so we can measure how far SGD moves it.

```
E_init = d2l.numpy(model.E).copy()
```

We next evaluate the untrained model's initial validation perplexity `ppl_init` and then train `model` for 10 epochs with gradient clipping at `gradient_clip_val=1`.

```
trainer = d2l.Trainer(max_epochs=10, gradient_clip_val=1, num_gpus=1)
ppl_init = math.exp(trainer.evaluate(model, data)['loss'])
print(f'initial validation perplexity {ppl_init:.0f}')
trainer.fit(model, data)
```

```
initial validation perplexity 1027
```



Exponentiating the final validation cross-entropy loss `trainer.history['val_loss'][-1]` gives the trained baseline's validation perplexity `ppl_plain`.

```
ppl_plain = math.exp(trainer.history['val_loss'][-1])
print(f'validation perplexity {ppl_plain:.1f}')
```

```
validation perplexity 88.0
```

Initial validation perplexity sits near $|\mathcal{V}| = 1{,}027$, and after ten epochs the clipped model reaches a validation perplexity of about 85 to 100, even though its loss often jumps well above the starting value during the first epoch (while training perplexity drops below half that value as the 222,915 parameters begin overfitting the 50,000 training tokens).

To see why gradient clipping is essential at $\eta = 4$, we define a helper `unclipped_model()` that constructs a fresh baseline instance with the exact same architecture and learning rate.

```
def unclipped_model():
    return RNNLMScratch(RNNScratch(num_inputs=64, num_hiddens=128),
                        vocab_size=len(data.vocab), lr=4)
```

We now train three independent unseeded instances without gradient clipping, recording each run's epoch-by-epoch validation perplexities in `ppl_runs` and reporting its peak and final values.

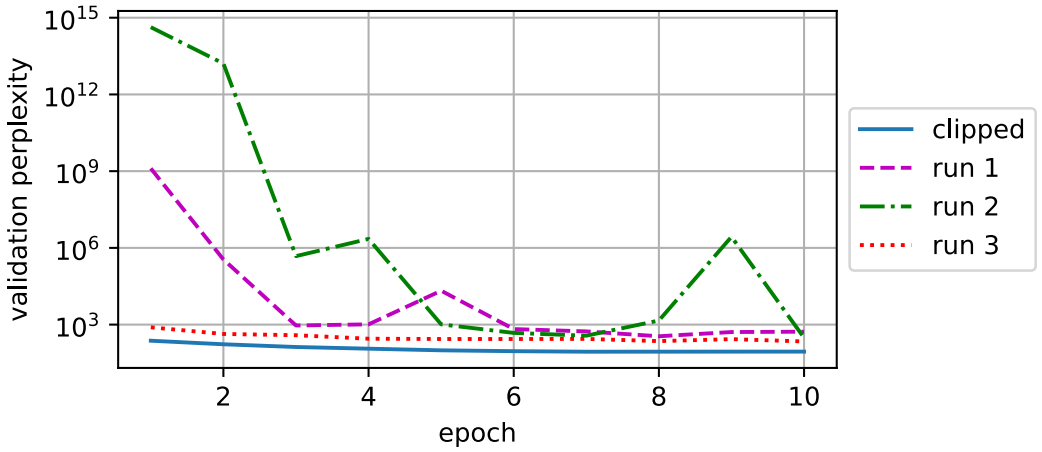
```
ppl_runs = []
for run in range(3):  # Three unseeded runs, no clipping
```

```
    trainer_unclipped = d2l.Trainer(max_epochs=10, num_gpus=1)
    trainer_unclipped.board.display = False  # One plot for all runs below
    trainer_unclipped.fit(unclipped_model(), data)
    ppls = [math.exp(loss) for loss in trainer_unclipped.history['val_loss']]
    ppl_runs.append(ppls)
    print(f'run {run + 1}: highest {max(ppls):.3g}, final {ppls[-1]:.3g}')
finals = [ppls[-1] for ppls in ppl_runs]
print(f'{sum(f > 2 * ppl_plain for f in finals)} of 3 runs end above twice '
      f'the clipped {ppl_plain:.3g}; finals from {min(finals):.3g} to '
      f'{max(finals):.3g}')
```

```
run 1: highest 1.28e+09, final 529
run 2: highest 4.26e+14, final 304
run 3: highest 782, final 219
3 of 3 runs end above twice the clipped 88; finals from 219 to 529
```

Plotting the three unclipped trajectories in `ppl_runs` against the clipped baseline `ppl_clipped` on a logarithmic axis makes the instability unmistakable.

```
ppl_clipped = [math.exp(loss) for loss in trainer.history['val_loss']]
d2l.plot(list(range(1, 11)), [ppl_clipped] + ppl_runs, 'epoch',
         'validation perplexity', yscale='log',
         legend=['clipped', 'run 1', 'run 2', 'run 3'],
         legend_loc='outside', figsize=(5, 2.5))
```



Without clipping, gradient spikes send validation perplexity jumping violently, sometimes far above the untrained baseline of 1,027, and most runs finish at more than twice the clipped model's perplexity. Converting our clipped validation perplexity to bits per byte (Section 8.3.2.4) divides $\log_2$ ppl by the average UTF-8 bytes per BPE token (≈ 2.8):

```
ids = d2l.numpy(data.X[data.num_train:data.num_train+data.num_val, 0]).tolist()
bytes_per_token = len(data.tokenizer.decode(ids).encode('utf-8')) / len(ids)
print(f'{bytes_per_token:.2f} bytes/token, '
      f'{math.log2(ppl_plain) / bytes_per_token:.2f} bits per byte')
```

```
2.78 bytes/token, 2.32 bits per byte
```

8.4.4.2 Embeddings under SGD and a Wider Model with Dropout

Inspecting how far each row of $\mathbf{E}$ moved over the ten epochs of SGD exposes a second bottleneck. We first extract the trained embedding table into a NumPy array `E_final`.

```
E_final = d2l.numpy(model.E)
```

Computing the relative Euclidean displacement `change` between `E_final` and `E_init` across all vocabulary rows reveals how little each token vector moved.

```
change = (((E_final - E_init) ** 2).sum(1) / (E_init ** 2).sum(1)) ** 0.5
print(f'mean change of an embedding row: {change.mean():.1%} of its length')
```

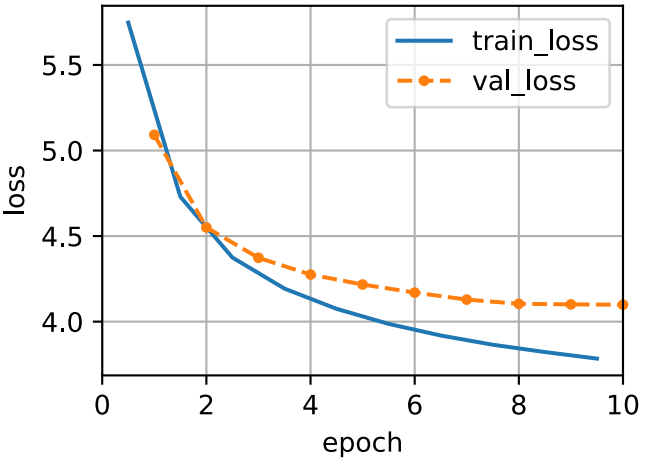
```
mean change of an embedding row: 4.3% of its length
```

Because row i receives a gradient only from positions where token i occurs, and the loss averages over all $1{,}024 \times 32 = 32{,}768$ tokens in a minibatch, rare or moderate-frequency tokens receive tiny gradients under a uniform SGD learning rate, leaving $\mathbf{E}$ almost frozen. While adaptive optimizers like Adam (Section 9.5) rescale per-coordinate step sizes to move embeddings much further (explored in the exercises), we can also address this directly within SGD: widening the embedding from $d = 64$ to $d = 512$ gives the densely updated input matrix $\mathbf{W}_{\text{xh}}$ a far richer random feature basis, while adding dropout (0.1 on embeddings, 0.5 on hidden states) curbs overfitting.

```
rnn = RNNScratch(num_inputs=512, num_hiddens=128)
model = RNNLMScratch(rnn, vocab_size=len(data.vocab), lr=4, dropout=0.5,
                     emb_dropout=0.1)
```

We train this wider regularized model for 10 epochs using the same learning rate `lr=4` and clipping threshold `gradient_clip_val=1`.

```
trainer = d2l.Trainer(max_epochs=10, gradient_clip_val=1, num_gpus=1)
trainer.fit(model, data)
```



From the final validation loss we compute both the validation perplexity `ppl_scratch` and the corresponding compression rate in bits per byte.

```
ppl_scratch = math.exp(trainer.history['val_loss'][-1])
print(f'validation perplexity {ppl_scratch:.1f}, '
      f'{math.log2(ppl_scratch) / bytes_per_token:.2f} bits per byte')
```

```
validation perplexity 60.3, 2.13 bits per byte
```

Validation perplexity drops from 85–100 down to around 60, cutting bits per byte from 2.3–2.4 to roughly 2.1.

8.4.4.3 Comparison with Kneser–Ney and Effective Context Horizon

How does 2.1 bits per byte compare against interpolated Kneser–Ney smoothing (Section 8.3.2.3)? The 1.9 bits per byte that the Kneser–Ney trigram reached in Section 8.3 cannot serve as the yardstick: that number was measured on lowercased text stripped to letters and spaces, whereas this section models the raw novella with its capitals, punctuation, and line breaks. We therefore fit Kneser–Ney n-grams ($n = 2, 3, 4$) on the exact same 50,000 BPE training tokens and evaluate them on `val[32:]`, where each token is scored as the final target of its 32-token validation window.

```
first_tokens = d2l.numpy(data.X[:, 0]).tolist()  # First token of each window
train = first_tokens[:data.num_train]
val = first_tokens[data.num_train:data.num_train + data.num_val]
lp_kns = {n: d2l.kneser_ney(train, val, len(data.vocab), n)[32:]
          for n in (2, 3, 4)}
for n, lp in lp_kns.items():
    bits = -sum(lp) / len(lp) / math.log(2)
    print(f'KN {n}-gram: {bits / bytes_per_token:.2f} bits per byte')
lp_kn = lp_kns[3]
```

```
KN 2-gram: 2.24 bits per byte
KN 3-gram: 2.09 bits per byte
KN 4-gram: 2.08 bits per byte
```

To score our trained RNN on the exact same held-out tokens `val[32:]`, we select the slice `[start:stop]` of validation windows whose final target covers `val[32:]`.

```
start, stop = data.num_train, data.num_train + data.num_val - 32  # val[32:]
```

Running `model` in evaluation mode over `data.X[start:stop]` in batches of 1,024 windows collects the matrix `logp` of per-position target log-probabilities across all 32 window steps.

```
model.eval()
with torch.inference_mode():  # Log-probability of every target
    logp = d2l.numpy(torch.cat([
        F.log_softmax(model(X.to(d2l.try_gpu())), -1).gather(
            -1, Y.to(d2l.try_gpu())[..., None])[..., 0]
        for X, Y in zip(data.X[start:stop].split(1024),
                        data.Y[start:stop].split(1024))]))
```

The RNN and the trigram arrive at nearly the same average. Do they also make the same predictions? The last column of `logp` scores the same tokens `val[32:]` as `lp_kn`, so we can compare the two models token by token. We also score their equal mixture $0.5\,P_{\text{RNN}} + 0.5\,P_{\text{KN}}$, the usual way of combining a neural language model with a count-based one (Mikolov et al. 2011).

```
lp_rnn, lp_tri = logp[:, -1], np.array(lp_kn)
lp_mix = np.logaddexp(lp_rnn, lp_tri) + math.log(0.5)  # 0.5 P_RNN + 0.5 P_KN
for name, lp in [('RNN', lp_rnn), ('KN trigram', lp_tri), ('mixture', lp_mix)]:
    print(f'{name}: {-lp.mean() / math.log(2) / bytes_per_token:.2f} '
          'bits per byte')
print(f'RNN more confident on {np.mean(lp_rnn > lp_tri):.0%} of the tokens')
```

```
RNN: 2.12 bits per byte
KN trigram: 2.09 bits per byte
mixture: 2.03 bits per byte
RNN more confident on 50% of the tokens
```

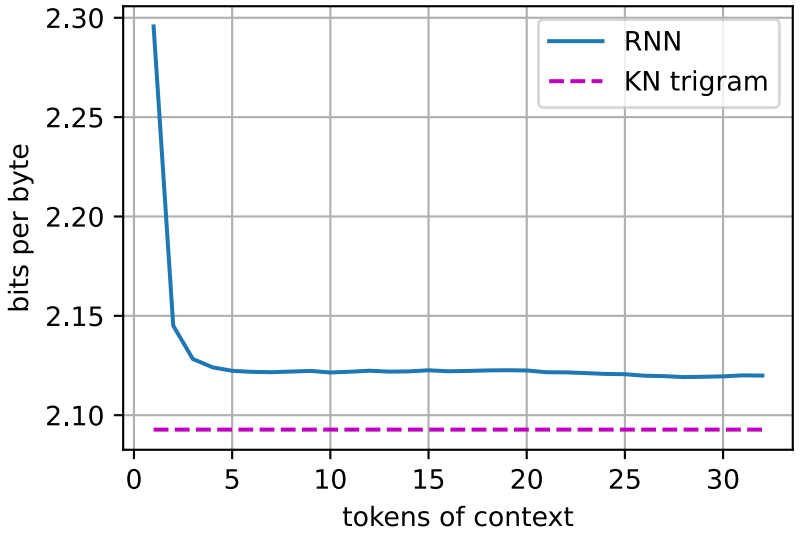
The two models tie on average, but not token by token. The RNN assigns the higher probability on about half the tokens and the trigram on the other half, and the mixture lands about 0.06 bits per byte below the trigram, ahead of both of its members.

A mixture can gain only where its members disagree, so the two models have learned partly different things about the text. That makes the tie more puzzling, not less. Why does a recurrent network reading 32 tokens of history land near the same $\approx$ 2.1 bits per byte as a Kneser–Ney trigram that looks back only two tokens? To see how much context the trained RNN actually uses, we plot its average validation bits per byte at each window position $1, \ldots, 32$ against the Kneser–Ney trigram baseline.

```
bits = -logp.mean(0) / math.log(2) / bytes_per_token  # One per position
kn_bits = -sum(lp_kn) / len(lp_kn) / math.log(2) / bytes_per_token
positions = np.arange(1, data.num_steps + 1)
d2l.plot(positions, [bits, [kn_bits] * data.num_steps],
         'tokens of context', 'bits per byte', legend=['RNN', 'KN trigram'],
```

```
          figsize=(4.5, 3))
print('bits per byte with 1, 2, 3, 4, 8, 32 tokens of context:',
      [round(float(bits[k - 1]), 2) for k in (1, 2, 3, 4, 8, 32)])
```

```
bits per byte with 1, 2, 3, 4, 8, 32 tokens of context: [2.3, 2.15, 2.13, 2.12, 2.12, 2.12]
```



The diagnostic curve explains the tie. Almost all of the gain arrives with the second token of context, which lowers the loss by about 0.15, from about 2.3 to about 2.14 bits per byte. The third token adds a hundredth or two, and from position 4 through 32 the curve is flat. Although the hidden state *can* depend on all 32 tokens in the forward pass, the trained model exploits only the last two or three.

Why so few? One explanation is optimization: the gradient that would teach the cell to use a distant token shrinks at every step on its way back, as we measure in Section 8.5. The other is data: 50,000 training tokens may hold little that can be learned about a token from those more than two or three positions before it. The Kneser–Ney models printed above speak for the second explanation. They are fit by counting, with no gradient that could vanish, and they saturate at the same place: the trigram improves on the bigram by 0.15 bits per byte, and the 4-gram improves on the trigram by only 0.01. Our experiment cannot tell the two explanations apart. A clean test needs a model whose gradients survive over long lags, trained on the same tokens, and Chapter 12 builds such models.

8.4.5 Text Generation

Once trained, our language model generates text autoregressively via `predict`: it first warms up the hidden state on the user's prefix tokens, then generates `num_tokens` continuation tokens by feeding each chosen token back in as the next input (using greedy argmax when `temperature=0.0`, or sampling from the temperature-scaled softmax otherwise; see Section 8.6).

```
@d2l.add_to_class(RNNLMScratch)
@torch.inference_mode()  # Generation needs no autograd graph
```

```
def predict(self, prefix, num_tokens, tok, device=None, temperature=0.0):
    outputs, state = tok.encode(prefix), None
    for i in range(len(outputs) - 1):  # Warm up on the prefix
        X = d2l.tensor([[outputs[i]]], device=device)
        _, state = self.rnn(self.embedding(X), state)
    for _ in range(num_tokens):  # Generate num_tokens continuation tokens
        X = d2l.tensor([[outputs[-1]]], device=device)
        rnn_outputs, state = self.rnn(self.embedding(X), state)
        logits = self.output_layer(rnn_outputs)[0, 0]
        if temperature == 0:  # Greedy: the highest-scoring token
            token = logits.argmax()
        else:  # Sample from the softmax of logits / temperature
            token = torch.distributions.Categorical(
                logits=logits / temperature).sample()
        outputs.append(int(token))
    return tok.decode(outputs)
```

Let us test `model.predict` with greedy decoding (`temperature=0.0`) by generating a 50-token continuation from the prompt `'The Time Traveller'`.

```
d2l.print_wrapped(repr(model.predict('The Time Traveller', 50,
                                     data.tokenizer, d2l.try_gpu())))
```

```
'The Time Traveller,\nthe\nfrom the\nfound,\ntheir\nfound,\ntheir\nfound,\nt
    heir\nfound,\ntheir\nfound,\ntheir\nfound,\ntheir\n'
```

Because greedy decoding is deterministic, once the hidden state revisits a recent phrase it cycles through that loop indefinitely—a failure mode we study and fix in Section 8.6.

8.4.6 Concise Implementation

Framework-supplied recurrent layers fuse the recurrent step and time loop into optimized low-level kernels. The `RNN` module below matches the interface and initialization of `RNNScratch`, and `RNNLM` subclasses `RNNLMScratch` by swapping in built-in embedding and linear layers.

`nn.RNN` keeps two bias vectors, one for the input term and one for the recurrent term. Their sum plays the role of $\mathbf{b}_\text{h}$, and both start at zero.

```
class RNN(d2l.Module):
    """The RNN model implemented with high-level APIs."""
    def __init__(self, num_inputs, num_hiddens, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.rnn = nn.RNN(num_inputs, num_hiddens)
        for name, param in self.rnn.named_parameters():
            if name.startswith('weight'):
                nn.init.normal_(param, std=sigma)
            else:  # Both bias vectors start at zero
                nn.init.zeros_(param)
```

```
    def forward(self, inputs, H=None):
        return self.rnn(inputs, H)
```

`nn.Embedding` draws its rows from $\mathcal{N}(0, 1)$ by default; the weights of `nn.Linear` are set explicitly.

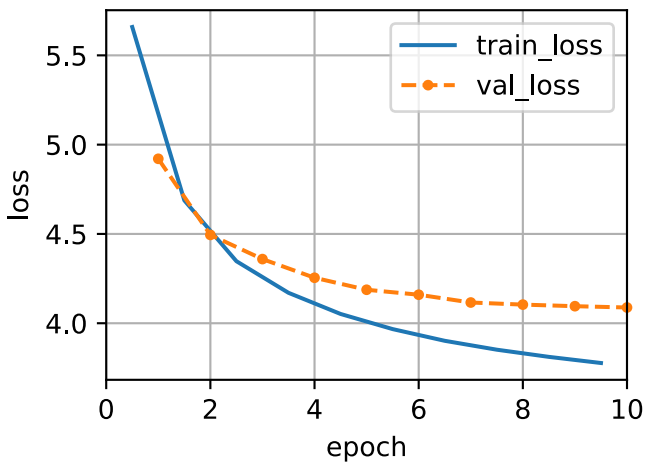
```
class RNNLM(d2l.RNNLMScratch):
    """The RNN-based language model implemented with high-level APIs."""
    def init_params(self):
        self.emb = nn.Embedding(self.vocab_size, self.rnn.num_inputs)
        self.linear = nn.Linear(self.rnn.num_hiddens, self.vocab_size)
        nn.init.normal_(self.linear.weight, std=self.sigma)
        nn.init.zeros_(self.linear.bias)

    def embedding(self, X):
        return self.emb(X.T)

    def output_layer(self, hiddens):
        return d2l.swapaxes(self.linear(hiddens), 0, 1)
```

To verify our high-level implementation, we instantiate `RNN` and `RNNLM` with the same hyperparameters ($d = 512$, $h = 128$, and dropout).

```
rnn = RNN(num_inputs=512, num_hiddens=128)
model = RNNLM(rnn, vocab_size=len(data.vocab), lr=4, dropout=0.5,
              emb_dropout=0.1)
```

We now train `RNNLM` for 10 epochs with `gradient_clip_val=1`, running the fused recurrent kernel at a fraction of the from-scratch wall-clock time.

```
trainer = d2l.Trainer(max_epochs=10, gradient_clip_val=1, num_gpus=1)
trainer.fit(model, data)
```



Finally, we report the concise model's validation perplexity `ppl_concise` alongside a 30-token greedy continuation from `'The Time Traveller'`.

```
ppl_concise = math.exp(trainer.history['val_loss'][-1])
pred = model.predict('The Time Traveller', 30, data.tokenizer, d2l.try_gpu())
d2l.print_wrapped(f'perplexity {ppl_concise:.1f}, {pred!r}')
```

```
perplexity 59.6, "The Time Traveller.\n\n'I
    had\nfossibly\nthere\nfoundly\nfossibly\nthere\nf"
```

8.4.7 Summary

A recurrent neural network updates a fixed-size hidden state from the current input and previous state using weights shared across all time steps, achieving constant memory and compute per step at the cost of compressing history into a lossy state vector. An RNN language model pairs an embedding table, a tanh recurrent cell, and a vocabulary-sized output projection (where 89% of our baseline's parameters reside), trained by teacher forcing to minimize per-token cross-entropy. Global gradient clipping (Equation 8.18) prevents exploding gradient spikes from derailing training at high learning rates. On *The Time Machine*, a wider embedding with dropout brings the SGD-trained RNN to ≈ 2.1 bits per byte, tying an interpolated Kneser–Ney trigram: the trained vanilla RNN effectively exploits only the last two or three tokens of context. Section 8.5 measures how far its gradients reach.

8.4.8 Exercises

1. **Separating the input product.** Let $\mathbf{X}_t \in \mathbb{R}^{b\times d}$ and $\mathbf{H}_{t-1} \in \mathbb{R}^{b\times h}$ be the input and the state in Equation 8.17.
    1. During training, the inputs $\mathbf{X}_1, \ldots, \mathbf{X}_T$ of a window are all known before the first time step is computed. Which of the two products $\mathbf{X}_t\mathbf{W}_{\text{xh}}$ and $\mathbf{H}_{t-1}\mathbf{W}_{\text{hh}}$ can be computed for all T steps in a single matrix multiplication, and what are the shapes of its factors? Which product must stay inside the sequential loop over time steps, and why?
    2. Compare the work per step left inside the loop with the work per step of the concatenated form $[\mathbf{X}_t, \mathbf{H}_{t-1}]$. During generation, the input at step t is the token generated at step $t-1$. Does the advantage of the separate form survive there?
2. **Exposure bias.** Teacher forcing (Section 8.4.1.2) conditions every training step on the true prefix, whereas generation conditions each step on the model's own earlier outputs. The resulting mismatch is called *exposure bias* (Ranzato et al. 2016).
    1. Suppose generation samples each token from the model's next-token distribution, and the model assigns probability $1-\epsilon$ to the correct next token whenever its prefix is correct. What is the probability that T generated tokens contain no error? Evaluate it for $\epsilon = 0.01$ and $T = 100$.
    2. *Scheduled sampling* (Bengio et al. 2015) exposes the model to its own outputs during training: at each step, with some probability, the input is a token sampled from the model's prediction at the previous step instead of the true token. Which property of teacher forcing that Section 8.4.1.2 relies on does this give up, and how does that change the computation of a training window?

3. **A window inside a recurrence.** The neural n-gram reads a fixed window of the last m input embeddings. Take ϕ to be the identity and $\mathbf{b}_{\text{h}} = \mathbf{0}$ in Equation 8.17, and give the state $h = md$ units.

    1. Choose $\mathbf{W}_{\text{xh}}$ and $\mathbf{W}_{\text{hh}}$ so that, starting from $\mathbf{H}_0 = \mathbf{0}$, the state is the concatenation $\mathbf{H}_t = [\mathbf{X}_t, \mathbf{X}_{t-1}, \dots, \mathbf{X}_{t-m+1}]$ for every $t \geq m$.
    2. Show that $\mathbf{W}_{\text{hh}}^m = \mathbf{0}$, and explain why this means that no input older than m steps affects $\mathbf{H}_t$.
    3. For $d = 256$ and $m = 20$, how many state units does this construction need? Explain what an RNN whose state has only 256 units must do instead of storing the window.

4. code **Embedding lookup as a product.** Let $\mathbf{E} \in \mathbb{R}^{|\mathcal{V}| \times d}$ be an embedding table, and let $\mathbf{x}$ hold the token ids of a minibatch of b tokens.

    1. Compute the inputs once by selecting the rows of $\mathbf{E}$ that $\mathbf{x}$ indexes and once as the product of the $b \times |\mathcal{V}|$ one-hot matrix of $\mathbf{x}$ with $\mathbf{E}$, and check that the two agree. Time both for $|\mathcal{V}| \in \{10^3, 10^4, 10^5\}$ with $d = b = 256$, and state how the running time of each grows with $|\mathcal{V}|$.
    2. Differentiate the sum of all entries of the selected rows with respect to $\mathbf{E}$ by automatic differentiation. Which rows of the gradient are nonzero, and what does a row contain when its token occurs several times in the minibatch? For $|\mathcal{V}| = 10^4$ and $b = 256$ ids drawn uniformly at random, estimate the fraction of rows that receive a nonzero gradient.

5. **Parity beyond any window.** ● Let $x_1, x_2, \dots$ be a sequence of bits, and let the desired output at step t be the parity of $x_1 + \cdots + x_t$.

    1. Show that no model that reads only the last m inputs can produce the correct output at every step of a sequence longer than m, whatever the finite m.
    2. Let ϕ be a threshold activation. Construct an RNN of the form Equation 8.17 with two hidden units whose output equals the parity at every step, for any sequence length. Give the weights, and show that one hidden unit does not suffice.
    3. How many bits of information about $x_1, \dots, x_t$ must the hidden state retain for the outputs at all later steps to be correct? Relate this to the description of the state as a lossy fixed-size summary in Section 8.4.1.1.

6. code **Gradient clipping ablation.** `d2l.Trainer` applies the projection Equation 8.18 whenever `gradient_clip_val` is positive.

    1. Record the norm of the gradient before clipping at every update of the first epoch, and plot it. How often does clipping act, and when?
    2. Most unclipped runs of this section jumped at `lr=4`. Halve the learning rate until three unclipped runs in a row train without a jump in the validation perplexity. How does their validation perplexity after ten epochs compare with that of the clipped run at `lr=4`? What, then, does clipping make possible?
    3. Replace tanh by ReLU in `RNNScratch` and train again, with and without clipping. Is clipping still needed? Explain the difference from the tanh cell in terms of the range of the hidden state.

7. code **Usable history.** Every training window starts from a zero state, so the model is never trained to use more than `num_steps` tokens of context, and the loss by position showed that it uses far fewer. `predict`, by contrast, carries the state across the whole prefix and every generated token. Evaluate the trained model on 512 consecutive val-

idation tokens in two ways: resetting the state to zero every 32 tokens as in training, and carrying the state across all 512 tokens. Compare the average cross-entropy of the two. Does the carried state help, hurt, or make no difference, and what does the answer say about the states the model reaches beyond its training horizon?

8. code **Adam.** Train the model with the wider embedding with the Adam optimizer of Section 9.5 instead of SGD: subclass `RNNLMScratch`, override `configure_optimizers` to return Adam with learning rate 0.003, and keep clipping at 1.

    1. With `emb_dropout=0.2`, score the tokens `val[32:]` as in the comparison above, and compare its bits per byte with the SGD model, the Kneser–Ney trigram and 4-gram, and the mixture of the RNN and the trigram at $\lambda = 0.5$.
    2. Record the validation loss after every epoch, with `emb_dropout=0.1` and with 0.2. At which epoch is it lowest, and what happens after that?
    3. Measure again how far the embedding rows move during training. Why does Adam move them further than SGD does?

9. code **Vocabulary size and bits per byte.** ● Retrain the model with the wider embedding for `vocab_size` 512, 1,024, 2,048, and 4,096 in `d2l.TimeMachine`. The number of windows depends on the vocabulary size, so split by fraction: build each dataset once to read off `n = len(data.X)`, then build it again with `num_train=int(0.8 * n)` and `num_val=n - int(0.8 * n)`. (Setting `data.num_train` afterwards would skip the step that keeps validation tokens out of the training targets.) Every model then trains and validates on about the same stretch of text. Their per-token perplexities are not comparable, so compare them by bits per byte.

    1. Which vocabulary size gives the lowest bits per byte?
    2. How do the parameter counts of the embedding table and the output layer, and the length of the corpus in tokens, change with the vocabulary size? Which of these effects best explains the trend you measured?

10. code **Cross-book generalization.** ● Subclass `d2l.TimeMachine` and override `_download` to return the text of a different H. G. Wells novel, such as *The War of the Worlds* (Project Gutenberg ebook 36). Train the tokenizer and the model on it.

    1. Evaluate the model on held-out text of its own novel and on *The Time Machine*, encoding both with the new novel's tokenizer. Report perplexity and bits per byte for both.
    2. Write $s = \log_2(\text{ppl})$ for the per-token surprisal and b for the bytes per token of an evaluated text, so that bits per byte is s/b. The ratio of bits per byte on *The Time Machine* (TM) to bits per byte on the held-out text of the new novel (own) factors as
    $$\frac{s_{\text{TM}}}{s_{\text{own}}} \cdot \frac{b_{\text{own}}}{b_{\text{TM}}}. \tag{8.19}$$
    The first factor reflects the model's predictions and the second the tokenizer's compression. Measure both factors. Which lies further from one?

8.5 Backpropagation Through Time

Our experiments with the RNN language model in Section 8.4.2 left us with two striking empirical findings: without gradient clipping, most training runs destabilized; with clipping,

training converged reliably, yet the learned network used only the last few tokens of its 32-token context window. Both phenomena share the exact same mathematical root. Because a recurrent network applies the same weight matrix at every time step, any gradient propagating backward through a sequence is multiplied by a state Jacobian at each step. Over many steps these factors compound, much like compound interest, causing the gradient to grow or shrink geometrically.

Even a scalar recurrence $h_t = w_{\text{h}} h_{t-1} + x_t$ shows why: unrolling gives $h_t = \sum_{s \le t} w_{\text{h}}^{t-s} x_s$, so $\partial h_t / \partial h_{t-k} = w_{\text{h}}^k$. Over forty steps, $w_{\text{h}} = 0.9$ shrinks a gradient to $0.9^{40} \approx 0.015$, whereas $w_{\text{h}} = 1.1$ inflates it to $1.1^{40} \approx 45$. Exploding gradients destabilize optimization immediately, forcing us to clip updates; vanishing gradients are quieter yet more insidious, attenuating error signals from distant tokens until the model cannot learn long-range dependencies.

Computing gradients in an RNN requires no new differentiation algorithm: we unroll the recurrence across time into a deep feedforward network with shared weights and apply standard backpropagation (Section 4.3), a procedure known as *backpropagation through time* (BPTT) (Werbos 1990). Below, we derive the backward recursion directly in matrix form, measure how far gradients actually travel inside our trained RNN from Section 8.4.2, and examine truncated backpropagation through time.

8.5.1 Vanishing and Exploding Gradients

Consider an RNN unrolled across T steps with shared recurrent parameters and average sequence loss $L = \frac{1}{T} \sum_{t=1}^{T} l(\mathbf{y}_t, \mathbf{o}_t)$. Backpropagation handles weight sharing by summing the local gradient contributions from every time step where a weight matrix appears. To see the structure of the backward pass transparently, take the RNN of Section 8.4 and temporarily omit biases and the tanh nonlinearity (we restore tanh immediately afterward). For a minibatch of b sequences with step inputs $\mathbf{X}_t \in \mathbb{R}^{b \times d}$, the linear RNN computes

$$\begin{aligned} \mathbf{H}_t &= \mathbf{X}_t \mathbf{W}_{\text{xh}} + \mathbf{H}_{t-1} \mathbf{W}_{\text{hh}}, \\ \mathbf{O}_t &= \mathbf{H}_t \mathbf{W}_{\text{hq}}, \end{aligned} \tag{8.20}$$

with $\mathbf{W}_{\text{xh}} \in \mathbb{R}^{d \times h}$, $\mathbf{W}_{\text{hh}} \in \mathbb{R}^{h \times h}$, and $\mathbf{W}_{\text{hq}} \in \mathbb{R}^{h \times q}$. Figure 8.10 depicts three unrolled time steps of this network, showing the forward pass in grey and the backward pass in red.

Let $\mathbf{G}_t = \partial L / \partial \mathbf{H}_t$ denote the gradient of the total loss with respect to $\mathbf{H}_t$. Because $\mathbf{H}_t$ feeds into both its immediate output $\mathbf{O}_t$ and the next hidden state $\mathbf{H}_{t+1}$, and backpropagating through a right-multiplication multiplies by the transposed weight matrix (Section 4.3), $\mathbf{G}_t$ obeys the backward recursion

$$\begin{aligned} \mathbf{G}_t &= \frac{\partial L}{\partial \mathbf{O}_t} \mathbf{W}_{\text{hq}}^\top + \mathbf{G}_{t+1} \mathbf{W}_{\text{hh}}^\top, \\ \mathbf{G}_{T+1} &= \mathbf{0}, \end{aligned} \tag{8.21}$$

corresponding to the two incoming red arrows at each hidden state in Figure 8.10. Unrolling Equation 8.21 backward from $\mathbf{G}_T$ down to $\mathbf{G}_t$ yields

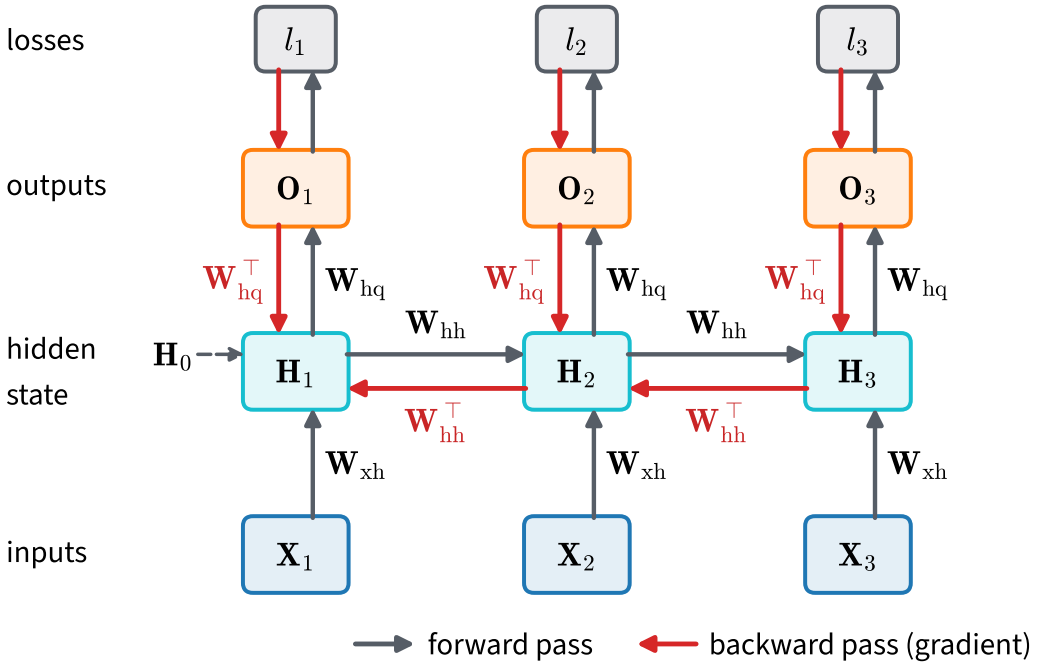


Figure 8.10: Forward pass (grey) and backward pass (red) of a linear RNN unrolled over three time steps. The gradient that reaches each hidden state except the last is the sum of two red arrows: one from its own output, through $\mathbf{W}_{\text{hq}}^\top$, and one from the next state, through $\mathbf{W}_{\text{hh}}^\top$.

$$\mathbf{G}_t = \sum_{j=t}^{T} \frac{\partial L}{\partial \mathbf{O}_j} \mathbf{W}_{\text{hq}}^\top \left(\mathbf{W}_{\text{hh}}^\top\right)^{j-t}. \tag{8.22}$$

Thus, the loss at future step j communicates back to the state $k = j - t$ steps earlier through the k^{th} power of $\mathbf{W}_{\text{hh}}^\top$ (for a general nonlinear transition $\mathbf{H}_t = g(\mathbf{H}_{t-1}, \mathbf{X}_t, \mathbf{W}_{\text{hh}})$, this matrix power is replaced by the product of the $j - t$ intermediate state Jacobians $\prod_{i=t+1}^{j} \partial \mathbf{H}_i / \partial \mathbf{H}_{i-1}$). Once $\mathbf{G}_1, \ldots, \mathbf{G}_T$ are computed in a single backward sweep, the parameter gradients simply sum one outer product per step:

$$\begin{aligned} \frac{\partial L}{\partial \mathbf{W}_{\text{hh}}} &= \sum_{t=1}^{T} \mathbf{H}_{t-1}^\top \mathbf{G}_t, \\ \frac{\partial L}{\partial \mathbf{W}_{\text{xh}}} &= \sum_{t=1}^{T} \mathbf{X}_t^\top \mathbf{G}_t. \end{aligned} \tag{8.23}$$

As we saw in Section 1.3.5 and Section 4.4, raising a matrix to the k^{th} power scales each eigenvector by λ^k. Letting ρ denote the *spectral radius* (largest $|\lambda|$) of $\mathbf{W}_{\text{hh}}$, the norm of $(\mathbf{W}_{\text{hh}}^\top)^k$ scales asymptotically as ρ^k: when $\rho < 1$, contributions from distant steps vanish exponentially; when $\rho > 1$, they explode exponentially. Reintroducing the tanh activation multiplies each step's backward gradient entrywise by its local derivative $1 - \mathbf{H}_t^2 \in (0, 1]$ before multiplying by $\mathbf{W}_{\text{hh}}^\top$. Because $1 - h^2 \leq 1$, tanh never amplifies a gradient and pulls strongly toward vanishing whenever hidden units saturate near ± 1, while still permitting explosion if $\mathbf{W}_{\text{hh}}$ is large and activations stay near zero.

Let us measure what actually happens inside our trained language model from Section 8.4.2. We first load the `TimeMachine` dataset with 32-token windows.

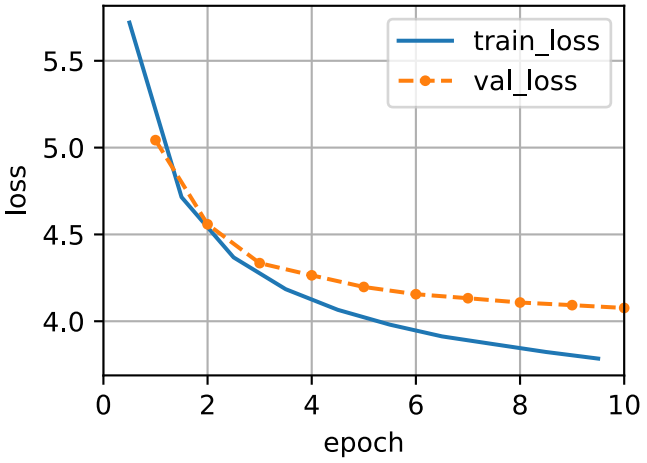
```
data = d2l.TimeMachine(batch_size=1024, num_steps=32)
```

Next, we instantiate the from-scratch recurrent language model `RNNLMScratch` with embedding dimension $d = 512$, hidden size $h = 128$, and dropout.

```
rnn = d2l.RNNScratch(num_inputs=512, num_hiddens=128)
model = d2l.RNNLMScratch(rnn, vocab_size=len(data.vocab), lr=4, dropout=0.5,
                         emb_dropout=0.1)
```

We train `model` for 10 epochs with learning rate `lr=4` and global gradient clipping at `gradient_clip_val=1`.

```
trainer = d2l.Trainer(max_epochs=10, gradient_clip_val=1, num_gpus=1)
trainer.fit(model, data)
```



To probe gradient flow over a longer horizon, we sample 256 random validation segments `seqs` of length $L = 64$ (twice the training window) along with their following tokens `targets`.

```
L = 64
val = d2l.numpy(data.X[data.num_train:data.num_train + data.num_val, 0])
starts = np.random.default_rng().choice(len(val) - L, 256, replace=False)
seqs = np.stack([val[s:s + L] for s in starts])  # (256, L) token ids
targets = val[starts + L]                        # The token after each stretch
```

We now run `model.rnn` forward across `seqs` to collect all 64 per-step hidden states in `H`, the final-step `logits`, and the weight matrices `W_hh` and `W_hq`.

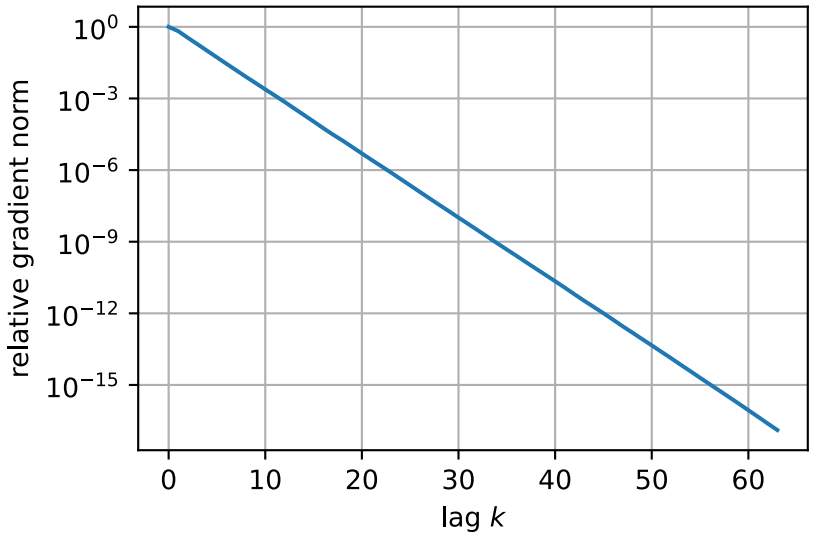
```
with torch.inference_mode():
    states, _ = model.rnn(model.embedding(
        d2l.tensor(seqs, device=d2l.try_gpu())))
    logits = model.output_layer(states[-1:])[:, 0]  # The last step only
H, logits = d2l.numpy(d2l.stack(states, 0)), d2l.numpy(logits)
W_hh, W_hq = d2l.numpy(model.rnn.W_hh), d2l.numpy(model.W_hq)
```

Starting from the final-step loss gradient $\partial L_L/\partial \mathbf{O}_L = \mathrm{softmax}(\mathbf{O}_L) - \mathbf{Y}_L$, we step backward through time via $\mathbf{G}_{t-1} = (\mathbf{G}_t \odot (1 - \mathbf{H}_t^2))\mathbf{W}_{\mathrm{hh}}^\top$ and plot the geometric mean of the relative gradient norm $\|\mathbf{G}_{L-k}\|/\|\mathbf{G}_L\|$ as a function of the lag k.

```
P = np.exp(logits - logits.max(1, keepdims=True))
P /= P.sum(1, keepdims=True)
P[np.arange(len(targets)), targets] -= 1  # dL/dO: softmax minus one-hot
G = (P @ W_hq.T).astype(np.float64)       # dL/dH at the last step
norms = []
for t in range(L - 1, -1, -1):            # Backward through time
    norms.append(np.linalg.norm(G, axis=1))
    G = (G * (1 - H[t] ** 2)) @ W_hh.T    # Through the tanh, then W_hh
rel = np.exp(np.log(np.array(norms) / norms[0]).mean(1))
print(f'spectral radius of W_hh: {np.abs(np.linalg.eigvals(W_hh)).max():.2f}; '
      f'mean tanh derivative: {(1 - H ** 2).mean():.2f}')
```

```
print('relative gradient norm at lags 1, 2, 4, 8, 16, 32:',
      [float(f'{rel[k]:.2g}') for k in (1, 2, 4, 8, 16, 32)])
d2l.plot(np.arange(L), rel, 'lag $k$', 'relative gradient norm',
         yscale='log', figsize=(4.5, 3))
```

```
spectral radius of W_hh: 1.82; mean tanh derivative: 0.27
relative gradient norm at lags 1, 2, 4, 8, 16, 32: [0.66, 0.35, 0.099, 0.0081, 5.7e-05, 3e-09]
```



Even though the trained $\mathbf{W}_{\text{hh}}$ has a spectral radius near 1.8, which would explode in a purely linear recurrence, the actual gradient vanishes rapidly: it drops below 1% of its initial norm within roughly eight steps and shrinks to a few billionths across 32 tokens. Each backward step retains a little over half of the gradient norm (the printed value at lag 8 gives $0.008^{1/8} \approx 0.55$), close to the rough estimate $1.8 \times 0.27 \approx 0.5$ that multiplies the spectral radius by the average tanh derivative. This decay is consistent with our finding in Section 8.4.2 that per-position validation loss flattens out after two or three tokens of context, but it does not prove that vanishing gradients are the cause. At lags 2 and 4 the gradient still has about a third and a tenth of its size, an attenuation that SGD routinely learns through, and the Kneser–Ney counts of Section 8.4.2 saturate at the same context length without using gradients at all. What the decay does rule out is learning from tokens much further back, whatever the data may hold.

8.5.1.1 Remedies for Exploding and Vanishing Gradients

Exploding and vanishing gradients demand fundamentally different remedies. Because an exploding gradient threatens to corrupt the parameters in a single oversized update, global gradient clipping (Section 8.4) provides a simple safeguard—yet multiplying the entire gradient sum Equation 8.22 by a scalar never changes the *relative* weight of distant versus recent lags. Overcoming vanishing gradients therefore requires changing the recurrence itself so that state Jacobians stay near unity along memory directions: either via *gated* recurrent cells (such as LSTMs and GRUs) that create an additive memory path controlled by learned gates, or via *structured linear recurrences* and state space models that remove the state nonlinearity and constrain eigenvalues near the unit circle. Both families are developed in Chapter 12.

8.5.2 Truncated Backpropagation Through Time

Evaluating Equation 8.21 across an entire corpus of T tokens requires retaining all T forward hidden states in memory and traversing all T steps for a single parameter update. *Truncated backpropagation through time* caps both memory and compute per update by slicing long streams into fixed-length segments of length T_0 and halting backpropagation at each segment boundary (Figure 8.11).

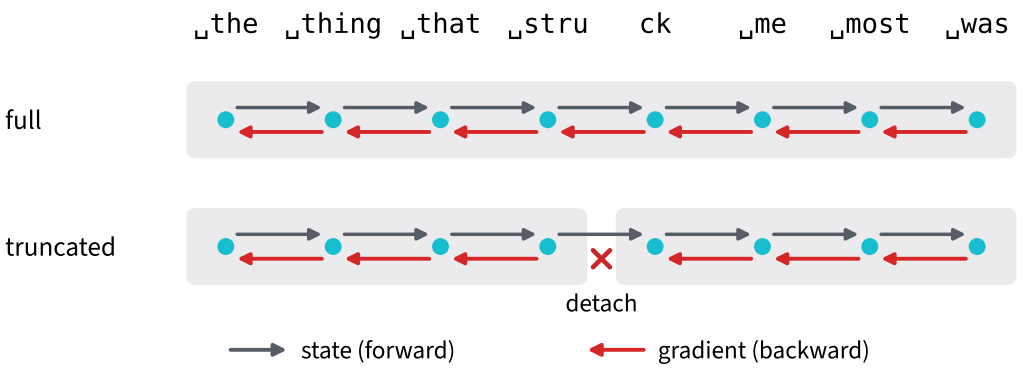


Figure 8.11: Full and truncated backpropagation through time over a token sequence. Both run the same forward pass (grey). Full backpropagation sends the gradient (red) back along the entire sequence. Truncation cuts the sequence into segments of four tokens and detaches the state at each boundary: its value crosses into the next segment, but no gradient flows back across.

Whether each window resets $\mathbf{H}_0 = \mathbf{0}$ (as in Section 8.4.2) or carries the previous segment's final state forward while *detaching* it from the computational graph (as in Figure 8.11), no gradient ever crosses a segment boundary, imposing a hard learning horizon of at most T_0 steps (Jaeger 2002). In a vanilla RNN, however, vanishing gradients already choke off the backward signal after roughly eight steps, well inside our 32-token window, so lengthening the window alone cannot extend the model's effective memory.

8.5.3 Summary

Backpropagation through time applies ordinary backpropagation to an unrolled RNN, propagating loss sensitivities backward via Equation 8.21 and summing parameter gradients across time steps (Equation 8.23). Because a gradient traveling back k steps multiplies k state Jacobians, which reduces in a linear RNN to $(\mathbf{W}_{\text{hh}}^\top)^k \sim \rho^k$, gradients explode when $\rho > 1$ and vanish when $\rho < 1$. Including tanh multiplies each step by $1 - \mathbf{H}_t^2 \in (0, 1]$, causing gradients in our trained RNN to fall below 1% within roughly eight steps despite $\rho \approx 1.8$. Gradient clipping contains exploding spikes, truncated BPTT bounds memory and compute per update, and architectural innovations (Chapter 12) are required to preserve gradients over long horizons.

8.5.4 Exercises

1. **Backward pass by hand.** One step of `RNNScratch` (Section 8.4.2) computes

$$\mathbf{H}_t = \tanh(\mathbf{X}_t \mathbf{W}_{\text{xh}} + \mathbf{H}_{t-1} \mathbf{W}_{\text{hh}} + \mathbf{b}_{\text{h}}).$$

   Let $\mathbf{G}_t = \partial L / \partial \mathbf{H}_t$ be the gradient arriving from later steps and from the output at step t.

   1. Derive the contributions of this step to $\partial L / \partial \mathbf{W}_{\text{xh}}$, $\partial L / \partial \mathbf{W}_{\text{hh}}$, $\partial L / \partial \mathbf{b}_{\text{h}}$, and $\partial L / \partial \mathbf{H}_{t-1}$ as functions of $\mathbf{G}_t$, $\mathbf{X}_t$, $\mathbf{H}_t$, $\mathbf{H}_{t-1}$, and $\mathbf{W}_{\text{hh}}$.
   2. Adding the output contribution at step $t-1$ to the contribution to $\partial L / \partial \mathbf{H}_{t-1}$ gives a recursion for $\mathbf{G}_{t-1}$. Show that it reduces to Equation 8.21 when the tanh and the bias are dropped, and name the factor that the tanh adds at every step. Explain also how the parameter gradients accumulate over the T steps of a window, and compare with Equation 8.23.

   Adapted from CMU 11-785, Homework 3 Part 1.

2. **Power iteration and alignment.** Let $\mathbf{M} \in \mathbb{R}^{h \times h}$ be symmetric with eigenvalues λ_i and eigenvectors $\mathbf{v}_i$ for $i = 1, \dots, h$, ordered so that $|\lambda_i| \geq |\lambda_{i+1}|$.

   1. Show that $\mathbf{M}^k$ has eigenvalues λ_i^k with the same eigenvectors.
   2. Show that for a random vector $\mathbf{x} \in \mathbb{R}^h$ with independent standard normal entries, $\mathbf{M}^k \mathbf{x}$ becomes aligned with $\mathbf{v}_1$ as k grows. State the condition on the eigenvalues that this requires and the sense in which the alignment holds.
   3. What does this imply for the gradient Equation 8.22 of an RNN, beyond the change in its norm?

3. code **Backward pass in code.** ● Train `d2l.RNNLMScratch` on `d2l.TimeMachine` as in Section 8.4.2. Using the single-step gradients and the recursion of problem 1, implement the backward pass of its `RNNScratch` cell over a full window with tensor operations only, starting from the gradient of the loss with respect to the logits (Section 3.1.2.2). Run it on one training minibatch, and compare your gradients of `W_xh`, `W_hh`, and `b_h` with those computed by automatic differentiation.

   Adapted from CMU 11-785, Homework 3 Part 1.

4. code **Nonlinear Jacobians.** ● Draw a random symmetric 100×100 matrix, rescale it to spectral radius $\rho \in \{0.9, 1.0, 1.1\}$, and plot the norm of its k^{th} power $\|\mathbf{J}^k\|$ against k on a logarithmic axis: three straight lines, as the eigenvalues predict. Then leave the symmetric linear case.

   1. Replace the symmetric matrix by a general, nonsymmetric $\mathbf{W}_{\text{hh}}$ rescaled to each radius, and plot $\|\mathbf{J}^k\|$ against k. Report the largest value of $\|\mathbf{J}^k\|$ at $\rho = 0.9$ and the lag at which it occurs. Explain why the ratio $\|\mathbf{J}^k\| / \rho^k$ can exceed one and why it stays bounded as k grows (a random matrix is diagonalizable with probability one).
   2. Return to the symmetric $\mathbf{W}_{\text{hh}}$ and run the tanh RNN

$$\mathbf{H}_t = \tanh(\mathbf{H}_{t-1} \mathbf{W}_{\text{hh}} + \mathbf{X}_t), \qquad \mathbf{H}_0 = \mathbf{0}, \tag{8.24}$$

   for a single sequence ($\mathbf{H}_t \in \mathbb{R}^{1 \times h}$) with inputs $\mathbf{X}_t$ whose entries are drawn independently from $\mathcal{N}(0, s^2)$. Plot the norm of the product of the Jacobians $\partial \mathbf{H}_j / \partial \mathbf{H}_{j-1}$ along the trajectory for $s = 0.1$ and $s = 1$. How do the nonlinearity and the input scale change the growth of the norm, and why?

3. Explain why a largest singular value of $\mathbf{W}_{\text{hh}}$ below one is sufficient for the product to shrink, whereas a largest singular value above one is necessary but not sufficient for it to grow (Bengio et al. 1994; Pascanu et al. 2013). Relate the argument to your tanh curves at $\rho = 0.9$ and $\rho = 1.1$.

5. code **Window length and memory horizon.** ● In the model of Section 8.5.1, the gradient of the last loss falls below 1% of its size within about eight steps, well inside the training window of 32 tokens.

   1. Before running anything: if the model is retrained with `num_steps=128`, do you expect this horizon to move? Argue from the product of state Jacobians in Equation 8.22.
   2. Retrain with `num_steps` set to 128, measure the gradient against the lag on stretches of 256 validation tokens, and compare the result with your prediction.
   3. Repeat the loss-by-position measurement of Section 8.4.2 for the retrained model. Does the loss now keep improving beyond the third token of context?

8.6 Decoding and Generation

So far in this chapter, we have judged language models strictly by how well they assign probabilities to the next token. To *generate* text, however, a model must also select a concrete token at each step from its predicted distribution—and this decision matters far more than one might expect, because different selection rules cause the very same model to produce dramatically different prose. In Section 8.4.2, we saw the simplest rule fall short: greedily picking the single most probable token at every step rapidly traps the generator in repetitive loops. How, then, should we turn next-token distributions into fluent text?

The algorithm that converts a sequence of next-token distributions into output tokens is called *decoding* (Graves 2013; Holtzman et al. 2020), and decoding strategies fall into two broad families. *Maximization* algorithms search for an output sequence with high joint probability under the model, whereas *sampling* algorithms draw each token stochastically after reshaping or pruning the unreliable tail of the distribution. Below, we develop and compare both families in turn. Crucially, every decoding rule interacts with the language model solely through its next-token distribution, so the algorithms we build here apply unchanged to any autoregressive language model—whether an RNN, a state space model, or a Transformer.

8.6.1 The Decoding Problem

A trained language model supplies next-token conditional distributions: given any prefix $x_{<t} = x_1, \dots, x_{t-1}$, it outputs $P(x_t \mid x_{<t})$, and by the chain rule of probability (Section 8.1), these conditionals define the joint probability of any complete sequence via $P(x_1, \dots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_{<t})$.

Decoding poses the inverse problem: given those conditionals, which sequence should we emit? At first glance, finding the *most probable* sequence seems like the obvious goal, yet exact maximization is completely intractable. Over a vocabulary $\mathcal{V}$ and horizon T, there are $|\mathcal{V}|^T$ possible continuations—for our 1,027-token vocabulary and a 50-token continuation, $1027^{50} > 10^{150}$ candidates, far beyond anything we could ever enumerate. Every practical decoder must therefore explore only a tiny slice of the search tree, deciding at each step which prefixes to retain.

Even if exact search were feasible, the single highest-probability sequence is frequently *not* what we want. When a task admits only a narrow band of correct outputs, such as transcribing speech or translating a technical sentence, seeking a high-probability sequence works well. By contrast, when many continuations are equally valid—as in dialogue or creative writing—each individual continuation necessarily carries tiny probability mass, because a well-calibrated model spreads probability across many plausible trajectories. Forcing the model to pick the single most probable path destroys that natural variety, whereas sampling preserves it by drawing directly from the learned distribution.

Figure 8.12 organizes decoding strategies along this divide. Maximization methods (greedy decoding and beam search) remain the standard choice for constrained tasks with few valid outputs, while sampling methods augmented with temperature scaling or truncation are the default for open-ended generation.

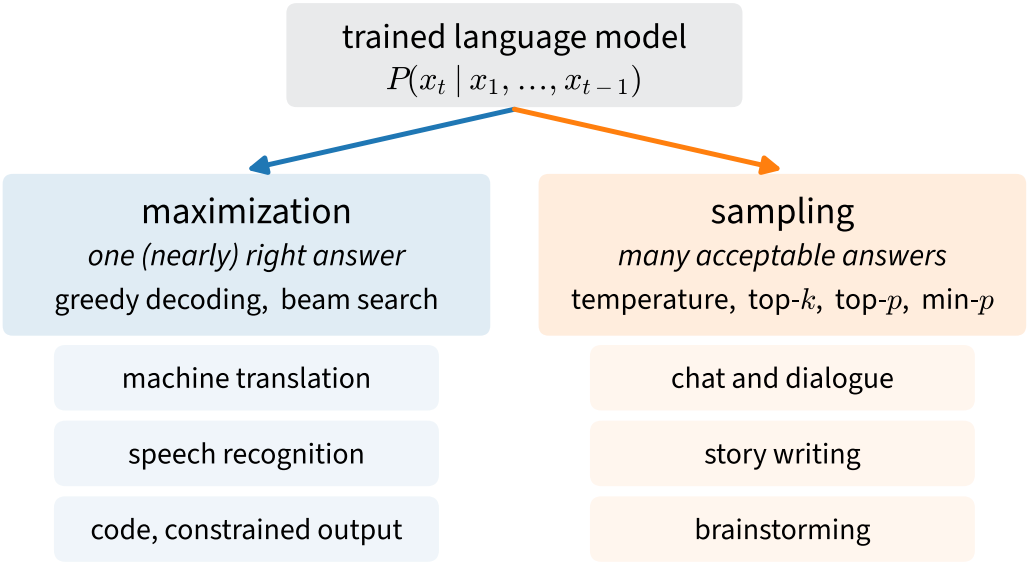


Figure 8.12: Two families of decoding rules. Searching for high-probability sequences suits tasks with few acceptable outputs; sampling preserves diversity in open-ended generation.

8.6.1.1 A Model to Decode From

For our experiments, we use the concise RNN language model from Section 8.4.6: 512-dimensional embeddings, 128 hidden units, dropout on both embeddings and hidden states, and an output layer over the 1,027 BPE tokens of *The Time Machine*. We begin by loading the 32-token training and validation windows from `d2l.TimeMachine`.

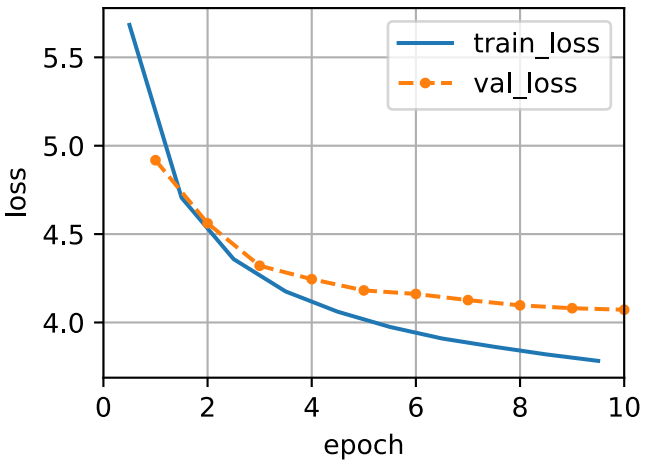
```
data = d2l.TimeMachine(batch_size=1024, num_steps=32)
```

Next, we construct the concise recurrent cell `d2l.RNN` and language model `d2l.RNNLM` with $d = 512$, $h = 128$, dropout, and learning rate `lr=4`.

```
rnn = d2l.RNN(num_inputs=512, num_hiddens=128)
model = d2l.RNNLM(rnn, vocab_size=len(data.vocab), lr=4, dropout=0.5,
                  emb_dropout=0.1)
```

Training `model` for 10 epochs with gradient clipping at `gradient_clip_val=1` reproduces a validation perplexity near 60 on *The Time Machine*.

```
trainer = d2l.Trainer(max_epochs=10, gradient_clip_val=1, num_gpus=1)
trainer.fit(model, data)
print(f'validation perplexity {math.exp(trainer.history["val_loss"][-1]):.1f}')
```




```
validation perplexity 58.7
```

Every decoding algorithm in this section accesses the language model through a single model-agnostic interface `step_fn(ids)` that takes the prefix token ids so far and returns a 1-D NumPy vector of next-token logits. Returning plain NumPy arrays lets us write every decoder once across all four frameworks, and omitting any explicit hidden state is deliberate: any autoregressive model, recurrent or Transformer, fits this exact signature.

```
model.eval()  # Dropout off

def seq_logits(ids):  # Token ids in, NumPy logits at every position out
    with torch.inference_mode():
        logits = model(d2l.tensor([ids], device=d2l.try_gpu()))
    return d2l.numpy(logits)[0]

def step_fn(ids):  # Token ids in, NumPy logits for the next token out
    return seq_logits(ids)[-1]
```

The trade-off is efficiency: each call re-evaluates the network across the *entire* prefix just to extract one vector of logits. For an RNN, that extra work is avoidable, and we return to its computational cost at the end of the section.

8.6.2 Greedy Decoding

The simplest possible decoder commits at every step to the single token with highest conditional probability:

$$x_t = \operatorname*{argmax}_{x \in \mathcal{V}} P(x \mid x_{<t}). \tag{8.25}$$

Score the prefix, pick the argmax, append it, and repeat. Greedy decoding requires just one model evaluation per generated token and is completely deterministic—the same prefix always yields the exact same continuation. The helper `show` below decodes token ids into a quoted string, displaying line breaks as `\n` and replacing any unprintable bytes with `U+FFFD`.

```
def show(ids):  # Decoded text as a literal; control characters -> U+FFFD
    text = data.tokenizer.decode(ids)
    return repr(''.join(c if c.isprintable() or c == '\n' else '\ufffd'
                        for c in text))

def decode_greedy(text, num_tokens=50):
    ids = data.tokenizer.encode(text)
    for _ in range(num_tokens):
        ids.append(int(step_fn(ids).argmax()))
    return ids

for text in ('The Time Traveller', 'It seemed to me that'):
    d2l.print_wrapped(show(decode_greedy(text)))
```

```
"The Time Traveller.\n\n'I had\nfrom the\nsidea.\n\n'I had\nfrom
    the\nsidea.\n\n'I had\nfrom the\nsidea.\n\n'I had\nf"
"It seemed to me that\ntheir.\n\n'I had\nfrom the\nsidea.\n\n'I had\nfrom
    the\nsidea.\n\n'I had\nfrom the\nsidea.\n\n'I"
```

Just as in Section 8.4.2, both continuations quickly collapse into repetitive loops. Two structural weaknesses are built into greedy decoding. The first is *myopia*: stringing together locally optimal choices does not necessarily yield the globally optimal sequence. Suppose the model assigns first-token probabilities $P(A) = 0.5$ and $P(B) = 0.4$, followed by an uncertain second step after A (top continuation probability 0.3) but a highly confident step after B (top probability 0.9). Greedy decoding immediately commits to A and finishes with joint probability $0.5 \times 0.3 = 0.15$, missing the far more probable sequence beginning with B at $0.4 \times 0.9 = 0.36$ (Figure 8.13).

The second weakness is repetition itself. Because greedy decoding feeds its own deterministic argmax back into the recurrence at every step, the next hidden state is a deterministic function of the current state; once the trajectory falls into an attracting cycle, there is no randomness to knock it out. Nor is this merely a quirk of our small RNN: Holtzman et al. (2020) demonstrated the exact same *neural text degeneration* in language models a thousand times larger, showing that likelihood maximization on open-ended text consistently yields repetitive, bland prose noticeably unlike human writing.

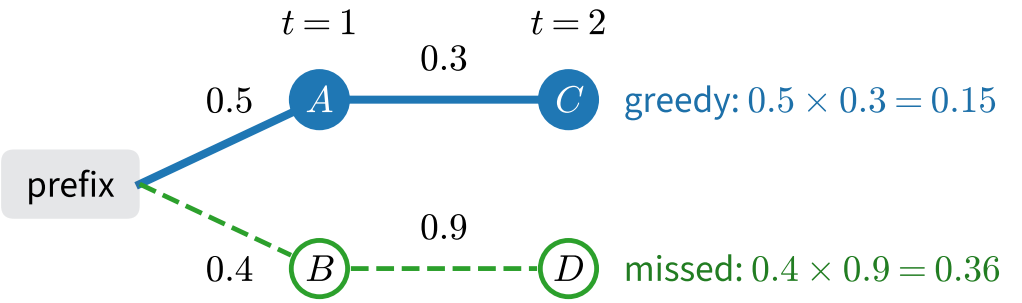


Figure 8.13: The myopia of greedy decoding on a two-step example. Greedy decoding takes the more probable first token A and ends with the sequence AC at joint probability $0.5 \times 0.3 = 0.15$. The path through B, which it never explores, reaches $0.4 \times 0.9 = 0.36$.

8.6.3 Beam Search

Whereas greedy decoding commits immediately to a single path and exhaustive search retains every possible path, *beam search* strikes a controlled middle ground parameterized by a single hyperparameter—the *beam width* w. At every step, beam search keeps the w highest-scoring prefixes alive. Starting from the w most probable first tokens, each subsequent step expands every surviving prefix across the vocabulary, scores all $w \cdot |\mathcal{V}|$ candidate extensions by their cumulative log-probability, and prunes back down to the top w sequences. Setting $w = 1$ recovers greedy decoding, while letting w grow large enough that no candidate is ever pruned recovers exhaustive search.

Figure 8.14 illustrates beam search with $w = 2$ over a five-token vocabulary $\mathcal{V} = \{A, B, C, D, E\}$ across three steps (suppressing the shared prompt from the notation). Suppose A and C are the two most probable first tokens. At step 2, we score all ten extensions $x' \in \mathcal{V}$,

$$P(A, x') = P(A)\, P(x' \mid A),$$
$$P(C, x') = P(C)\, P(x' \mid C),$$

and retain the top two, say AB and CE, repeating the same expansion at step 3 to keep ABD and CED. Rather than scoring all $5^3 = 125$ sequences, beam search evaluates only $5 + 10 + 10 = 25$. In general, generating T tokens costs $O(w|\mathcal{V}|T)$—just w times the cost of greedy decoding and exponentially less than exhaustive search.

When sequences are allowed to terminate early by emitting an end-of-sequence token, comparing candidates of different lengths requires a length correction. Because every token log-probability is non-positive, raw cumulative log-probability monotonically decreases as a sequence grows, unfairly favoring short hypotheses. The standard remedy is *length normalization* (Wu et al. 2016), scoring a candidate that adds T tokens to a prefix of length t_0 by

$$\frac{1}{T^{\beta}} \sum_{t=t_0+1}^{t_0+T} \log P(x_t \mid x_{<t}), \tag{8.26}$$

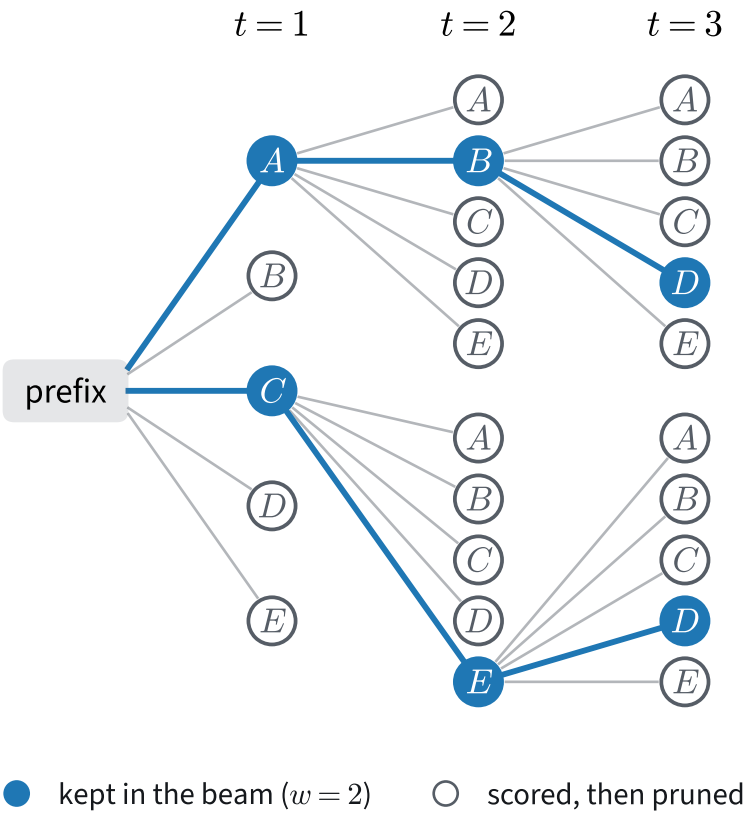


Figure 8.14: Beam search with beam width 2 over a five-token vocabulary. At every step each surviving prefix is expanded over the whole vocabulary and only the two highest-scoring prefixes are kept; the survivors here are A and C, then AB and CE, then ABD and CED.

where the exponent β (typically around 0.75) interpolates between unnormalized log-probability ($\beta = 0$) and per-token average log-probability ($\beta = 1$); in the code below, we name this argument `alpha` following Wu et al. (2016) .

The `beam_search` implementation below carries completed hypotheses (those ending in `eos_id`) forward untouched while expanding only active beams. Note one clean optimization: because the overall top w candidates can include at most w extensions of any single prefix, we only need to take the top w next tokens from each active beam rather than sorting all $|\mathcal{V}|$ extensions.

```
def beam_search(step_fn, prefix, num_tokens, beam_size=4,
                alpha=0.75, eos_id=None):
    """Length-normalized beam search (score = log P / len^alpha).
    Returns candidates sorted best-first; [0][1] is the argmax sequence."""
    def log_softmax(logits):
        logits = [float(l) for l in logits]
        m = max(logits)
        lse = m + math.log(sum(math.exp(l - m) for l in logits))
        return [l - lse for l in logits]
    score = lambda logp, ids: logp / max(1, len(ids) - len(prefix))**alpha
    beams = [(0.0, list(prefix), False)]  # (log-prob, sequence, finished)
    for _ in range(num_tokens):
        if all(done for _, _, done in beams):
            break
        candidates = [b for b in beams if b[2]]  # Finished pass through
        for logp, ids, _ in (b for b in beams if not b[2]):
            logprobs = log_softmax(step_fn(ids))
            best = sorted(range(len(logprobs)),
                          key=lambda i: -logprobs[i])[:beam_size]
            candidates += [(logp + logprobs[i], ids + [i], i == eos_id)
                           for i in best]
        beams = sorted(candidates, key=lambda b: score(b[0], b[1]),
                       reverse=True)[:beam_size]
    return sorted([(score(logp, ids), ids) for logp, ids, _ in beams],
                  reverse=True)
```

Let us decode the same two prompts with $w = 1$ (reproducing greedy decoding) and $w = 4$, printing the length-normalized score Equation 8.26 of the top hypothesis. Since our corpus has no end-of-sequence token, all candidates reach length 30 and the constant denominator $30^{0.75} \approx 12.8$ leaves their relative ranking unchanged.

```
for text in ('The Time Traveller', 'It seemed to me that'):
    prefix = data.tokenizer.encode(text)
    for w in (1, 4):
        score, ids = beam_search(step_fn, prefix, 30, beam_size=w)[0]
        d2l.print_wrapped(f'w={w}, score {score:.2f}: {show(ids)}')
```

```
w=1, score -4.13: "The Time Traveller.\n\n'I had\nfrom the\nsidea.\n\n'I
    had\nfrom the\nsidea.\n"
w=4, score -3.43: "The Time Traveller.\n\n'Then I had\ndeliculture.\n\n'Then
    I had\ndeliculture.\n"
w=1, score -4.21: "It seemed to me that\ntheir.\n\n'I had\nfrom
    the\nsidea.\n\n'I had\nfrom the\nside"
```

```
w=4, score -3.60: "It seemed to me that it was a\npossibly.\n\n'As I saw
    a\npossibly.\n\n'As"
```

Expanding the beam generally discovers higher-probability continuations than greedy decoding—though not with certainty, since approximate pruning can still discard a prefix that would have paid off later. Yet higher probability does not automatically equate to *better* prose: on open-ended generation, beam search in large language models produces text markedly more repetitive than human writing (Holtzman et al. 2020), a contrast we quantify shortly. By contrast, on tightly constrained tasks such as speech recognition and machine translation, beam search with a modest width (typically 4 to 10) remains the standard decoder (Sutskever et al. 2014; Wu et al. 2016).

8.6.4 Sampling

If our model has learned a reasonable distribution over continuations, the most natural alternative to maximization is direct *ancestral sampling*: draw $x_t \sim P(x_t \mid x_{<t})$, append it, and repeat. Sampling restores the rich variety that maximization suppresses, but raw sampling runs straight into a tail problem. Because the softmax assigns nonzero probability to every entry in the vocabulary (a thousand tokens here, and over a hundred thousand in modern LLMs), the tiny probabilities across thousands of implausible tokens add up to a substantial collective mass at every step. Consequently, pure sampling regularly draws a wildly implausible token, and feeding that token back in as context can derail the rest of the passage—the exact rollout error compounding discussed in Section 8.1. Holtzman et al. (2020) termed this the *unreliable tail*: a language model's probabilities are well calibrated near the head of the distribution, yet noisy far down the tail. The practical remedy is to reshape or truncate the distribution before sampling, and four widely used controls accomplish this in distinct ways.

Temperature scaling divides every logit o_x by a positive scalar $\tau > 0$ prior to the softmax:

$$P_\tau(x \mid x_{<t}) = \frac{\exp(o_x/\tau)}{\sum_{x' \in \mathcal{V}} \exp(o_{x'}/\tau)} \propto P(x \mid x_{<t})^{1/\tau}. \tag{8.27}$$

Lowering τ below 1 sharpens the distribution (converging to greedy decoding as $\tau \to 0$), whereas raising τ above 1 flattens it toward uniform. Because temperature rescales the entire distribution uniformly, however, it is a blunt instrument: cooling the softmax enough to suppress the unreliable tail simultaneously collapses legitimate diversity among plausible head tokens.

Top-k sampling truncates by rank, keeping only the k most probable tokens, renormalizing, and sampling (Fan et al. 2018; Radford et al. 2019). **Top-p** (or *nucleus*) sampling truncates by cumulative probability mass instead, retaining the smallest head of tokens whose probabilities sum to at least p (commonly $p = 0.9$) (Holtzman et al. 2020). This distinction matters because the model's entropy fluctuates dramatically from token to token: when the next word is nearly certain, a fixed $k = 20$ lets in nineteen poor candidates, whereas when dozens of words are equally natural, $k = 20$ may clip genuine variety. Top-p adapts dynamically, shrinking its candidate set on peaked distributions and widening it on flat ones, though it can still behave overly greedily when a single top token exceeds p (e.g., probability 0.95 under $p = 0.9$, excluding a solid runner-up).

Min-p sampling sets the threshold relative to the model's top probability (Nguyen et al. 2025), retaining every token whose probability is at least p_{min} times that of the single most

likely token (for instance, $p_{\min} = 0.05$). When the model is confident, the cutoff rises automatically and admits only strong contenders; when the model is uncertain, the cutoff lowers to allow broader exploration. Nguyen et al. (2025) show that this relative threshold degrades far more gracefully than top-p at elevated temperatures, where an absolute mass threshold sweeps in much of the flattened tail.

8.6.4.1 A Unified Sampler

Because temperature, top-k, top-p, and min-p all operate on the exact same next-token distribution, a single compact function handles every combination. Below, `sample_next` returns the greedy argmax when `temperature == 0`, and otherwise forms P_τ via Equation 8.27, sorts tokens in descending probability order, computes how many leading tokens each active cutoff permits, keeps the tightest bound, renormalizes, and draws one token using a NumPy random generator `rng`.

```
def sample_next(logits, temperature=1.0, k=None, p=None, min_p=None,
                rng=None):
    """Draw a token id from 1-D NumPy logits; temperature 0 is greedy.
    Top-k, top-p, and min-p truncate the temperature-scaled distribution."""
    logits = np.asarray(logits, dtype=np.float64)
    if temperature == 0:
        return int(logits.argmax())
    probs = np.exp((logits - logits.max()) / temperature)
    order = np.argsort(-probs)              # token ids, most probable first
    probs = probs[order] / probs.sum()
    keep = len(probs)
    if k is not None:                       # top-k: a fixed count
        keep = min(keep, k)
    if p is not None:                       # top-p: smallest head of mass p
        keep = min(keep, int(np.searchsorted(probs.cumsum(), p)) + 1)
    if min_p is not None:                   # min-p: a fraction of the top
        keep = min(keep, int((probs >= min_p * probs[0]).sum()))
    rng = np.random.default_rng() if rng is None else rng
    return int(order[rng.choice(keep, p=probs[:keep] / probs[:keep].sum())])
```

The wrapper `generate` packages the autoregressive decoding loop: call `step_fn`, draw the next token with `sample_next` (forwarding any strategy keyword arguments), append it, and terminate early if `eos_id` is emitted. Because open-ended prose from *The Time Machine* has no end-of-sequence token, our examples generate a fixed number of tokens.

```
def generate(step_fn, prefix, num_tokens, eos_id=None, **strategy):
    """Extend prefix by up to num_tokens ids, each chosen by sample_next
    from step_fn(ids), the next-token logits; stop early at eos_id."""
    ids = list(prefix)
    for _ in range(num_tokens):
        ids.append(sample_next(step_fn(ids), **strategy))
        if eos_id is not None and ids[-1] == eos_id:
            break
    return ids
```

A five-token toy distribution with probabilities $0.45, 0.25, 0.15, 0.10$, and 0.05 lets us verify each cutoff directly: top-k with $k = 2$ retains only tokens 0 and 1; top-p with $p = 0.8$ retains

tokens 0, 1, and 2 (whose cumulative mass reaches 0.85); and min-p with $p_{\text{min}} = 0.5$ retains tokens with probability at least $0.5 \times 0.45 = 0.225$, again tokens 0 and 1. Drawing 1,000 samples under each rule confirms the exact support.

```
logits = np.log([0.45, 0.25, 0.15, 0.10, 0.05])
print('greedy:', sample_next(logits, temperature=0))
for cutoff in (dict(k=2), dict(p=0.8), dict(min_p=0.5)):
    drawn = {sample_next(logits, **cutoff) for _ in range(1000)}
    print(cutoff, 'tokens drawn:', sorted(drawn))
```

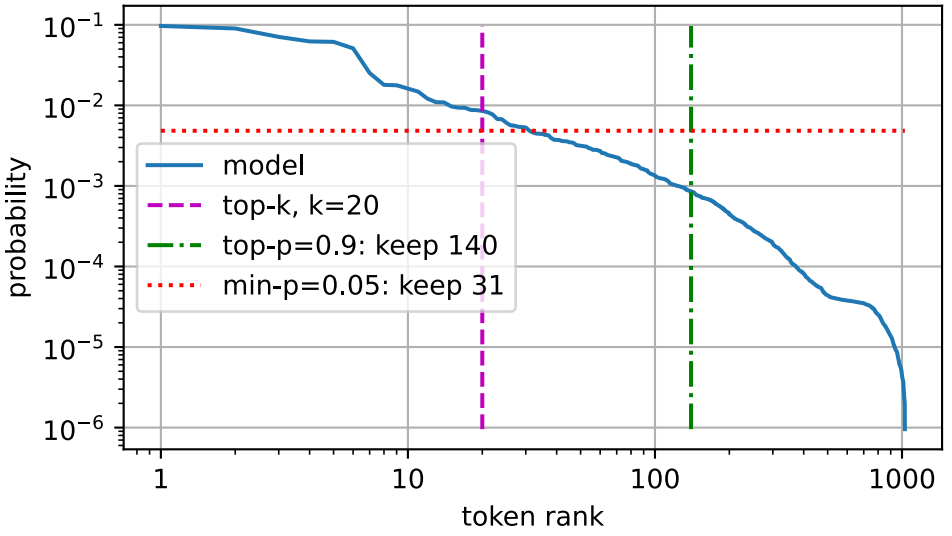
```
greedy: 0
{'k': 2} tokens drawn: [0, 1]
{'p': 0.8} tokens drawn: [0, 1, 2]
{'min_p': 0.5} tokens drawn: [0, 1]
```

As expected, each rule samples strictly from its surviving support: tokens 0 and 1 for top-k and min-p, and tokens 0 through 2 for top-p.

8.6.4.2 Cutoffs on a Trained Model

To see how these three rules behave on a real language model, let us plot our RNN's sorted next-token distribution following `"The Time Traveller"` on log–log axes. Viewed this way, each truncation rule has a clean geometric interpretation: top-k draws a fixed vertical line at rank k regardless of the curve's shape; top-p draws a vertical line at whichever rank accumulates probability mass p; and min-p draws a *horizontal* line at p_{min} times the top probability, rising and falling with the model's confidence. Everything to the right of a vertical cutoff, or beneath the horizontal cutoff, is pruned away.

```
def next_probs(text):  # Next-token probabilities after text, sorted
    logits = step_fn(data.tokenizer.encode(text))
    probs = np.exp(logits - logits.max())
    return np.sort(probs / probs.sum())[::-1]

def cutoffs(probs, p=0.9, min_p=0.05):  # Tokens kept by top-p and min-p
    return (int(np.searchsorted(probs.cumsum(), p)) + 1,
            int((probs >= min_p * probs[0]).sum()))

probs = next_probs('The Time Traveller')
n_p, n_m = cutoffs(probs)
lo, hi = probs[-1], probs[0]
d2l.plot([np.arange(1, len(probs) + 1), [20, 20], [n_p, n_p],
          [1, len(probs)]], [probs, [lo, hi], [lo, hi], [0.05 * hi] * 2],
         xlabel='token rank', ylabel='probability', xscale='log',
         yscale='log', figsize=(5.5, 3),
         legend=['model', 'top-k, k=20', f'top-p=0.9: keep {n_p}',
                 f'min-p=0.05: keep {n_m}'])
```



On log–log axes, the sorted probabilities fall steadily from between 0.1 and 0.25 at rank 1 down to about 10^{-3} near rank 100 before plunging more steeply across the remaining hundreds of tokens—the vast low-probability tail that truncation removes. To observe how adaptively each rule responds to model confidence, we compare `"The Time Traveller"` against the shorter prefix `"The Time"`, which is followed by `" Traveller"` in 11 of its 13 occurrences in the novella and therefore yields a sharply peaked distribution.

```
for text in ('The Time Traveller', 'The Time'):
    probs = next_probs(text)
    n_p, n_m = cutoffs(probs)
    print(f'{text!r}: top probability {probs[0]:.2f}; kept: top-k 20, '
          f'top-p {n_p}, min-p {n_m}')
```

```
'The Time Traveller': top probability 0.10; kept: top-k 20, top-p 140, min-p 31
'The Time': top probability 0.69; kept: top-k 20, top-p 6, min-p 2
```

While top-k rigidly retains 20 tokens regardless of context, both adaptive rules contract sharply once the model is confident: after `"The Time"`, where the top token dominates, both prune aggressively, with min-p retaining only a handful of candidates. When the model is uncertain after `"The Time Traveller"`, however, the two diverge: top-p must reach deep into the tail to accumulate 90% of the mass—admitting well over a hundred tokens, most individually below 1%—whereas min-p anchors solely on the leading candidate, retaining at most a few dozen tokens within a factor of twenty of the top probability.

8.6.4.3 Strategies Compared on One Prefix

Let us inspect the actual continuations produced by each strategy, starting with pure temperature scaling at $\tau = 0.5$, 1.0, and 2.0.

```
prefix = data.tokenizer.encode('The Time Traveller')
for tau in (0.5, 1.0, 2.0):
```

```
    out = generate(step_fn, prefix, 50, temperature=tau)
    d2l.print_wrapped(f'tau={tau}: {show(out)}')
```

```
tau=0.5: "The Time Traveller pale of the\ndelight of\nthe lamp and\npresent
    bushes.\n\n'I had\npresently\nmetinessed\nof a time. I felt\nwhich\n"
tau=1.0: "The Time Traveller.'\n\n'From the powersing solack, and silent it
    would by vanished! I came gave me to me.\n\n'I took with me up of
    them?ly dancing ru"
tau=2.0: 'The Time Traveller went co tsprockated d\ufffdir can\ufffd stoelt-lice slow
    against\nich\ufffdined to hished theirsentantworld in dararact disItvesj\ufffd
    perllow\nWild5ise?3 seemed'
```

At $\tau = 0.5$, the sampler stays conservatively close to high-probability tokens; at $\tau = 1.0$, outputs become noticeably more varied and occasionally slip into garbled words; and at $\tau = 2.0$, flattening the distribution lets tail tokens overwhelm the stream, breaking out of English and even emitting byte sequences that fail to form valid UTF-8 characters. Next, we compare our main decoding strategies head to head on the same prompt at their standard settings.

```
strategies = {'greedy': dict(temperature=0), 'tau=1': dict(),
              'top-k=20': dict(k=20), 'top-p=0.9': dict(p=0.9),
              'min-p=0.05': dict(min_p=0.05)}
for name, strategy in strategies.items():
    out = generate(step_fn, prefix, 50, **strategy)
    d2l.print_wrapped(f'{name:>10}: {show(out)}')
```

```
    greedy: "The Time Traveller.\n\n'I had\nfrom the\nsidea.\n\n'I had\nfrom
    the\nsidea.\n\n'I had\nfrom the\nsidea.\n\n'I had\nf"
     tau=1: 'The Time Traveller, and sgat; so like a sttyj dayshindting,
    posserent and buth on that Gecting it leve against\nbut7 Sa crying\ndark
    weight at first'
  top-k=20: "The Time Traveller looked round in these gone. His animals
    had\nsoon.\n\n'I wood to see some\nthe Psychologist in\nmank, and I
    directive, and my hand. I was in"
 top-p=0.9: 'The Time Traveller raraments. Then I thought "As of trift his
    ownisterse of the Palace of Gallen Proburned this white.\n\n\'Chen my an
    exton.\n'
min-p=0.05: "The Time Traveller came to pates of\nbit\nmels. Then\n'Then.
    As\nbefore a bigroud of these chambuge steading\ninsmination. I"
```

Naturally, a small single-layer RNN trained on one novella cannot produce flawless prose under any decoder, and a single sample per rule is noisy evidence. Even so, the qualitative contrast is unmistakable: greedy decoding is deterministic and falls into a repeating cycle, whereas every sampler produces fresh, non-looping text on every call. In practice, combining a moderate temperature with top-p or min-p truncation balances fluency and diversity well and serves as the default for open-ended generation.

8.6.5 Evaluation and Cost

8.6.5.1 The Quality of Generated Text

Beyond eyeballing individual passages, how can we compare decoding strategies systematically? Perplexity cannot answer this question: perplexity evaluates the *model* on held-out human text, not a particular decoding rule, and scoring generated samples by their own model likelihood would tautologically crown maximization the winner even though sampling deliberately trades peak likelihood for diversity.

Automated diagnostics can nonetheless pinpoint specific failure modes. To quantify repetition, we use *distinct-3*—the ratio of unique 3-grams to total 3-grams in a generated continuation, equal to 1 when no three-token span ever repeats and approaching 0 when text locks into a cycle. Below, we generate 100-token continuations from 20 random six-token validation prefixes under greedy decoding, beam search ($w = 4$), and each sampler, reporting both mean per-token log-probability under the model and mean distinct-3.

```
def log_prob(ids, start):  # Mean log-probability of ids[start:] in one pass
    logits = seq_logits(ids[:-1])[start - 1:]
    lse = logits.max(1) + np.log(np.exp(logits - logits.max(1, keepdims=True)).sum(1))
    return float(np.mean(logits[np.arange(len(logits)), ids[start:]] - lse))

def distinct(ids, n=3):  # Fraction of distinct n-grams in ids
    grams = [tuple(ids[i:i + n]) for i in range(len(ids) - n + 1)]
    return len(set(grams)) / len(grams)

val = d2l.numpy(data.X[data.num_train:data.num_train + data.num_val])
rows = np.random.default_rng().choice(len(val), 20, replace=False)
prefixes = [[int(i) for i in val[r, :6]] for r in rows]
decoders = {
    'greedy': lambda pre: generate(step_fn, pre, 100, temperature=0),
    'beam, w=4': lambda pre: beam_search(step_fn, pre, 100, beam_size=4)[0][1],
    'tau=1': lambda pre: generate(step_fn, pre, 100),
    'top-k=20': lambda pre: generate(step_fn, pre, 100, k=20),
    'top-p=0.9': lambda pre: generate(step_fn, pre, 100, p=0.9),
    'min-p=0.05': lambda pre: generate(step_fn, pre, 100, min_p=0.05)}
for name, decode in decoders.items():
    outs = [decode(pre) for pre in prefixes]
    print(f'{name:>10}: log-prob per token '
          f'{np.mean([log_prob(o, 6) for o in outs]):.2f}, distinct-3 '
          f'{np.mean([distinct(o[6:]) for o in outs]):.2f}')
```

```
    greedy: log-prob per token -1.89, distinct-3 0.15
 beam, w=4: log-prob per token -1.46, distinct-3 0.19
     tau=1: log-prob per token -3.93, distinct-3 0.99
  top-k=20: log-prob per token -2.83, distinct-3 0.99
 top-p=0.9: log-prob per token -3.38, distinct-3 0.99
min-p=0.05: log-prob per token -2.78, distinct-3 0.98
```

The two metrics pull cleanly in opposite directions. Both maximization strategies achieve markedly higher per-token log-probability than any sampler (with beam search typically edging out greedy decoding), yet their distinct-3 scores plunge because they repeat most of their 3-grams. All four samplers achieve distinct-3 near 1, reproducing on a miniature scale the exact likelihood-versus-diversity trade-off documented by Holtzman et al. (2020) .

Within the sampling family, diversity is nearly saturated across all four rules, while per-token log-probability cleanly separates them. Pure ancestral sampling ($\tau = 1$) incurs the lowest average log-probability (around -4 nats per token), whereas all three truncated samplers raise likelihood substantially without sacrificing distinct-3. Consistent with our earlier cutoff counts, top-p (0.9) sits closest to pure sampling because it retains the broadest support, while top-k (20) and min-p (0.05) prune more aggressively and boost per-token log-probability by roughly a full nat—demonstrating directly that tail truncation works as intended.

Automatic metrics like distinct-3 are valuable for catching concrete failure modes, yet they cannot measure coherence, factuality, or style. Ultimately, open-ended text generation is judged by human readers (or calibrated proxy evaluators), while constrained tasks with ground-truth references such as translation can be measured against reference outputs. In short, perplexity guides *model* selection, whereas human preference—or task-specific benchmarks—guides *decoder* selection.

8.6.5.2 The Cost of Generation

Because we kept `step_fn` stateless so that it could wrap any autoregressive model, every newly generated token re-runs the RNN over the *entire* prefix, making a T-token continuation cost $O(T^2)$ cell updates. Autoregressive generation is also inherently *sequential*: unlike teacher-forced training, where all positions are known up front, token $t + 1$ cannot be scored until token t has been chosen. While that sequential dependency is fundamental, the quadratic recomputation is completely unnecessary for an RNN: because the hidden state already summarizes the prefix, carrying that state forward from step to step (as `predict` did in Section 8.4.2) brings the cost down to exactly one cell update per token, independent of sequence length. Models without a recurrent state avoid the same recomputation by caching what they computed for each past token, at a memory cost that grows with the length of the prefix (Chapter 10).

Constant compute and constant memory per generated token are the defining efficiency advantages of recurrent language models. Yet that same fixed-size state is also our model's bottleneck, forcing everything remembered from the prefix through just 128 floating-point numbers (Section 8.4.1.1). No decoding trick can coax nuance out of a model whose next-token distributions fail to capture long-range structure—though because our decoders treat the model purely as a black box, every algorithm developed here transfers unchanged to far stronger models.

With this, Part *Basics* draws to a close—and it closes right on a sharp architectural boundary. Our RNN predicts text one token at a time at constant cost, yet it effectively uses only the last two or three tokens of history (Section 8.4.2), and with gradients that vanish within a handful of steps (Section 8.5) it has little chance of learning to use many more. Two paths lead forward from here. The first retains a fixed-size state while redesigning the recurrence so that gradients survive across long horizons (Chapter 12). The second abandons fixed-size compression altogether, retaining representations of every past token and attending directly across them at a cost that grows with sequence length—the *attention* mechanism (Chapter 10) at the heart of modern large language models. Whichever architecture lies underneath, the subword tokenizers, bits-per-byte evaluation, and decoding algorithms of this chapter carry directly over.

8.6.6 Summary

Decoding converts a language model's next-token distributions into concrete token sequences. Maximization methods seek high-probability outputs: greedy decoding commits locally at each step and suffers from both myopia and repetitive loops, while beam search tracks the top w hypotheses at w times the cost (using length normalization when candidates can terminate early) and excels on constrained tasks with narrow output spaces. For open-ended generation, sampling preserves natural diversity, while temperature scaling (Equation 8.27) and tail truncation via top-k, top-p, or min-p prevent low-probability tail tokens from derailing the rollout—producing text that matches pure sampling in distinct-3 diversity while remaining substantially more probable. Finally, perplexity evaluates models rather than decoders, and carrying a recurrent hidden state avoids the $O(T^2)$ recomputation of naive stateless decoding.

8.6.7 Exercises

1. **Beam search properties.**
    1. Take beam width 2, the five-token vocabulary of Figure 8.14, and three steps. Assign conditional probabilities for which beam search does not return the most probable three-token sequence, and compute the probabilities of both sequences. What does your construction imply about how beam search can miss the most probable sequence?
    2. What is the smallest beam width that guarantees that beam search over T steps returns the most probable of all $|\mathcal{V}|^T$ sequences, whatever the model?
    3. Let $w \leq |\mathcal{V}|$, and assume that no end-of-sequence token occurs. Count the calls to `step_fn` that `beam_search` makes over T steps. Count also the calls that exhaustive search makes when it evaluates every prefix once. Evaluate both counts for $|\mathcal{V}| = 1{,}027$, $T = 50$, and $w = 4$.
2. code **Length normalization.** Run `beam_search` with $w = 4$ for up to 40 tokens after `'The Time Traveller'` and `'It seemed to me that'`. Pass the id of the newline character, `data.tokenizer.encode('\n')[0]`, as `eos_id`, and vary β (the argument `alpha`) over $\{0, 0.75, 1.5\}$. Compare the lengths and the total log-probabilities of the winning candidates, and explain the trend using Equation 8.26.
3. **Truncation under temperature.** `sample_next` applies every cutoff to the temperature-scaled distribution P_τ of Equation 8.27. Write the min-p condition $P_\tau(x \mid x_{<t}) \geq p_{\text{min}} \max_{x'} P_\tau(x' \mid x_{<t})$ as a condition on the untempered probabilities. For the five-token distribution 0.45, 0.25, 0.15, 0.10, 0.05, compute the sets that top-k with $k = 2$, top-p with $p = 0.85$, and min-p with $p_{\text{min}} = 0.5$ keep at $\tau = 2$. Which of the three rules keeps the same set at every temperature?
4. code **Top-p versus min-p.** ● At $\tau = 1.5$, sweep p for top-p and p_{min} for min-p over a grid of values. For each setting, generate 50-token continuations of five prefixes of your choice and record two averages over the generated tokens: the number of tokens the rule keeps, and the log-probability of the drawn token under the untempered model. Plot the second average against the first, one curve per rule. Which rule draws more probable tokens at the same average number of kept tokens? Compare your finding with those of Nguyen et al. (2025) .
5. code **Constant cost per token.** ● `step_fn` re-runs the network over the entire prefix

for every generated token, so a T-token continuation costs $O(T^2)$ cell updates (Section 8.6.5.2).

1. Write a stateful replacement for `step_fn` that keeps the hidden state of `model.rnn` and consumes only the newest token, so that each call performs one cell update. Verify that `generate` produces the same greedy continuation with both versions.
2. Time `generate` for 50, 200, and 800 tokens with both versions. Estimate from the ratios how each time grows with the number of tokens, and explain any departure from the quadratic and linear growth that the count of cell updates predicts.
3. Why does the same replacement not carry over directly to `beam_search`, and what bookkeeping would make it work?

References

Abadi, Martín, Paul Barham, Jianmin Chen, et al. 2016. "TensorFlow: A System for Large-Scale Machine Learning." *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 265–83. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi.

Alsallakh, Bilal, Narine Kokhlikyan, Vivek Miglani, Jun Yuan, and Orion Reblitz-Richardson. 2020. "Mind the PAD – CNNs Can Develop Blind Spots." *ArXiv:2010.02178*. https://arxiv.org/abs/2010.02178.

Arora, Sanjeev, Zhiyuan Li, and Kaifeng Lyu. 2019. "Theoretical Analysis of Auto Rate-Tuning by Batch Normalization." *International Conference on Learning Representations*. https://arxiv.org/abs/1812.03981.

Arpit, Devansh, Stanisław Jastrzębski, Nicolas Ballas, et al. 2017. "A Closer Look at Memorization in Deep Networks." *International Conference on Machine Learning*, 233–42.

Ba, Jimmy Lei, Jamie Ryan Kiros, and Geoffrey E Hinton. 2016. "Layer Normalization." *ArXiv:1607.06450*. https://arxiv.org/abs/1607.06450.

Bachlechner, Thomas, Bodhisattwa Prasad Majumder, Huanru Henry Mao, Garrison W. Cottrell, and Julian McAuley. 2021. "ReZero Is All You Need: Fast Convergence at Large Depth." *Uncertainty in Artificial Intelligence*. https://arxiv.org/abs/2003.04887.

Balduzzi, David, Marcus Frean, Lennox Leary, J. P. Lewis, Kurt Wan-Duo Ma, and Brian McWilliams. 2017. "The Shattered Gradients Problem: If ResNets Are the Answer, Then What Is the Question?" *Proceedings of the 34th International Conference on Machine Learning*, Proceedings of machine learning research, vol. 70: 342–50.

Bay, Herbert, Tinne Tuytelaars, and Luc Van Gool. 2006. "SURF: Speeded up Robust Features." *European Conference on Computer Vision*, 404–17. https://doi.org/10.1007/11744023_32.

Bayes, Thomas. 1763. "An Essay Towards Solving a Problem in the Doctrine of Chances." *Philosophical Transactions of the Royal Society of London* 53: 370–418.

Belkin, Mikhail, Daniel Hsu, Siyuan Ma, and Soumik Mandal. 2019. "Reconciling Modern Machine-Learning Practice and the Classical Bias–Variance Trade-Off." *Proceedings of the National Academy of Sciences* 116 (32): 15849–54.

Bengio, Samy, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. 2015. "Scheduled Sampling for Sequence Prediction with Recurrent Neural Networks." *Advances in Neural Information Processing Systems* 28. https://arxiv.org/abs/1506.03099.

Bengio, Yoshua, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. 2003. "A Neural Probabilistic Language Model." *Journal of Machine Learning Research* 3 (Feb): 1137–55. https://jmlr.org/papers/v3/bengio03a.html.

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.

Beyer, Lucas, Olivier J. Hénaff, Alexander Kolesnikov, Xiaohua Zhai, and Aäron van den Oord. 2020. "Are We Done with ImageNet?" *ArXiv:2006.07159*. https://arxiv.org/abs/2006.07159.

Bishop, Chris M. 1995. "Training with Noise Is Equivalent to Tikhonov Regularization." *Neural Computation* 7 (1): 108–16. https://doi.org/10.1162/neco.1995.7.1.108.

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.

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.

Bradley, Ralph Allan, and Milton E Terry. 1952. "Rank Analysis of Incomplete Block Designs: I. The Method of Paired Comparisons." *Biometrika* 39 (3/4): 324–45. https://doi.org/10.1093/biomet/39.3-4.324.

Brock, Andrew, Soham De, Samuel L. Smith, and Karen Simonyan. 2021. "High-Performance Large-Scale Image Recognition Without Normalization." *International Conference on Machine Learning*. https://arxiv.org/abs/2102.06171.

Byrd, Jonathon, and Zachary C. Lipton. 2019. "What Is the Effect of Importance Weighting in Deep Learning?" *Proceedings of the 36th International Conference on Machine Learning*, Proceedings of machine learning research, vol. 97: 872–81.

Chen, Stanley F., and Joshua Goodman. 1999. "An Empirical Study of Smoothing Techniques for Language Modeling." *Computer Speech & Language* 13 (4): 359–93. https://doi.org/10.1006/csla.1999.0128.

Chen, Tianqi, Mu Li, Yutian Li, et al. 2015. "MXNET: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems." *ArXiv:1512.01274*. https://arxiv.org/abs/1512.01274.

Chen, Tianqi, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. "Training Deep Nets with Sublinear Memory Cost." *arXiv Preprint arXiv:1604.06174*.

Chollet, François. 2017. "Xception: Deep Learning with Depthwise Separable Convolutions." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. https://arxiv.org/abs/1610.02357.

Chu, Xiangxiang, Liang Li, and Bo Zhang. 2024. "Make RepVGG Greater Again: A Quantization-Aware Approach." *AAAI Conference on Artificial Intelligence*. https://arxiv.org/abs/2212.01593.

Cover, Thomas M., and Peter E. Hart. 1967. "Nearest Neighbor Pattern Classification." *IEEE Transactions on Information Theory* 13 (1): 21–27.

Cubuk, Ekin D., Barret Zoph, Jonathon Shlens, and Quoc V. Le. 2020. "RandAugment: Practical Automated Data Augmentation with a Reduced Search Space." *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*. https://arxiv.org/abs/1909.13719.

Cybenko, George. 1989. "Approximation by Superpositions of a Sigmoidal Function." *Mathematics of Control, Signals and Systems* 2 (4): 303–14. https://doi.org/10.1007/BF02551274.

Dalal, Navneet, and Bill Triggs. 2005. "Histograms of Oriented Gradients for Human Detection." *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)* 1: 886–93. https://doi.org/10.1109/cvpr.2005.177.

De Cock, Dean. 2011. "Ames, Iowa: Alternative to the Boston Housing Data as an End of Semester Regression Project." *Journal of Statistics Education* 19 (3). https://doi.org/10.1080/10691898.2011.11889627.

De, Soham, and Samuel L. Smith. 2020. "Batch Normalization Biases Residual Blocks Towards the Identity Function in Deep Networks." *Advances in Neural Information Processing Systems* 33. https://arxiv.org/abs/2002.10444.

Deng, Jia, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. "Imagenet: A Large-Scale Hierarchical Image Database." *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 248–55. https://doi.org/10.1109/cvpr.2009.5206848.

Der Kiureghian, Armen, and Ove Ditlevsen. 2009. "Aleatory or Epistemic? Does It Matter?" *Structural Safety* 31 (2): 105–12. https://doi.org/10.1016/j.strusafe.2008.06.020.

Devlin, Jacob, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. "BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding." *ArXiv:1810.04805*. https://arxiv.org/abs/1810.04805.

Ding, Xiaohan, Xiangyu Zhang, Ningning Ma, Jungong Han, Guiguang Ding, and Jian Sun. 2021. "RepVGG: Making VGG-Style ConvNets Great Again." *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. https://arxiv.org/abs/2101.03697.

Ding, Xiaohan, Xiangyu Zhang, Yizhuang Zhou, Jungong Han, Guiguang Ding, and Jian Sun.

2022. "Scaling up Your Kernels to 31x31: Revisiting Large Kernel Design in CNNs." *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. https://arxiv.org/abs/2203.06717.

Dosovitskiy, Alexey, Lucas Beyer, Alexander Kolesnikov, et al. 2021. "An Image Is Worth 16 x 16 Words: Transformers for Image Recognition at Scale." *International Conference on Learning Representations*. https://openreview.net/forum?id=YicbFdNTTy.

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.

Elman, Jeffrey L. 1990. "Finding Structure in Time." *Cognitive Science* 14 (2): 179–211. https://doi.org/10.1016/0364-0213(90)90002-E.

Falcon, William, and The PyTorch Lightning team. 2019. *PyTorch Lightning*. https://doi.org/10.5281/zenodo.3828935.

Fan, Angela, Mike Lewis, and Yann Dauphin. 2018. "Hierarchical Neural Story Generation." *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*, 889–98.

Field, David J. 1987. "Relations Between the Statistics of Natural Images and the Response Properties of Cortical Cells." *JOSA A* 4 (12): 2379–94. https://doi.org/10.1364/josaa.4.002379.

Frostig, Roy, Matthew James Johnson, and Chris Leary. 2018. "Compiling Machine Learning Programs via High-Level Tracing." In *Proceedings of Systems for Machine Learning*. https://mlsys.org/Conferences/2018/doc/19.pdf.

Fukushima, Kunihiko. 1980. "Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position." *Biological Cybernetics* 36 (4): 193–202. https://doi.org/10.1007/BF00344251.

Gage, Philip. 1994. "A New Algorithm for Data Compression." *The C Users Journal* 12 (2): 23–38.

Gal, Yarin, and Zoubin Ghahramani. 2016. "Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning." *International Conference on Machine Learning*, 1050–59. https://arxiv.org/abs/1506.02142.

Gauss, Carl Friedrich. 1809. "Theoria Motus Corporum Coelestium." In *Werke*. Königlich Preussische Akademie der Wissenschaften. https://doi.org/10.1007/978-3-642-92478-1.

Girshick, Ross, Jeff Donahue, Trevor Darrell, and Jitendra Malik. 2014. "Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 580–87. https://doi.org/10.1109/CVPR.2014.81.

Glorot, Xavier, and Yoshua Bengio. 2010. "Understanding the Difficulty of Training Deep

Feedforward Neural Networks." *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics*, 249–56. https://proceedings.mlr.press/v9/glorot10a.html.

Glorot, Xavier, Antoine Bordes, and Yoshua Bengio. 2011. "Deep Sparse Rectifier Neural Networks." *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics*, 315–23. https://proceedings.mlr.press/v15/glorot11a.html.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press.

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.

Grattafiori, Aaron, Abhimanyu Dubey, Abhinav Jauhri, et al. 2024. "The Llama 3 Herd of Models." *arXiv Preprint arXiv:2407.21783*.

Graves, Alex. 2013. "Generating Sequences with Recurrent Neural Networks." *ArXiv:1308.0850*. https://arxiv.org/abs/1308.0850.

Grinsztajn, Léo, Edouard Oyallon, and Gaël Varoquaux. 2022. "Why Do Tree-Based Models Still Outperform Deep Learning on Tabular Data?" *Advances in Neural Information Processing Systems* 35: 507–20. https://arxiv.org/abs/2207.08815.

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.

Harris, Charles R., K. Jarrod Millman, Stéfan J. van der Walt, et al. 2020. "Array Programming with NumPy." *Nature* 585 (7825): 357–62. https://doi.org/10.1038/s41586-020-2649-2.

He, Kaiming, Xinlei Chen, Saining Xie, Yanghao Li, Piotr Dollár, and Ross Girshick. 2022. "Masked Autoencoders Are Scalable Vision Learners." *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 16000–16009. https://doi.org/10.1109/cvpr52688.2022.01553.

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.

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016b. "Identity Mappings in Deep Residual Networks." *European Conference on Computer Vision*, 630–45. https://doi.org/10.1007/978-3-319-46493-0_38.

He, Tong, Zhi Zhang, Hang Zhang, Zhongyue Zhang, Junyuan Xie, and Mu Li. 2019. "Bag of Tricks for Image Classification with Convolutional Neural Networks." *IEEE/CVF Confer-*

ence on Computer Vision and Pattern Recognition, 558–67. https://arxiv.org/abs/1812.01187.

Hendrycks, Dan, and Kevin Gimpel. 2016. "Gaussian Error Linear Units (GELUs)." *ArXiv:1606.08415*. https://arxiv.org/abs/1606.08415.

Hinton, Geoffrey E., Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov. 2012. "Improving Neural Networks by Preventing Co-Adaptation of Feature Detectors." *arXiv Preprint arXiv:1207.0580*.

Hoeffding, Wassily. 1963. "Probability Inequalities for Sums of Bounded Random Variables." *Journal of the American Statistical Association* 58 (301): 13–30.

Hoerl, Arthur E, and Robert W Kennard. 1970. "Ridge Regression: Biased Estimation for Nonorthogonal Problems." *Technometrics* 12 (1): 55–67.

Holtzman, Ari, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2020. "The Curious Case of Neural Text Degeneration." *International Conference on Learning Representations*. https://arxiv.org/abs/1904.09751.

Howard, Andrew G., Menglong Zhu, Bo Chen, et al. 2017. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications." *ArXiv:1704.04861*. https://arxiv.org/abs/1704.04861.

Hu, Jie, Li Shen, and Gang Sun. 2018. "Squeeze-and-Excitation Networks." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 7132–41. https://doi.org/10.1109/cvpr.2018.00745.

Huang, Gao, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. 2017. "Densely Connected Convolutional Networks." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 4700–4708. https://doi.org/10.1109/cvpr.2017.243.

Huang, Gao, Yu Sun, Zhuang Liu, Daniel Sedra, and Kilian Weinberger. 2016. "Deep Networks with Stochastic Depth." *European Conference on Computer Vision*. https://arxiv.org/abs/1603.09382.

Hubel, David H, and Torsten N Wiesel. 1959. "Receptive Fields of Single Neurones in the Cat's Striate Cortex." *Journal of Physiology* 148 (3): 574–91. https://doi.org/10.1113/jphysiol.1959.sp006308.

Hubel, David H, and Torsten N Wiesel. 1962. "Receptive Fields, Binocular Interaction and Functional Architecture in the Cat's Visual Cortex." *Journal of Physiology* 160 (1): 106–54. https://doi.org/10.1113/jphysiol.1962.sp006837.

Hubel, David H, and Torsten N Wiesel. 1968. "Receptive Fields and Functional Architecture of Monkey Striate Cortex." *Journal of Physiology* 195 (1): 215–43. https://doi.org/10.1113/jphysiol.1968.sp008455.

Inan, Hakan, Khashayar Khosravi, and Richard Socher. 2017. "Tying Word Vectors and Word Classifiers: A Loss Framework for Language Modeling." *International Conference*

on Learning Representations.

Ioffe, Sergey, and Christian Szegedy. 2015. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift." *International Conference on Machine Learning*, 448–56. https://arxiv.org/abs/1502.03167.

Izmailov, Pavel, Dmitrii Podoprikhin, Timur Garipov, Dmitry Vetrov, and Andrew Gordon Wilson. 2018. "Averaging Weights Leads to Wider Optima and Better Generalization." *Uncertainty in Artificial Intelligence*, 876–85. https://arxiv.org/abs/1803.05407.

Jaeger, Herbert. 2002. *Tutorial on Training Recurrent Neural Networks, Covering BPTT, RTRL, EKF and the "Echo State Network" Approach*. GMD-Forschungszentrum Informationstechnik Bonn.

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.

Kneser, Reinhard, and Hermann Ney. 1995. "Improved Backing-Off for M-Gram Language Modeling." *International Conference on Acoustics, Speech, and Signal Processing* 1: 181–84. https://doi.org/10.1109/ICASSP.1995.479394.

Koh, Pang Wei, Shiori Sagawa, Henrik Marklund, et al. 2021. "WILDS: A Benchmark of in-the-Wild Distribution Shifts." *International Conference on Machine Learning*, 5637–64. https://arxiv.org/abs/2012.07421.

Kohavi, Ron. 1995. "A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection." *International Joint Conference on Artificial Intelligence* 14: 1137–45.

Kolmogorov, Andrey. 1933. *Grundbegriffe Der Wahrscheinlichkeitsrechnung*. Springer. https://doi.org/10.1007/978-3-642-49888-6.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton. 2012. "ImageNet Classification with Deep Convolutional Neural Networks." *Advances in Neural Information Processing Systems*, 1097–105. https://doi.org/10.5555/2999134.2999257.

Kuzovkin, Ilya, Raul Vicente, Mathilde Petton, et al. 2018. "Activations of Deep Convolutional Neural Networks Are Aligned with Gamma Band Activity of Human Visual Cortex." *Communications Biology* 1 (1): 1–12. https://doi.org/10.1038/s42003-018-0110-y.

LeCun, Yann, Bernhard Boser, John S Denker, et al. 1989. "Backpropagation Applied to Handwritten Zip Code Recognition." *Neural Computation* 1 (4): 541–51. https://doi.org/10.1162/neco.1989.1.4.541.

LeCun, Yann, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. "Gradient-Based Learning Applied to Document Recognition." *Proceedings of the IEEE* 86 (11): 2278–324. https://doi.org/10.1109/5.726791.

LeCun, Yann, LD Jackel, Leon Bottou, et al. 1995. "Comparison of Learning Algorithms for Handwritten Digit Recognition." *International Conference on Artificial Neural Networks*,

53–60.

Legendre, Adrien Marie. 1805. *Nouvelles méthodes Pour La détermination Des Orbites Des Comètes*. Courcier.

Leshno, Moshe, Vladimir Ya. Lin, Allan Pinkus, and Shimon Schocken. 1993. "Multilayer Feedforward Networks with a Nonpolynomial Activation Function Can Approximate Any Function." *Neural Networks* 6 (6): 861–67.

Li, Hao, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. 2018. "Visualizing the Loss Landscape of Neural Nets." *Advances in Neural Information Processing Systems* 31: 6389–99.

Li, Xiang, Shuo Chen, Xiaolin Hu, and Jian Yang. 2019. "Understanding the Disharmony Between Dropout and Batch Normalization by Variance Shift." *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2682–90. https://arxiv.org/abs/1801.05134.

Lin, Min, Qiang Chen, and Shuicheng Yan. 2013. "Network in Network." *ArXiv:1312.4400*. https://arxiv.org/abs/1312.4400.

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.

Lipton, Zachary, Yu-Xiang Wang, and Alexander Smola. 2018. "Detecting and Correcting for Label Shift with Black Box Predictors." *International Conference on Machine Learning*, 3122–30. https://arxiv.org/abs/1802.03916.

Liu, Hanxiao, Karen Simonyan, and Yiming Yang. 2018. "DARTS: Differentiable Architecture Search." *ArXiv:1806.09055*. https://arxiv.org/abs/1806.09055.

Liu, Ze, Yutong Lin, Yue Cao, et al. 2021. "Swin Transformer: Hierarchical Vision Transformer Using Shifted Windows." *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 10012–22. https://doi.org/10.1109/iccv48922.2021.00986.

Liu, Zhuang, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhofer, Trevor Darrell, and Saining Xie. 2022. "A ConvNet for the 2020s." *ArXiv:2201.03545*. https://arxiv.org/abs/2201.03545.

Long, Jonathan, Evan Shelhamer, and Trevor Darrell. 2015. "Fully Convolutional Networks for Semantic Segmentation." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 3431–40. https://doi.org/10.1109/CVPR.2015.7298965.

Lopez-Paz, David, and Maxime Oquab. 2017. "Revisiting Classifier Two-Sample Tests." *International Conference on Learning Representations*.

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.

Lowe, David G. 2004. "Distinctive Image Features from Scale-Invariant Keypoints." *International Journal of Computer Vision* 60 (2): 91–110. https://doi.org/10.1023/b:visi.0000029664.99615.94.

Luo, Ping, Xinjiang Wang, Wenqi Shao, and Zhanglin Peng. 2018. "Towards Understanding Regularization in Batch Normalization." *ArXiv:1809.00846*. https://arxiv.org/abs/1809.00846.

Luo, Wenjie, Yujia Li, Raquel Urtasun, and Richard Zemel. 2016. "Understanding the Effective Receptive Field in Deep Convolutional Neural Networks." *Advances in Neural Information Processing Systems*. https://arxiv.org/abs/1701.04128.

Markowitz, Harry. 1952. "Portfolio Selection." *The Journal of Finance* 7 (1): 77–91.

McCulloch, Warren S, and Walter Pitts. 1943. "A Logical Calculus of the Ideas Immanent in Nervous Activity." *Bulletin of Mathematical Biophysics* 5 (4): 115–33. https://doi.org/10.1016/s0092-8240(05)80006-0.

McKinney, Wes. 2010. "Data Structures for Statistical Computing in Python." *Proceedings of the 9th Python in Science Conference*, 56–61. https://doi.org/10.25080/Majora-92bf1922-00a.

Micikevicius, Paulius, Sharan Narang, Jonah Alben, et al. 2018. "Mixed Precision Training." *International Conference on Learning Representations*.

Mikolov, Tomáš, Stefan Kombrink, Lukáš Burget, Jan Černocký, and Sanjeev Khudanpur. 2011. "Extensions of Recurrent Neural Network Language Model." *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 5528–31. https://doi.org/10.1109/ICASSP.2011.5947611.

Mishkin, Dmytro, and Jiri Matas. 2016. "All You Need Is a Good Init." *International Conference on Learning Representations*. https://arxiv.org/abs/1511.06422.

Moon, Taesup, Alex Smola, Yi Chang, and Zhaohui Zheng. 2010. "Intervalrank: Isotonic Regression with Listwise and Pairwise Constraints." *Proceedings of the 3rd ACM International Conference on Web Search and Data Mining*, 151–60. https://doi.org/10.1145/1718487.1718520.

Nair, Vinod, and Geoffrey E Hinton. 2010. "Rectified Linear Units Improve Restricted Boltzmann Machines." *ICML*, 807–14. https://dl.acm.org/doi/10.5555/3104322.3104425.

Ney, Hermann, Ute Essen, and Reinhard Kneser. 1994. "On Structuring Probabilistic Dependences in Stochastic Language Modelling." *Computer Speech & Language* 8 (1): 1–38. https://doi.org/10.1006/csla.1994.1001.

Nguyen, Minh Nhat, Andrew Baker, Clement Neo, Allen Roush, Andreas Kirsch, and Ravid Shwartz-Ziv. 2025. "Turning up the Heat: Min-p Sampling for Creative and Coherent

LLM Outputs." *International Conference on Learning Representations*. https://arxiv.org/abs/2407.01082.

Pascanu, Razvan, Tomas Mikolov, and Yoshua Bengio. 2013. "On the Difficulty of Training Recurrent Neural Networks." *International Conference on Machine Learning*, 1310–18.

Paszke, Adam, Sam Gross, Francisco Massa, et al. 2019. "PyTorch: An Imperative Style, High-Performance Deep Learning Library." *Advances in Neural Information Processing Systems* 32: 8026–37. https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

Peters, Jonas, Dominik Janzing, and Bernhard Schölkopf. 2017. *Elements of Causal Inference: Foundations and Learning Algorithms*. MIT Press. https://doi.org/10.7551/mitpress/11283.001.0001.

Pleiss, Geoff, Danlu Chen, Gao Huang, Tongcheng Li, Laurens Van Der Maaten, and Kilian Q Weinberger. 2017. "Memory-Efficient Implementation of Densenets." *ArXiv:1707.06990*. https://arxiv.org/abs/1707.06990.

Power, Alethea, Yuri Burda, Harri Edwards, Igor Babuschkin, and Vedant Misra. 2022. "Grokking: Generalization Beyond Overfitting on Small Algorithmic Datasets." *ArXiv:2201.02177*. https://arxiv.org/abs/2201.02177.

Press, Ofir, and Lior Wolf. 2017. "Using the Output Embedding to Improve Language Models." *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics*, 157–63.

Radford, Alec, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. "Language Models Are Unsupervised Multitask Learners." *OpenAI Blog* 1 (8): 9. https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf.

Radosavovic, Ilija, Justin Johnson, Saining Xie, Wan-Yen Lo, and Piotr Dollár. 2019. "On Network Design Spaces for Visual Recognition." *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 1882–90. https://doi.org/10.1109/iccv.2019.00052.

Radosavovic, Ilija, Raj Prateek Kosaraju, Ross Girshick, Kaiming He, and Piotr Dollár. 2020. "Designing Network Design Spaces." *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 10428–36. https://doi.org/10.1109/cvpr42600.2020.01044.

Rahimi, Ali, and Benjamin Recht. 2007. "Random Features for Large-Scale Kernel Machines." *Advances in Neural Information Processing Systems* 20.

Ranzato, Marc'Aurelio, Sumit Chopra, Michael Auli, and Wojciech Zaremba. 2016. "Sequence Level Training with Recurrent Neural Networks." *International Conference on Learning Representations*. https://arxiv.org/abs/1511.06732.

Recht, Benjamin, Rebecca Roelofs, Ludwig Schmidt, and Vaishaal Shankar. 2019. "Do ImageNet Classifiers Generalize to ImageNet?" *International Conference on Machine Learning*, 5389–400. https://arxiv.org/abs/1902.10811.

Riesenhuber, Maximilian, and Tomaso Poggio. 1999. "Hierarchical Models of Object Recognition in Cortex." *Nature Neuroscience* 2 (11): 1019–25. https://doi.org/10.1038/14819.

Ronneberger, Olaf, Philipp Fischer, and Thomas Brox. 2015. "U-Net: Convolutional Networks for Biomedical Image Segmentation." *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, Lecture notes in computer science, vol. 9351: 234–41. https://arxiv.org/abs/1505.04597.

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.

Russakovsky, Olga, Jia Deng, Hao Su, et al. 2015. "ImageNet Large Scale Visual Recognition Challenge." *International Journal of Computer Vision* 115 (3): 211–52. https://doi.org/10.1007/s11263-015-0816-y.

Saerens, Marco, Patrice Latinne, and Christine Decaestecker. 2002. "Adjusting the Outputs of a Classifier to New a Priori Probabilities: A Simple Procedure." *Neural Computation* 14 (1): 21–41. https://doi.org/10.1162/089976602753284446.

Sandler, Mark, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. "MobileNetV2: Inverted Residuals and Linear Bottlenecks." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. https://arxiv.org/abs/1801.04381.

Santurkar, Shibani, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. 2018. "How Does Batch Normalization Help Optimization?" *Advances in Neural Information Processing Systems*, 2483–93. https://doi.org/10.5555/3327345.3327508.

Sennrich, Rico, Barry Haddow, and Alexandra Birch. 2016. "Neural Machine Translation of Rare Words with Subword Units." *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, 1715–25. https://doi.org/10.18653/v1/P16-1162.

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.

Shimodaira, Hidetoshi. 2000. "Improving Predictive Inference Under Covariate Shift by Weighting the Log-Likelihood Function." *Journal of Statistical Planning and Inference* 90 (2): 227–44.

Shwartz-Ziv, Ravid, and Amitai Armon. 2022. "Tabular Data: Deep Learning Is Not All You Need." *Information Fusion* 81: 84–90. https://doi.org/10.1016/j.inffus.2021.11.011.

Simonyan, Karen, and Andrew Zisserman. 2015. "Very Deep Convolutional Networks for Large-Scale Image Recognition." *International Conference on Learning Representations*. https://arxiv.org/abs/1409.1556.

Sivic, Josef, and Andrew Zisserman. 2003. "Video Google: A Text Retrieval Approach to Object Matching in Videos." *Proceedings of the IEEE International Conference on Computer Vision* 3: 1470–77. https://doi.org/10.1109/iccv.2003.1238663.

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.

Srivastava, Nitish, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting." *Journal of Machine Learning Research* 15 (1): 1929–58. https://doi.org/10.5555/2627435.2670313.

Sutskever, Ilya, Oriol Vinyals, and Quoc V Le. 2014. "Sequence to Sequence Learning with Neural Networks." *Advances in Neural Information Processing Systems*, 3104–12. https://doi.org/10.5555/2969033.2969173.

Szegedy, Christian, Wei Liu, Yangqing Jia, et al. 2015. "Going Deeper with Convolutions." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 1–9. https://doi.org/10.1109/cvpr.2015.7298594.

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.

Tan, Mingxing, and Quoc Le. 2019. "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks." *International Conference on Machine Learning*, 6105–14. https://proceedings.mlr.press/v97/tan19a.html.

Tan, Mingxing, and Quoc V. Le. 2021. "EfficientNetV2: Smaller Models and Faster Training." *International Conference on Machine Learning*. https://arxiv.org/abs/2104.00298.

Telgarsky, Matus. 2016. "Benefits of Depth in Neural Networks." *Conference on Learning Theory*, 1517–39.

Tibshirani, Robert. 1996. "Regression Shrinkage and Selection via the Lasso." *Journal of the Royal Statistical Society: Series B* 58 (1): 267–88.

Tikhonov, A. N., and V. Y. Arsenin. 1977. *Solutions of Ill-Posed Problems*. W.H. Winston. https://doi.org/10.1137/1.9780898719741.

Touvron, Hugo, Matthieu Cord, Alexandre Sablayrolles, Gabriel Synnaeve, and Hervé Jégou. 2021. "Going Deeper with Image Transformers." *IEEE/CVF International Conference on Computer Vision*, 32–42. https://arxiv.org/abs/2103.17239.

Tsoumakas, Grigorios, and Ioannis Katakis. 2007. "Multi-Label Classification: An Overview." *International Journal of Data Warehousing and Mining* 3 (3): 1–13. https://doi.org/10.4018/jdwm.2007070101.

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.

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.

Waibel, Alex, Toshiyuki Hanazawa, Geoffrey Hinton, Kiyohiro Shikano, and Kevin J Lang. 1989. "Phoneme Recognition Using Time-Delay Neural Networks." *IEEE Transactions on Acoustics, Speech, and Signal Processing* 37 (3): 328–39. https://doi.org/10.1109/29.21701.

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.

Werbos, Paul J. 1990. "Backpropagation Through Time: What It Does and How to Do It." *Proceedings of the IEEE* 78 (10): 1550–60. https://doi.org/10.1109/5.58337.

Wightman, Ross, Hugo Touvron, and Hervé Jégou. 2021. "ResNet Strikes Back: An Improved Training Procedure in Timm." *ArXiv:2110.00476*. https://arxiv.org/abs/2110.00476.

Wolpert, David H. 1996. "The Lack of a Priori Distinctions Between Learning Algorithms." *Neural Computation* 8 (7): 1341–90.

Woo, Sanghyun, Shoubhik Debnath, Ronghang Hu, et al. 2023. "ConvNeXt V2: Co-Designing and Scaling ConvNets with Masked Autoencoders." *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. https://arxiv.org/abs/2301.00808.

Wu, Yonghui, Mike Schuster, Zhifeng Chen, et al. 2016. "Google's Neural Machine Translation System: Bridging the Gap Between Human and Machine Translation." *ArXiv:1609.08144*. https://arxiv.org/abs/1609.08144.

Wu, Yuxin, and Kaiming He. 2018. "Group Normalization." *European Conference on Computer Vision*. https://arxiv.org/abs/1803.08494.

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, Saining, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. 2017. "Aggregated Residual Transformations for Deep Neural Networks." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 1492–500. https://doi.org/10.1109/cvpr.2017.634.

Yamaguchi, Kouichi, Kenji Sakamoto, Toshio Akabane, and Yoshiji Fujimoto. 1990. "A Neural Network for Speaker-Independent Isolated Word Recognition." *First International Conference on Spoken Language Processing*. https://doi.org/10.21437/icslp.1990-282.

Yu, Fisher, and Vladlen Koltun. 2016. "Multi-Scale Context Aggregation by Dilated Convolutions." *International Conference on Learning Representations*. https://arxiv.org/abs/1511.07122.

Yun, Sangdoo, Dongyoon Han, Seong Joon Oh, Sanghyuk Chun, Junsuk Choe, and

Youngjoon Yoo. 2019. "CutMix: Regularization Strategy to Train Strong Classifiers with Localizable Features." *Proceedings of the IEEE/CVF International Conference on Computer Vision*. https://arxiv.org/abs/1905.04899.

Zhang, Biao, and Rico Sennrich. 2019. "Root Mean Square Layer Normalization." *Advances in Neural Information Processing Systems* 32.

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, Hongyi, Moustapha Cisse, Yann N. Dauphin, and David Lopez-Paz. 2018. "mixup: Beyond Empirical Risk Minimization." *International Conference on Learning Representations*. https://arxiv.org/abs/1710.09412.

Zhang, Hongyi, Yann N. Dauphin, and Tengyu Ma. 2019. "Fixup Initialization: Residual Learning Without Normalization." *International Conference on Learning Representations*. https://arxiv.org/abs/1901.09321.

Zhang, Richard. 2019. "Making Convolutional Networks Shift-Invariant Again." *International Conference on Machine Learning*. https://arxiv.org/abs/1904.11486.

Zhang, Wei, Jun Tanida, Kazuyoshi Itoh, and Yoshiki Ichioka. 1988. "Shift-Invariant Pattern Recognition Neural Network and Its Optical Architecture." *Proceedings of Annual Conference of the Japan Society of Applied Physics*.

Zhang, Xiangyu, Xinyu Zhou, Mengxiao Lin, and Jian Sun. 2018. "ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 6848–56.

Zoph, Barret, and Quoc V Le. 2016. "Neural Architecture Search with Reinforcement Learning." *ArXiv:1611.01578*. https://arxiv.org/abs/1611.01578.