NUMERICAL METHODS FOR MACHINE LEARNING

OPTIMIZATION, STABILITY, AND ALGORITHMS

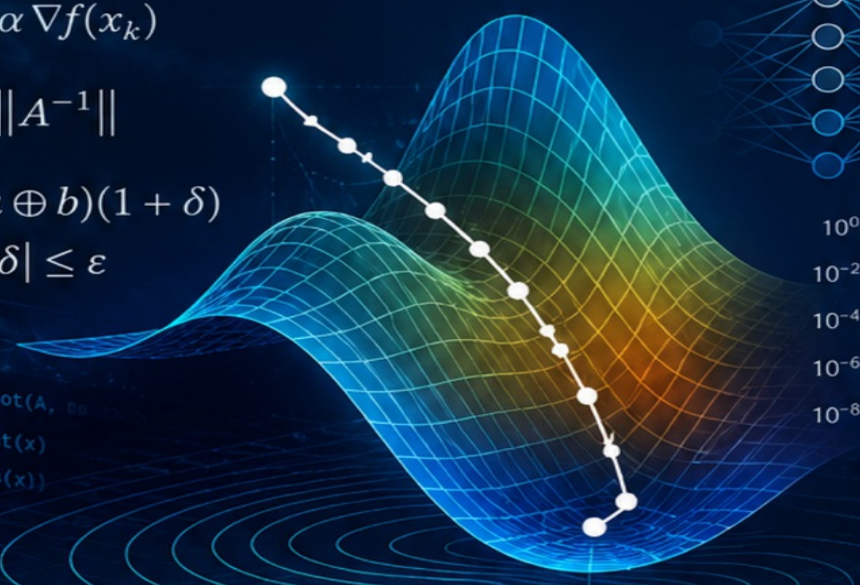
A rigorous and practical guide to the numerical foundations of modern machine learning systems

$$x_{k+1} = x_k - \alpha \nabla f(x_k)$$

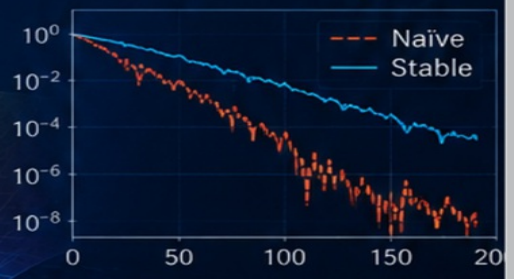
$$\kappa(A) = \|A\|\|A^{-1}\|$$

$$fl(a \oplus b) = (a \oplus b)(1 + \delta)$$
$$|\delta| \leq \varepsilon$$

```
import numpy as np
def gradient(x):
 return np.dot(A.T, np.dot(A,
 x = x - alpha * gradient(x)
 loss.append(compute_loss(x))
```







$$1.18 \times 10^{38}$$
$$\vdots$$
$$3.40 \times 10^{-38}$$

MIR HOSSAIN

FIRST EDITION • 2026

NUMERICAL FOUNDATIONS | OPTIMIZATION METHODS | LINEAR ALGEBRA AND ALGORITHMS | STABILITY AND PRACTICAL SYSTEMS

Numerical Methods for Machine Learning

Optimization, Stability, and Algorithms

Mir Hossain

Numerical Methods for Machine Learning: Optimization, Stability, and Algorithms

Copyright © 2026 Mir Hossain

All rights reserved.

Trademarks: Python, NumPy, PyTorch, TensorFlow, SciPy, NVIDIA, CUDA, and other product names mentioned in this book are trademarks of their respective owners and are used for identification purposes only. This book is not affiliated with or endorsed by any of these companies.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the author, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law.

The information in this book is distributed on an "as is" basis, without warranty. While every precaution has been taken in the preparation of this work, neither the author nor the publisher shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained herein.

Code listings, mathematical formulations, and case studies in this book are for educational purposes. Production deployments should follow appropriate validation, testing, and engineering practices.

First Edition: 2026

Preface

Machine learning is a discipline built on top of numerical computation. Every gradient that descends, every weight that updates, and every probability that gets computed, all happen in finite-precision arithmetic on hardware that does not honor the clean abstractions of pure mathematics. This is the gap this book sets out to bridge.

Most machine learning texts treat numerics as a side concern. They derive elegant update rules, prove convergence under idealized conditions, and leave the reader to discover, on their own, why those equations sometimes produce NaN values, why gradients explode the moment they touch real data, and why the same algorithm that converged in a paper diverges in production. This book is for the reader who has been on the receiving end of those surprises.

The audience I have in mind is the upper-undergraduate or graduate student in machine learning, the working data scientist who wants to understand why their models behave the way they do, and the engineer building systems that have to keep working as scale increases by orders of magnitude. The mathematics here is rigorous where rigor matters, but the goal is always to connect theory to what actually happens when code runs.

The book is organized in seven parts. Part I lays the foundations: floating-point arithmetic, error analysis, and the numerical view of linear algebra. Part II covers optimization in depth, from gradient descent through second-order and constrained methods. Part III is dedicated to numerical linear algebra, the workhorse of every modern ML system. Part IV applies these tools to deep learning specifically. Part V addresses probabilistic and statistical computation. Part VI covers stability, generalization, and the practical engineering of robust pipelines. Part VII brings everything together through extended case studies.

Throughout, I have tried to follow three rules. First, every important algorithm should be accompanied by an honest discussion of when it fails. Second, mathematical claims should be grounded in code that the reader can run. Third, the goal is intuition and competence, not encyclopedic coverage. There are excellent reference works for the latter; this is meant to be the book you read cover to cover.

I owe an intellectual debt to the long tradition of numerical analysis, Wilkinson, Higham, Trefethen, Golub, and many others, and to the more recent generation who have built the foundations of modern machine learning on top of it. My contribution, such as it is, is to draw the connections.

Mir Hossain

2026

How to Use This Book

This book can be read in three modes, depending on your goals and background.

Cover to cover

If you are encountering numerical methods for the first time in the context of machine learning, read the parts in order. Each part builds on the previous one. By the end you will have a complete mental model of how ML systems compute, why they fail, and how to make them robust.

As a reference

If you have a specific problem, exploding gradients, slow convergence, instability in a Bayesian sampler, go directly to the relevant chapter. The index and chapter summaries are designed to support this kind of targeted reading. Cross-references throughout point to the foundational material when needed.

Course companion

The book is designed to support a one-semester graduate course in numerical methods for ML. Parts I and II form the core. Part III is essential for any course covering large-scale methods. Part IV is the natural choice for a deep-learning-focused syllabus. Parts V through VII supply material for advanced topics, projects, and independent reading.

Code conventions

Code examples are written in Python with NumPy and, where relevant, PyTorch. They are designed to be minimal, illustrating the numerical idea rather than serving as production-ready libraries. Each code listing is self-contained and runnable as shown; readers are encouraged to copy these snippets into their own working environment, modify the parameters, and observe how the numerical behavior changes. The code is intentionally framework-agnostic where possible: most examples use only NumPy, and PyTorch is reserved for sections that genuinely require automatic differentiation or GPU acceleration. To get the most out of this book, run the code as you read.

Mathematical conventions

Vectors are lowercase bold or, in formulas, set in math italics. Matrices are uppercase. The norm without subscript denotes the 2-norm. Logarithms are natural unless otherwise noted. The symbol $\approx$ indicates approximate equality, while $\equiv$ indicates definition.

Contents

Preface
How to Use This Book
    Cover to cover
    As a reference
    Course companion
    Code conventions
    Mathematical conventions
Contents
Chapter 1: Why Numerical Methods Matter in Machine Learning
  1.1 Numerical vs Analytical Solutions
    The trade we make
  1.2 Floating-Point Arithmetic and Precision Limits
    The bits in a float32
    Machine epsilon
    Catastrophic cancellation
    Overflow and underflow
  1.3 Conditioning vs Stability
    Conditioning is a property of the problem
    Stability is a property of the algorithm
  1.4 Real ML Failures Caused by Numerical Issues
    Softmax overflow
    Loss of precision in attention
    Variance underflow in batch normalization
    Log of zero in cross-entropy
    Gradient explosion in RNNs
  1.5 Case Study: Exploding Gradients
    The fixes
  1.6 Looking Ahead
    Exercises
Chapter 2: Errors, Stability, and Conditioning
  2.1 Sources of Error: Truncation and Rounding
    Truncation error
    Rounding error
    The two errors interact
  2.2 Forward and Backward Error
    Forward error

Backward error
The relationship: forward = backward × condition
2.3 Condition Number: Intuitive and Formal
Intuitive definition
Condition number for solving linear systems
Condition of common ML operations
2.4 Stability in Iterative Algorithms
Three regimes for iterative algorithms
Convergence rates
2.5 Sensitivity Analysis in ML Models
Input sensitivity
Weight sensitivity
Data sensitivity
Sensitivity and conditioning are the same idea
Exercises
Chapter 3: Linear Algebra Refresher (Numerical Perspective)
3.1 Matrix Norms and Vector Norms
Vector norms
Matrix norms
Why we care
3.2 Orthogonality and Projections
Why orthogonality is numerically valuable
Projections
3.3 Eigenvalues and Eigenvectors
The spectrum
Eigenvalues are not singular values
Eigenvalues in ML
3.4 Positive Definite Matrices
Where they appear
Properties that matter numerically
3.5 Why Numerical Linear Algebra Is Not Textbook Linear Algebra
Inverses are forbidden
Determinants are also forbidden
Equality is conditional
Order matters
The right factorizations
Exercises
Chapter 4: The Optimization Landscape in Machine Learning

- 4.1 Loss Functions and Their Geometry
  - Squared loss
  - Cross-entropy
  - Adversarial and minimax losses
- 4.2 Convex vs Non-Convex Optimization
  - Why convexity is a gift
  - Why machine learning is not convex
- 4.3 Saddle Points vs Minima
  - Why saddle points dominate in high dimensions
- 4.4 Ill-Conditioning in Deep Learning
  - Why deep networks are ill-conditioned
  - Exercises

Chapter 5: Gradient-Based Methods

- 5.1 The Update Rule
  - What gradient descent assumes
- 5.2 Batch, Stochastic, and Mini-Batch
  - Batch gradient descent
  - Stochastic gradient descent
  - Mini-batch gradient descent
- 5.3 Choosing the Learning Rate
  - The mathematical bound
  - Practical recipes
- 5.4 Convergence Intuition
- 5.5 Vanishing and Exploding Gradients
  - Vanishing gradients
  - Exploding gradients
  - The structural fix: residual connections
  - Exercises

Chapter 6: Advanced Gradient Methods

- 6.1 Momentum
  - Why it works
- 6.2 Nesterov Acceleration
- 6.3 Adaptive Methods: AdaGrad, RMSProp, and Adam
  - AdaGrad
  - RMSProp
  - Adam
  - AdamW
- 6.4 When Adaptive Methods Fail

Generalization gap
Pathological learning rate behavior
The practitioner's guidance
Exercises
Chapter 7: Second-Order Methods
7.1 Newton's Method
Why we cannot simply use Newton's method
Modifications that work
7.2 Quasi-Newton Methods: BFGS and L-BFGS
BFGS
L-BFGS
Why L-BFGS is rarely used in deep learning
7.3 Hessian Approximation Techniques
Diagonal approximations
Hessian-vector products
Gauss-Newton and natural gradient
7.4 Trade-offs: Cost vs Convergence Speed
The deep learning calculus
Exercises
Chapter 8: Constrained Optimization
8.1 The Setting
Examples in machine learning
8.2 Lagrange Multipliers
Geometric intuition
8.3 KKT Conditions
The conditions
KKT in support vector machines
8.4 Projected Gradient Descent
Why it works
When the projection is hard
8.5 Applications in Regularization
Constraint vs penalty
L1 regularization and sparsity
Constrained optimization in modern ML
Exercises
Chapter 9: Solving Linear Systems
9.1 Direct Methods: Gaussian Elimination and LU
LU decomposition

- Pivoting
- Cost and limits
- 9.2 Iterative Methods: Jacobi and Gauss-Seidel
  - Jacobi iteration
  - Gauss-Seidel
- 9.3 Conjugate Gradient
  - How it works
  - Why CG is essential for ML
  - Preconditioning
- 9.4 Choosing the Right Solver
  - Exercises

Chapter 10: Matrix Factorizations

- 10.1 LU Decomposition
- 10.2 QR Decomposition
  - Three ways to compute QR
  - QR for least squares
- 10.3 Cholesky Decomposition
  - Where it appears in ML
  - Cholesky as a positive-definiteness test
- 10.4 SVD: The Most Powerful Factorization
  - What SVD tells you
  - Cost
- 10.5 Stability Considerations
  - LU stability
  - QR stability
  - Cholesky stability
  - SVD stability
- 10.6 When to Use Which Method
  - Exercises

Chapter 11: Eigenvalue Problems and PCA

- 11.1 Power Iteration
  - Convergence rate
  - Why power iteration matters
- 11.2 The QR Algorithm
- 11.3 Singular Value Decomposition
  - Connection to eigenvalues
- 11.4 PCA from a Numerical Perspective
  - The recipe

Numerical pitfalls
11.5 Dimensionality Reduction at Scale
Truncated SVD
Randomized SVD
Incremental PCA
PCA via Hebbian learning
Exercises
Chapter 12: Large-Scale Linear Algebra
12.1 Sparse Matrices
Sparse storage formats
Sparse matrix-vector multiplication
12.2 Memory-Efficient Computation
Blocked algorithms
In-place computation
Streaming computation
12.3 Iterative Solvers for Big Data
Krylov subspace methods
Preconditioning at scale
12.4 GPU Considerations
What GPUs love
What GPUs hate
Mixed-precision training
Distributed linear algebra
Exercises
Chapter 13: Backpropagation as a Numerical Algorithm
13.1 The Chain Rule as a Computation Graph
Forward and backward
Static vs dynamic graphs
13.2 Reverse-Mode Differentiation
Why reverse mode wins for ML
Memory: the price of reverse mode
13.3 Numerical Instability in Backpropagation
The matrix-product picture
Gradient quantization
Numerical gradient checking
Exercises
Chapter 14: Initialization and Conditioning
14.1 Why Initialization Matters

- 14.2 Weight Initialization Strategies
  - Xavier (Glorot) initialization
  - He initialization
  - Orthogonal initialization
- 14.3 Batch Normalization
  - What batch norm fixes
  - Inference complications
- 14.4 Layer Normalization
  - Where each is used
  - Other normalizations
- 14.5 Conditioning of Deep Networks
  - The conditioning of a deep network
  - Residual connections as conditioning
  - Exercises

Chapter 15: Regularization from a Numerical View

- 15.1 L2 Regularization
  - Numerical effect: conditioning
  - Statistical effect: bias-variance tradeoff
- 15.2 L1 Regularization
  - Why L1 produces sparsity
  - Numerical aspects
- 15.3 Dropout as Stochastic Approximation
  - What dropout does numerically
  - Inference and dropout
- 15.4 Early Stopping as Numerical Control
  - The connection to L2
  - Beyond early stopping: dynamic regularization
  - Exercises

Chapter 16: Optimization Pathologies

- 16.1 Vanishing Gradients
  - Diagnosing it
  - Causes
  - Fixes
- 16.2 Exploding Gradients
  - Gradient clipping
  - Architectural fixes
- 16.3 Saddle Point Plateaus
  - Diagnosing a plateau

Escaping
16.4 Flat vs Sharp Minima
Why optimizers find flatter minima
Sharpness-aware minimization
Exercises
Part V
Probabilistic & Statistical Numerical Methods
Chapter 17: Numerical Methods in Probability
17.1 Sampling Methods
Inverse transform sampling
Box-Muller and the Gaussian
Rejection sampling
Importance sampling
17.2 Monte Carlo Integration
Variance reduction
17.3 Law of Large Numbers in Practice
Code example: Monte Carlo estimate of π
Exercises
Chapter 18: Optimization in Probabilistic Models
18.1 Maximum Likelihood Estimation
Numerical issues in MLE
18.2 Gradient-Based Inference
Reparameterization trick
18.3 Variational Inference Basics
Mean-field VI
Stochastic VI
Numerical pitfalls in VI
Code example: VI for Bayesian regression
Exercises
Chapter 19: Bayesian Computation
19.1 Markov Chain Monte Carlo
Metropolis-Hastings
Hamiltonian Monte Carlo
19.2 Gibbs Sampling
19.3 Numerical Challenges in Bayesian ML
Log-space arithmetic
Mixing and convergence diagnostics
Multimodal posteriors

Scaling to big data
Code example: Random-walk Metropolis
Exercises
Part VI
Stability, Generalization & Practical Systems
Chapter 20: Numerical Stability in ML Pipelines
20.1 Stable Softmax Computation
Stable softmax: implementation
20.2 The Log-Sum-Exp Trick
Stable log-sum-exp
20.3 Avoiding Overflow and Underflow
Stable sigmoid
Stable log(1 + exp(x))
Stable sigmoid cross-entropy
Catastrophic cancellation
Mixed precision
Exercises
Chapter 21: Generalization vs Numerical Behavior
21.1 Overfitting and Conditioning
21.2 Noise as Regularization
Dropout as noise injection
Data augmentation
21.3 The Double Descent Phenomenon
The interpolation threshold
Beyond interpolation
Practical implications
Exercises
Chapter 22: Debugging Numerical Issues
22.1 Diagnosing NaNs and Infinities
Locating the source
Detecting when, not just where
22.2 Gradient Checking
Implementation
22.3 Unit Tests for Numerical Code
Property-based testing
Numerical tolerances
Edge cases to test
Regression tests

Exercises
Chapter 23: Performance & Scaling
23.1 Vectorization Strategies
Replace loops with array operations
Use BLAS-backed operations
Broadcasting
23.2 Parallel Computation
Multi-threading
Data parallelism
Model parallelism
Mixed strategies
23.3 GPU and TPU Numerical Considerations
Throughput vs. latency
Reduced-precision arithmetic
Determinism and reproducibility
Tensor cores and structured operations
Exercises
Part VII
Case Studies & Applications
Chapter 24: Training a Neural Network from Scratch
24.1 Step-by-Step Implementation
Data and shapes
Initialization
Forward pass
Backward pass
Training loop
24.2 Stability Checks
Initial loss
Gradient check on a small instance
Train on a tiny dataset
Watch the activation statistics
24.3 Performance Tuning
Learning rate
Momentum
Mini-batch size
Weight decay
Expected results
Exercises

Chapter 25: Logistic Regression at Scale
25.1 The Optimization Choice
Newton's method
L-BFGS
SGD and stochastic methods
Parallel and distributed
25.2 Numerical Stability Tricks
Stable log(1 + exp(x))
Sparse data structures
Feature scaling
Regularization path
Class imbalance
25.3 A Worked Comparison
Exercises
Chapter 26: PCA on Large Dataset
26.1 Efficient SVD
ARPACK and Lanczos iteration
Randomized SVD
Power iteration corrections
26.2 Approximation Methods
Online and incremental PCA
Stochastic methods for PCA
Sketching
26.3 Worked Example: Image Compression via PCA
Centering and scaling
Choosing k
Reconstruction quality
Performance
Exercises
Chapter 27: Real-World ML Failures
27.1 The Silent Loss-Function Bug
27.2 The Mixed-Precision Pitfall
27.3 The Distance-Metric Catastrophe
27.4 The Subtle Data-Skew Failure
27.5 The Quantization Surprise
27.6 What These Failures Have in Common
Exercises
Appendix A: Numerical Recipes for ML Engineers

- A.1 Stable Softmax
- A.2 Stable Log-Sum-Exp
- A.3 Stable Sigmoid
- A.4 Stable Softplus
- A.5 Stable Binary Cross-Entropy with Logits
- A.6 Stable Cross-Entropy with Logits
- A.7 Two-Pass Variance
- A.8 Welford's Online Variance
- A.9 Safe Cosine Similarity
- A.10 Stable LayerNorm
- A.11 Gradient Clipping
- A.12 Numerically Stable Distance Squared

Appendix B: Cheat Sheet of Algorithms

- B.1 Optimization Methods
  - Gradient Descent
  - SGD
  - Momentum
  - Adam
  - Newton's Method
  - L-BFGS
- B.2 Linear System Solvers
  - Cholesky factorization
  - LU factorization with partial pivoting
  - Conjugate Gradient
  - GMRES
- B.3 Matrix Factorizations
  - QR factorization
  - SVD
  - Eigendecomposition
- B.4 Sampling Methods
  - Inverse transform
  - Box-Muller
  - Rejection sampling
  - Importance sampling
  - Metropolis-Hastings
  - HMC / NUTS
- B.5 Stability Tricks

Appendix C: Python Implementations

C.1 Conjugate Gradient
C.2 Power Iteration for Top Eigenvector
C.3 Randomized SVD
C.4 Mini-Batch SGD with Momentum
C.5 Adam Optimizer
C.6 Backpropagation for a Two-Layer MLP
C.7 K-Means Clustering
C.8 Random-Walk Metropolis
Appendix D: Mathematical Background
D.1 Vectors and Norms
D.2 Inner Products and Orthogonality
D.3 Matrices: Rank, Trace, Determinant
D.4 Eigenvalues and Eigenvectors
D.5 Positive Definite Matrices
D.6 Gradients and Hessians
D.7 Probability Basics
D.8 Common Distributions
D.9 Convexity
D.10 Floating-Point Numbers
Appendix E: Solutions to Exercises
Chapter 1: Why Numerical Methods Matter in Machine Learning
Chapter 2: Errors, Stability, and Conditioning
Chapter 3: Linear Algebra Refresher
Chapter 4: The Optimization Landscape
Chapter 5: Gradient-Based Methods
Chapter 6: Advanced Gradient Methods
Chapter 7: Second-Order Methods
Chapter 8: Constrained Optimization
Chapter 9: Solving Linear Systems
Chapter 10: Matrix Factorizations
Chapter 11: Eigenvalue Problems and PCA
Chapter 12: Large-Scale Linear Algebra
Chapter 13: Backpropagation as a Numerical Algorithm
Chapter 14: Initialization and Conditioning
Chapter 15: Regularization from a Numerical View
Chapter 16: Optimization Pathologies
Chapter 17: Numerical Methods in Probability
Chapter 18: Optimization in Probabilistic Models

- Chapter 19: Bayesian Computation
- Chapter 20: Numerical Stability in ML Pipelines
- Chapter 21: Generalization vs Numerical Behavior
- Chapter 22: Debugging Numerical Issues
- Chapter 23: Performance & Scaling
- Chapter 24: Training a Neural Network from Scratch
- Chapter 25: Logistic Regression at Scale
- Chapter 26: PCA on Large Dataset
- Chapter 27: Real-World ML Failures

Appendix F: Glossary
- Numerical Foundations
- Linear Algebra
- Optimization
- Deep Learning
- Probability and Sampling
- Stability and Engineering
- Generalization

Appendix G: Chapter Summary Sheets
- Chapter 1: Why Numerical Methods Matter
- Chapter 2: Errors, Stability, and Conditioning
- Chapter 3: Linear Algebra Refresher
- Chapter 4: Optimization Landscape
- Chapter 5: Gradient-Based Methods
- Chapter 6: Advanced Gradient Methods
- Chapter 7: Second-Order Methods
- Chapter 8: Constrained Optimization
- Chapter 9: Solving Linear Systems
- Chapter 10: Matrix Factorizations
- Chapter 11: Eigenvalue Problems and PCA
- Chapter 12: Large-Scale Linear Algebra
- Chapter 13: Backpropagation as a Numerical Algorithm
- Chapter 14: Initialization and Conditioning
- Chapter 15: Regularization from a Numerical View
- Chapter 16: Optimization Pathologies
- Chapter 17: Numerical Methods in Probability
- Chapter 18: Optimization in Probabilistic Models
- Chapter 19: Bayesian Computation
- Chapter 20: Numerical Stability in ML Pipelines

Chapter 21: Generalization vs Numerical Behavior
Chapter 22: Debugging Numerical Issues
Chapter 23: Performance and Scaling
Chapter 24: Training a Neural Network from Scratch
Chapter 25: Logistic Regression at Scale
Chapter 26: PCA on Large Dataset
Chapter 27: Real-World ML Failures

Part I

Foundations of Numerical Computation for Machine Learning

Before we can talk about gradient descent, backpropagation, or any of the algorithms that drive modern machine learning, we have to understand the substrate they run on. Computers do not do mathematics; they do approximate arithmetic on bit patterns. The first three chapters establish what that means in practice, how floating-point numbers behave, how errors propagate, and how the linear algebra of textbooks differs from the linear algebra of working systems.

Chapter 1: Why Numerical Methods Matter in Machine Learning

There is a story that recurs in nearly every machine learning practitioner's career. A model is trained, the loss is going down, validation looks healthy, and then, at epoch 47, or batch 12,000, or after a seemingly innocuous change to the input pipeline, the loss becomes NaN. Weights become infinite. The model produces nothing but zeros. The training run is dead, and the cause is not a bug in the algorithm. It is a bug in the arithmetic.

This kind of failure is the everyday reality of numerical methods in machine learning. Algorithms that are mathematically correct on paper can produce wildly wrong answers on a computer, and the difference comes down to the fact that real machines work in finite precision. To build models that train reliably, scale gracefully, and produce trustworthy predictions, we need to understand not just the algorithms but the arithmetic they run on.

1.1 Numerical vs Analytical Solutions

Mathematical analysis gives us closed-form expressions: the derivative of x squared is 2x, the integral of e to the x is itself, the solution to a quadratic is given by the quadratic formula. These are analytical solutions, exact, symbolic, and infinite in precision.

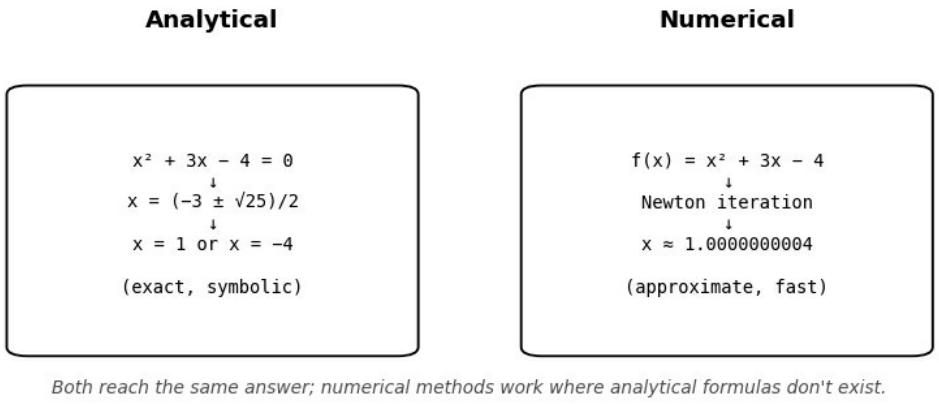


Figure 1.2: Analytical vs numerical solutions. Both reach the same answer; numerical methods work where analytical formulas don't exist.

Numerical methods, by contrast, produce approximate answers using a finite number of arithmetic operations. They estimate, iterate, and refine. The reason we use them is straightforward: the vast majority of problems we care about have no closed-form solution. There is no formula for the optimal weights of a neural network. There is no analytical expression for the eigenvectors of an arbitrary million-by-million matrix. There is no symbolic answer for the integral that defines the marginal likelihood of a Bayesian model.

Even when an analytical solution exists, the numerical answer is often more useful. Consider the linear system Ax = b. Cramer's rule gives a closed-form solution involving determinants, but computing it requires factorial-time arithmetic and produces catastrophic floating-point errors for any matrix larger than about ten by ten. Gaussian elimination, a numerical method, gives the same answer in cubic time with predictable precision.

The trade we make

Numerical methods buy us tractability and scale, but they cost us exactness. Every numerical answer carries some error. The discipline of numerical analysis is the study of how to keep that error small, predictable, and well-understood. In machine learning, where we routinely train models with billions of parameters across days of compute, this discipline is not optional.

1.2 Floating-Point Arithmetic and Precision Limits

Computers represent real numbers using floating-point arithmetic, defined by the IEEE 754 standard. The two formats that dominate machine learning are float32 (single precision) and float64 (double precision), with float16 and bfloat16 increasingly common in deep learning.

A floating-point number has three components: a sign bit, an exponent, and a mantissa (also called the significand). The value represented is, roughly, the mantissa multiplied by two raised to the exponent. The mantissa controls precision; the exponent controls range.

The bits in a float32

A float32 has one sign bit, eight exponent bits, and twenty-three mantissa bits. This gives it about seven decimal digits of precision and a range from roughly 1.18×10^{-38} to 3.40×10^{38}. A float64 has fifty-two mantissa bits, giving about sixteen decimal digits of precision and an enormous range.

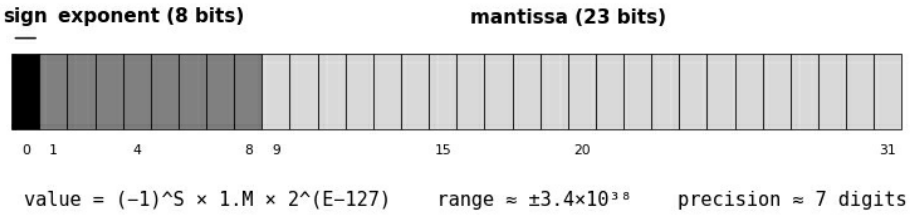


Figure 1.3: IEEE 754 float32 bit layout: 1 sign, 8 exponent, 23 mantissa bits.

Machine epsilon

The single most important number in floating-point arithmetic is machine epsilon, denoted ε (epsilon). It is the smallest number such that $1 + \varepsilon$ is distinguishable from 1 in the given precision. For float32, ε is about 1.19×10^{-7}. For float64, it is about 2.22×10^{-16}.

Machine epsilon governs every rounding error in your computations. When you add two numbers of similar magnitude, the result may differ from the true mathematical sum by up to ε times the magnitude. This sounds tiny, but errors compound. A computation involving a million operations can accumulate error a million times machine epsilon, which is no longer negligible.

```
import numpy as np

# Machine epsilon for different precisions
print(np.finfo(np.float32).eps)  # ~1.19e-07
print(np.finfo(np.float64).eps)  # ~2.22e-16

# The classic demonstration
print(1.0 + 1e-7 == 1.0)        # True in float32
print(1.0 + 1e-16 == 1.0)       # True in float64
```

Catastrophic cancellation

The most insidious source of floating-point error is catastrophic cancellation: the loss of significant digits when subtracting two nearly equal numbers. If both numbers carry seven digits of precision and they agree to six digits, their difference has only one significant digit left. The remaining six digits are noise.

This is why the naive variance formula, the mean of the squares minus the square of the mean, is numerically dangerous. Both terms can be large; their difference can be small; the small difference is what we want, but it is exactly the quantity most contaminated by cancellation. Welford's online algorithm, which we will see in chapter 20, avoids the cancellation entirely by computing the variance incrementally.

Overflow and underflow

Beyond precision, floating-point numbers have finite range. Multiply enough small numbers together and the result rounds to zero, underflow. Multiply enough large numbers and the result becomes infinity, overflow. Both are common in probabilistic models, where products of probabilities can easily underflow, and in deep networks, where products of large activations can overflow.

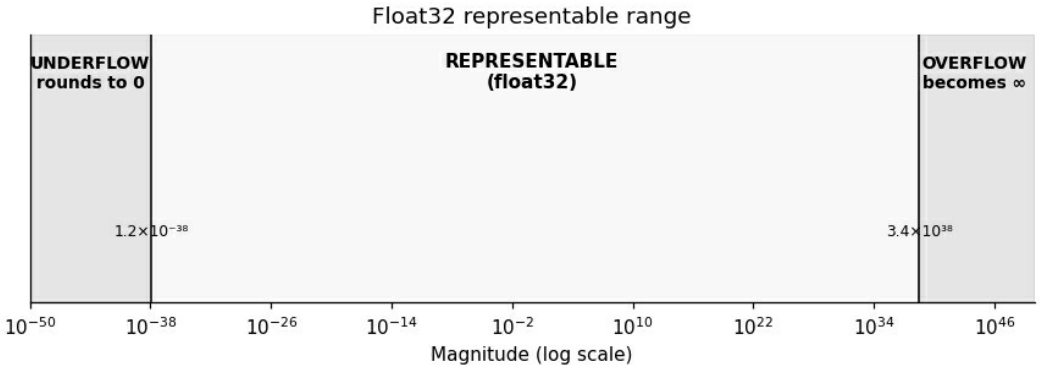


Figure 1.4: Float32 representable range. Values below ~1.2×10⁻³⁸ underflow to zero; values above 3.4×10³⁸ overflow to infinity.

The standard defense is to work in log space. The logarithm turns products into sums, which are immune to both overflow and underflow as long as no individual term is itself extreme. This is why log-likelihoods, log-probabilities, and the log-sum-exp trick (chapter 20) are everywhere in machine learning.

1.3 Conditioning vs Stability

Two concepts that are often confused but must be kept separate are conditioning and stability.

Conditioning is a property of the problem

A problem is well-conditioned if small changes in the input produce small changes in the output, and ill-conditioned if small changes in the input produce large changes in the output. Conditioning has nothing to do with the algorithm used to solve the problem; it is intrinsic to the problem itself.

The classic example is solving a linear system Ax = b where the matrix A is nearly singular. A small perturbation of b can cause an enormous change in x. No algorithm can save you from this; the problem itself amplifies error, and the best any algorithm can do is not amplify it further.

Stability is a property of the algorithm

An algorithm is numerically stable if it produces an answer close to the exact answer of a slightly perturbed problem. In other words, a stable algorithm doesn't amplify error beyond what the problem itself requires. An unstable algorithm introduces error of its own, on top of the unavoidable conditioning of the problem.

The right mental model is this: conditioning sets a floor on how accurate your answer can be; stability determines whether you reach that floor or fall short of it. Choosing a stable algorithm for a well-conditioned problem is the recipe for reliable computation. Using an unstable algorithm on an ill-conditioned problem is a recipe for nonsense.

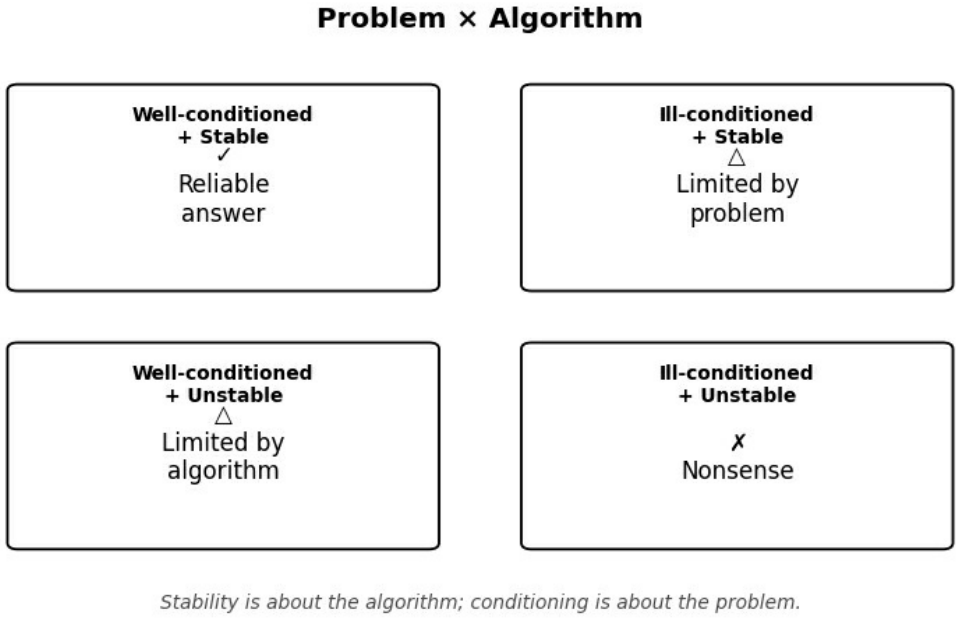


Figure 1.5: Conditioning is a property of the problem; stability is a property of the algorithm.

1.4 Real ML Failures Caused by Numerical Issues

To make this concrete, here are failure modes that appear in actual machine learning systems.

Softmax overflow

The softmax function computes exp(x_i) divided by the sum of exp(x_j). For inputs of moderate size, say, logits of 1000, which can occur in unscaled networks, exp(1000) is wildly larger than the maximum representable float and produces infinity. The naive softmax then divides infinity by infinity and produces NaN. The fix, the log-sum-exp trick, is to subtract the maximum logit before exponentiating.

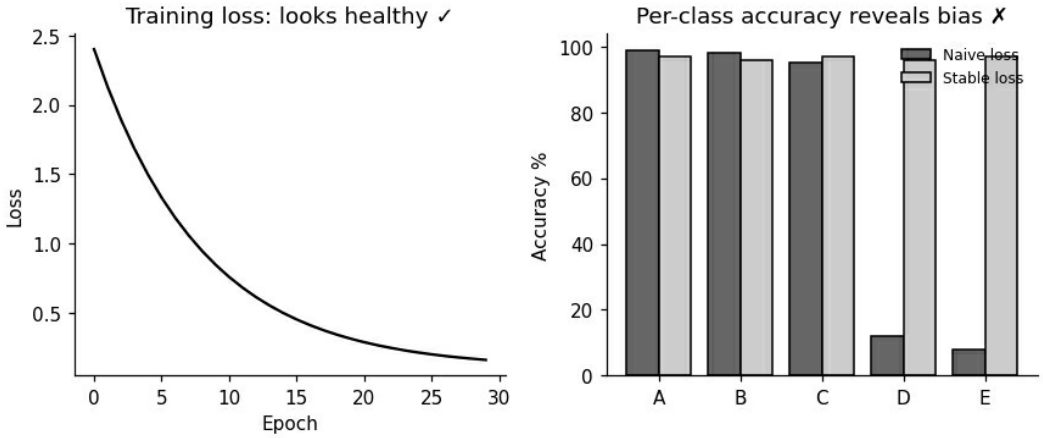


Figure 1.6: Naive softmax overflows for large logits; subtracting the max gives the same result, stably.

Loss of precision in attention

In transformer attention, the dot product of two large vectors can grow with their dimension. Without the famous division by the square root of d_k, attention scores grow large, the softmax saturates, and gradients vanish. This scaling factor is not aesthetic, it is a numerical necessity.

Variance underflow in batch normalization

Batch normalization divides by the standard deviation of activations. If the variance is very small, early in training, or for a saturated unit, the division underflows or produces a huge gradient. Every implementation includes a small epsilon added to the variance for exactly this reason.

Log of zero in cross-entropy

The cross-entropy loss involves log(p) where p is a predicted probability. If the network predicts p = 0 for the correct class, log(0) is negative infinity, and the gradient is undefined. Implementations clip probabilities away from zero or, better, compute the loss directly from logits using the log-sum-exp formulation.

Gradient explosion in RNNs

Recurrent networks multiply the same weight matrix at every time step. If the largest singular value of that matrix is greater than one, gradients grow exponentially with sequence length. By step 100, the gradient may be 10^30 times its initial value, well past the float32 overflow limit. The result is NaN gradients and a dead model.

1.5 Case Study: Exploding Gradients

Let us walk through the exploding-gradient phenomenon in detail, because it ties together every theme of this chapter: conditioning, stability, precision, and the gap between the math we write and the arithmetic that runs.

Consider a simple recurrent unit that, at each step, multiplies its hidden state by a matrix W and adds a small input. Mathematically, after T steps, the hidden state is the input convolved with powers of W. Differentiating with respect to the original input gives derivatives that involve W to the power T.

$$\partial h_T / \partial h_0 = W^T$$

Now suppose the largest eigenvalue of W is 1.1. After 100 steps, the gradient is scaled by 1.1 to the 100th power, which is roughly 13,800. After 200 steps, it is 190 million. After 300 steps, it is two and a half trillion. Float32, our standard precision, overflows at around 3.4 × 10^38, so by around step 931, the gradient is literally infinity.

This is exploding gradients. The problem is well-defined mathematically, the gradient has a precise value at every step, but the numerical representation cannot hold it. The conditioning of the problem (how sensitive the loss is to early hidden states) is intrinsically bad for deep recurrence. The stability of the algorithm (we are computing the gradient correctly) does not save us, because no algorithm can compute a number that is too big to represent.

The fixes

- Gradient clipping: detect when the gradient norm exceeds a threshold and rescale it. This sacrifices the magnitude of the gradient direction in exchange for finite arithmetic. It is the most common intervention in practice.
- Architectural choices: LSTMs and GRUs include gating mechanisms that bound the effective recurrence and prevent eigenvalues from drifting far from one. Transformers replace recurrence with attention, sidestepping the problem entirely.
- Better initialization: orthogonal initialization sets W to a matrix with all singular values equal to one, postponing the explosion until training has had a chance to stabilize.
- Higher precision: training in float64 buys a few orders of magnitude more headroom, but at twice the memory and compute cost. It is rarely the right answer in deep learning.

None of these fixes change the underlying math. What they change is how the math interacts with finite-precision arithmetic. This is the central insight: machine learning systems live in the gap between mathematical algorithms and numerical computation, and bridging that gap is what numerical methods are for.

1.6 Looking Ahead

The rest of this book is organized around the consequences of working in finite precision. Chapter 2 develops the language of errors, conditioning, and stability with more rigor. Chapter 3 revisits linear algebra from the numerical perspective, what we already know about matrices, but with the corrections that finite arithmetic forces on us. From there we proceed through optimization, deep learning, probabilistic computation, and finally the engineering of robust ML pipelines.

By the end, the reader should have not just a toolbox of techniques but a way of thinking about numerical computation in machine learning. When a model misbehaves, the goal is to be able to diagnose whether the cause is in the algorithm, in the data, or in the arithmetic, and to know what to do about it.

Key Takeaways

- Machine learning runs on finite-precision arithmetic; the gap between continuous math and floating-point hardware is where most production failures live.
- Float32 has about 7 digits of precision and a range from roughly 1.18×10^{-38} to 3.40×10^{38}; outside that range you get overflow or underflow.
- Conditioning is a property of the problem; stability is a property of the algorithm. Both must be

good for reliable results.

- Always prefer stable formulations: softmax with max-subtraction, log-sum-exp, Welford's variance, BCE-with-logits.
- Understanding floating-point precision, conditioning, and stability is the foundation for everything that follows.

Exercises

- [Basic] Compute machine epsilon for float16, float32, and float64. Verify that $1 + \varepsilon$ is distinguishable from 1 but $1 + \varepsilon/2$ is not.
- [Intermediate] Implement the naive variance formula and Welford's online algorithm. Compare their accuracy on a sequence of values around 10^9 with small variation.
- [Intermediate] Construct a 2×2 matrix with condition number approximately 10^7. Solve a linear system with this matrix and observe how a small perturbation of the right-hand side affects the solution.
- [Intermediate] Implement softmax both naively and with the log-sum-exp trick. Find an input where the naive version returns NaN and the stable version does not.
- [Intermediate] Simulate exploding gradients by repeatedly multiplying a vector by a matrix with spectral radius 1.1. At what step does the result overflow in float32?

Chapter 2: Errors, Stability, and Conditioning

Every numerical computation produces an answer that differs, however slightly, from the mathematically exact answer. The art of numerical analysis is to understand where that error comes from, how it propagates through a chain of operations, and how to keep it bounded. In this chapter we build the formal vocabulary for talking about error, vocabulary that will reappear in every subsequent chapter.

2.1 Sources of Error: Truncation and Rounding

There are two fundamental sources of numerical error, and they have very different characters.

Truncation error

Truncation error arises when we replace an exact mathematical operation with an approximation. The Taylor series for sine has infinitely many terms; we truncate it after a few. A derivative is a limit; we approximate it with a finite difference. An iterative algorithm runs forever; we stop after some number of iterations. Each of these substitutions introduces truncation error.

Truncation error is, in principle, controllable. If we use more terms of the Taylor series, take a smaller step size, or run more iterations, the error shrinks. The relationship between effort and error is what we call the order of accuracy of a method. A first-order method has error proportional to the step size h; a second-order method has error proportional to h squared.

Rounding error

Rounding error arises from finite precision. Even if we use the exact mathematical operation, the answer cannot be stored exactly in a floating-point number; it must be rounded to the nearest representable value. This introduces an error of at most machine epsilon, relative to the magnitude of the result.

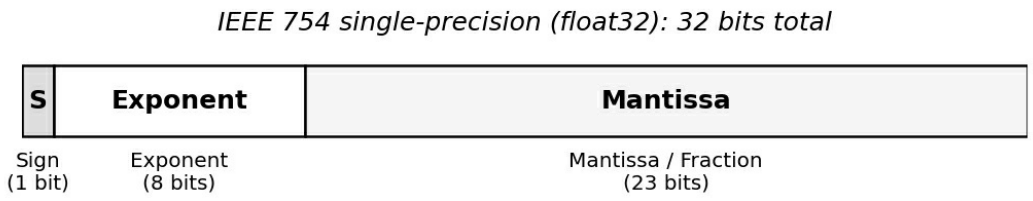


Figure 1.1: Spacing of floating-point numbers on the real line. Density is highest near zero and decreases as magnitude grows.

Rounding error is, in principle, uncontrollable. We cannot make machine epsilon smaller without using more bits, and even then there is a limit. What we can do is arrange our computations so that rounding errors do not compound, by choosing stable algorithms and avoiding catastrophic cancellation.

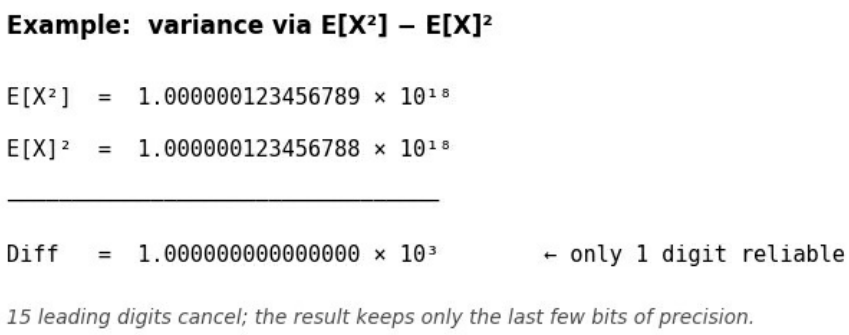


Figure 1.7: Catastrophic cancellation. Subtracting nearly equal floating-point numbers leaves only the rightmost few bits of precision.

The two errors interact

Often, reducing one error increases the other. Take numerical differentiation: the central difference approximation to f'(x) uses a step size h. Smaller h reduces truncation error but amplifies rounding error, because we are subtracting two nearly equal values. The optimal step size balances the two, and it is typically around the square root of machine epsilon. This is the canonical example of an error trade-off.

2.2 Forward and Backward Error

When a numerical algorithm produces an answer ŷ for a problem with exact answer y, we can ask two distinct questions about the quality of the answer.

Forward error

Forward error is the difference between the computed answer and the true answer: ŷ − y, or its norm. This is what most users care about: how wrong is my number?

Forward error is what we usually want to bound, but it can be very hard to compute. To know the forward error, we would have to know the true answer, which is the thing we are trying to compute.

Backward error

Backward error takes a different perspective. Instead of asking how far ŷ is from the true answer y, we ask: for which slightly modified problem is ŷ the exact answer? If ŷ is the exact solution to a problem only slightly different from the original, the algorithm is doing its job well. Even if the forward error is large, that error must then be the fault of the problem, not the algorithm.

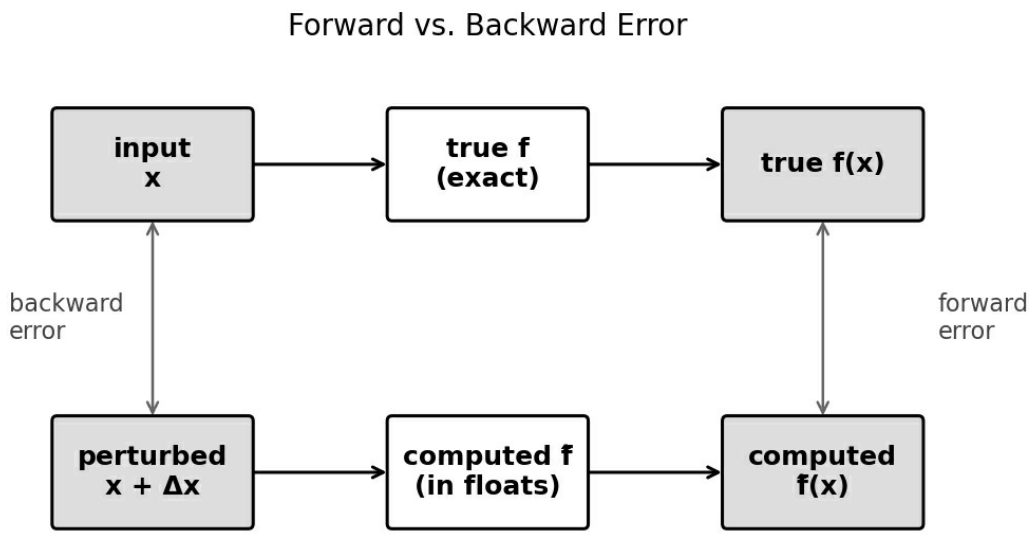


Figure 2.1: Forward error measures how wrong the answer is; backward error measures how perturbed the problem would need to be to make the computed answer exact.

Backward error is the central concept of modern numerical analysis. James Wilkinson popularized it in the 1960s and used it to give clean error analyses of algorithms like Gaussian elimination and QR decomposition that had previously seemed mysterious. Most analyses you will encounter in numerical linear algebra are in terms of backward error.

The relationship: forward = backward × condition

There is a beautiful equation that connects forward error, backward error, and conditioning.

$$\textit{forward error} \lesssim \textit{condition number} \times \textit{backward error}$$

This says: the forward error of a computation is bounded by the condition number of the problem times the backward error of the algorithm. If we want small forward error, we need either a well-conditioned problem (small condition number) or a stable algorithm (small backward error), and ideally both.

2.3 Condition Number: Intuitive and Formal

The condition number quantifies how sensitive a problem is to perturbations in its input. We have used the term informally; it is time to define it.

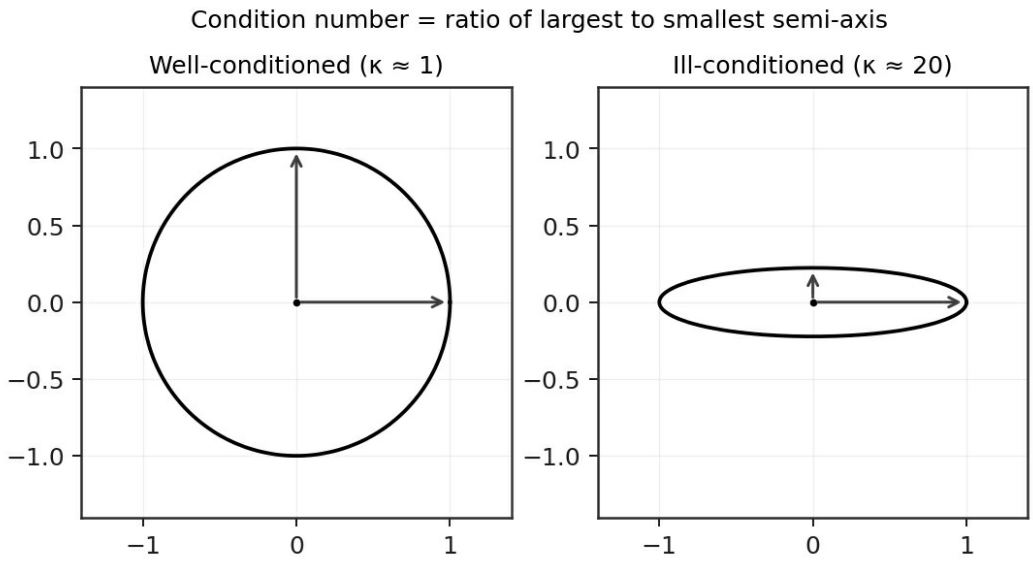


Figure 2.2: A well-conditioned problem (left) maps small input changes to small output changes; an ill-conditioned problem (right) amplifies them.

Intuitive definition

If the input to a problem changes by a small amount δx, the output changes by some amount δy. The condition number is, roughly, the ratio of the relative change in the output to the relative change in the input.

$$\kappa \approx (\|\delta y\|/\|y\|) / (\|\delta x\|/\|x\|)$$

A condition number of 1 means perturbations are not amplified at all. A condition number of 100 means a 1% input error becomes a 100% output error. A condition number of 10^16, in float64, means the algorithm can produce zero correct digits.

Condition number for solving linear systems

For the linear system Ax = b, the condition number of the matrix A (in the 2-norm) is the ratio of its largest to its smallest singular value.

$$\kappa(A) = \sigma_{max}(A) / \sigma_{min}(A)$$

A matrix with very small smallest singular value is nearly singular, and κ(A) is huge. Solving a linear system with such a matrix is fundamentally difficult, regardless of algorithm. This is one reason why regularization, adding a small multiple of the identity to the matrix, is everywhere in machine learning. It bounds the condition number from below.

Condition of common ML operations

- Matrix-vector multiplication y = Ax: condition number depends on A in a complicated way, but is bounded by κ(A) when y is the relevant output.
- Solving Ax = b: condition number is κ(A).
- Inverting a matrix: same as solving, κ(A).
- Eigenvalue computation: well-conditioned for symmetric matrices, can be ill-conditioned for nonsymmetric ones.
- Computing the loss of a neural network: depends on the network architecture and

depth; deep networks are typically very ill-conditioned with respect to early-layer weights.

2.4 Stability in Iterative Algorithms

So far we have discussed conditioning of static problems. But many algorithms in machine learning are iterative, they produce a sequence x_0, x_1, x_2, ... that ideally converges to the right answer. For such algorithms, stability has a slightly different meaning: small perturbations should not cause the sequence to diverge or to converge to a different limit.

Three regimes for iterative algorithms

- Stable convergent: small perturbations do not affect the limit. Most well-designed algorithms fall here.
- Stable but slow: the algorithm converges but at a rate that may be impractical. Conjugate gradient on an ill-conditioned matrix can be in this regime.
- Unstable: small perturbations cause divergence or convergence to a wrong limit. Power iteration on a matrix with two equal-magnitude largest eigenvalues can fail this way.

Convergence rates

Even when an iterative algorithm is stable, how fast it converges matters. We classify convergence rates as:

- Linear: error at step k+1 is a constant fraction of the error at step k. Each iteration reduces error by a fixed multiplicative factor. Gradient descent on a strongly convex function exhibits linear convergence.
- Sublinear: the constant fraction approaches one as k grows; convergence slows down. Stochastic gradient descent on a non-strongly-convex objective is typically sublinear.
- Superlinear: the convergence factor goes to zero. The error shrinks faster and faster. Quasi-Newton methods often achieve superlinear convergence.
- Quadratic: the error squares at each step. Once you are close, you double your number of correct digits with every iteration. Newton's method achieves quadratic convergence near a root.

Quadratic convergence is incredibly fast, once it starts. The catch is the phrase "once you are close." Newton's method may not converge at all from a bad starting point. The art of using these methods is matching the algorithm to the problem and ensuring you get into the regime where the fast convergence applies.

2.5 Sensitivity Analysis in ML Models

Sensitivity analysis is the practical application of these ideas to machine learning models. The core question is: how does the output of a trained model change when its inputs, weights, or training data change slightly?

Input sensitivity

How much does the prediction change if the input changes by a small amount? This is, in a single phrase, the question of adversarial robustness. Models with large input sensitivity, high

gradient norms with respect to input, can be fooled by small perturbations. Quantifying input sensitivity is a way of measuring how brittle a model is.

Weight sensitivity

How much does the prediction change if a weight changes? This question matters for quantization (where weights are stored in low precision), pruning (where weights are zeroed), and continual learning (where weights drift over time). High weight sensitivity means small changes break the model; low weight sensitivity gives you headroom.

Data sensitivity

How much does the trained model change if a single training example is changed or removed? This is the influence function perspective on machine learning, and it underlies tools for debugging training data, detecting outliers, and analyzing fairness.

Sensitivity and conditioning are the same idea

All three of these are conditioning questions in different guises. The condition number of a problem with respect to a particular input is exactly the sensitivity of its output to that input. Numerical analysis has a unified language for this kind of question, and machine learning would benefit from using it more often.

Key Takeaways

- Errors come in two kinds: truncation (shrinks with more effort) and rounding (cannot be eliminated).
- Forward error is what we want to know; backward error is what we can usually bound.
- Forward error ≈ backward error × condition number — reliable computation requires both stable algorithms AND well-conditioned problems.
- Backward stability analysis is the cleanest framework for evaluating numerical algorithms.

Exercises

- [Basic] Derive the optimal step size for the central difference approximation to f'(x) by balancing truncation and rounding error. What is it as a function of machine epsilon?
- [Intermediate] For a 3 × 3 matrix of your choice, compute its condition number using NumPy. Then construct a slightly perturbed right-hand side and observe the change in the solution.
- [Basic] Show that Newton's method for finding a root of a smooth function exhibits quadratic convergence in a neighborhood of the root.
- [Intermediate] Explain in your own words why backward error is often easier to bound than forward error.
- [Intermediate] Take a small neural network and compute the gradient of its output with respect to a single input pixel. How does this gradient relate to the network's adversarial robustness?

Chapter 3: Linear Algebra Refresher (Numerical Perspective)

Machine learning is, in large part, applied linear algebra. Every dense layer is a matrix multiplication. Every convolutional layer is a structured matrix multiplication. Every attention mechanism is a sequence of matrix products. Every gradient computation walks through a chain of linear maps. To understand what happens numerically in an ML system, we need to understand how matrix operations behave in finite precision, which is rather different from how they are presented in a pure linear algebra course.

3.1 Matrix Norms and Vector Norms

A norm is a way of measuring the size of a vector or matrix. The choice of norm affects how we measure error, how we define stability, and how we bound conditioning. The three norms that come up most often in machine learning are the 1-norm, 2-norm, and infinity-norm.

Vector norms

The 1-norm of a vector is the sum of the absolute values of its components. The 2-norm, also called the Euclidean norm or simply the norm, is the square root of the sum of squares. The infinity-norm is the maximum absolute value of any component.

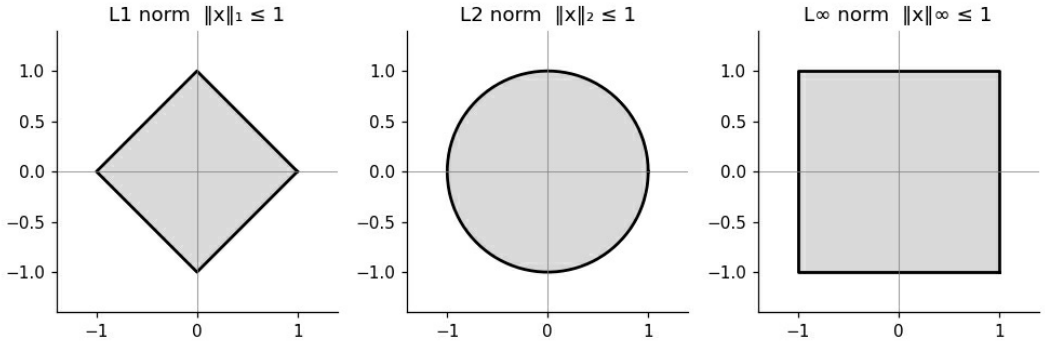


Figure 3.1: Unit balls for L1, L2, and L∞ norms.

$$\|x\|_1 = \Sigma |x_i|, \quad \|x\|_2 = \sqrt{(\Sigma\, x_i^2)}, \quad \|x\|_\infty = max|x_i|$$

These three norms are equivalent in the sense that bounding one bounds the others, but the constants depend on dimension and they have different geometric interpretations. The 1-norm and infinity-norm are cheap to compute and have favorable robustness properties; the 2-norm has the cleanest mathematical theory and is what we usually mean when we say "norm."

Matrix norms

Every vector norm induces a matrix norm by asking: what is the maximum amount this matrix can stretch a vector, measured in that norm? This is called the operator norm or induced norm.

$$\|A\| = max_{x \neq 0} \|Ax\| / \|x\|$$

For the 2-norm, this maximum is the largest singular value of A. For the 1-norm, it is the maximum absolute column sum. For the infinity-norm, it is the maximum absolute row sum. The Frobenius norm, the square root of the sum of all squared entries, is not an operator norm but is convenient because it is easy to compute and equals the square root of the sum of squared singular values.

Why we care

Matrix norms appear everywhere in numerical analysis and ML. The condition number is a ratio of norms. The Lipschitz constant of a linear map is its operator norm. Spectral normalization in GANs constrains the 2-norm of weight matrices. Gradient clipping uses the 2-norm of the gradient. Each application has a reason for choosing a particular norm; understanding the choice is part of understanding the algorithm.

3.2 Orthogonality and Projections

Two vectors are orthogonal if their inner product is zero. A matrix is orthogonal if its columns are mutually orthogonal unit vectors, equivalently, if its inverse equals its transpose. Orthogonal matrices are the cleanest objects in numerical linear algebra: they preserve norms, they are perfectly conditioned (condition number equals one), and they are immune to many of the instabilities that plague other operations.

Why orthogonality is numerically valuable

When we apply an orthogonal matrix to a vector, we do not amplify the vector's norm or its error. If x has rounding error δx, then Qx has rounding error $Q\delta x$, which has the same norm as δx. Compare this with a general matrix A, which can amplify error by a factor of $\kappa(A)$, potentially enormous.

This is why orthogonal transformations are the workhorses of stable numerical algorithms. QR decomposition, the SVD, Householder reflections, Givens rotations, all use orthogonal matrices to manipulate other matrices without introducing instability. When we want to reduce a matrix to triangular form for solving a linear system, we use orthogonal transformations rather than row eliminations whenever stability is paramount.

Projections

A projection matrix P satisfies $P^2 = P$. If P is symmetric, it is an orthogonal projection, it projects vectors onto a subspace along the orthogonal complement. Orthogonal projections are central to least squares: the projection onto the column space of A is $A(A^TA)^{-1}A^T$, and the least-squares solution to $Ax = b$ minimizes the distance between Ax and b.

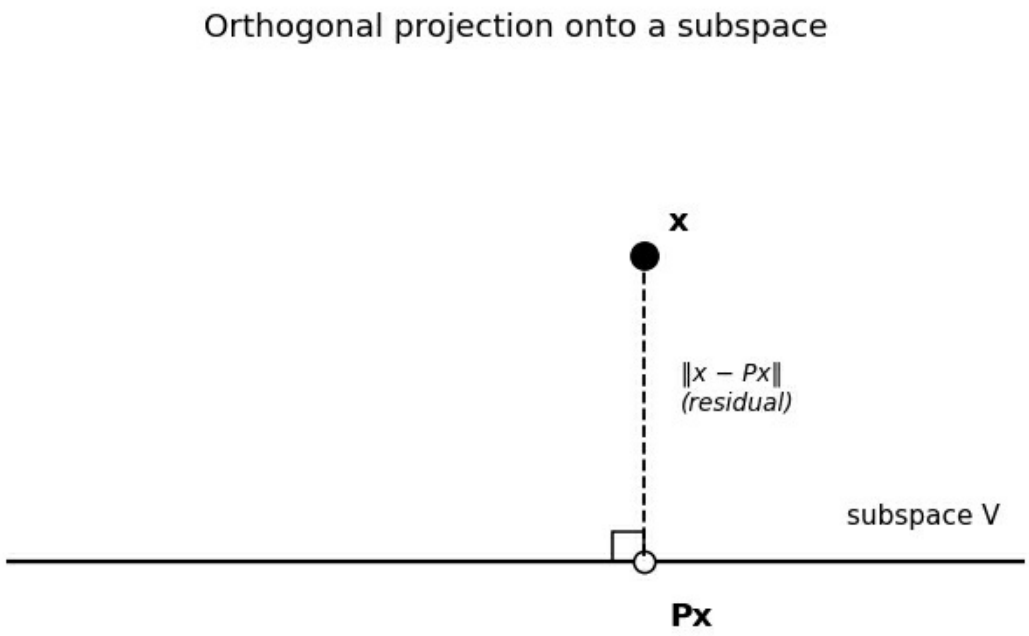


Figure 3.2: Orthogonal projection. The residual is perpendicular to the subspace.

In machine learning, projections appear in PCA (projection onto the leading singular vectors), in regularization (projection onto a constraint set), and in many optimization methods (projected gradient descent).

3.3 Eigenvalues and Eigenvectors

A scalar λ and a nonzero vector v are an eigenvalue and eigenvector of a matrix A if $Av = \lambda v$. Eigenvalues capture the directions in which a matrix acts as pure scaling. They are the most powerful tool in linear algebra for understanding what a matrix does, and they appear constantly in machine learning.

The spectrum

The set of all eigenvalues of a matrix is its spectrum. For an n by n matrix, the spectrum has n entries (counting multiplicity), though some may be complex. The spectral radius is the largest absolute value of any eigenvalue.

For a symmetric matrix, the eigenvalues are real and the eigenvectors can be chosen orthogonal, this is the spectral theorem, and it is one of the most useful results in all of mathematics. The eigendecomposition of a symmetric matrix A is $A = Q\Lambda Q^T$, where Q is orthogonal and Λ is diagonal.

Eigenvalues are not singular values

It is essential to keep eigenvalues and singular values distinct. Eigenvalues come from the eigenequation $Av = \lambda v$ and exist for any square matrix. Singular values come from the SVD $A = U\Sigma V^T$ and exist for any matrix, square or not. For a symmetric positive semidefinite matrix, the

singular values equal the eigenvalues. For a general matrix, they are different, and singular values are usually what we want for stability analysis.

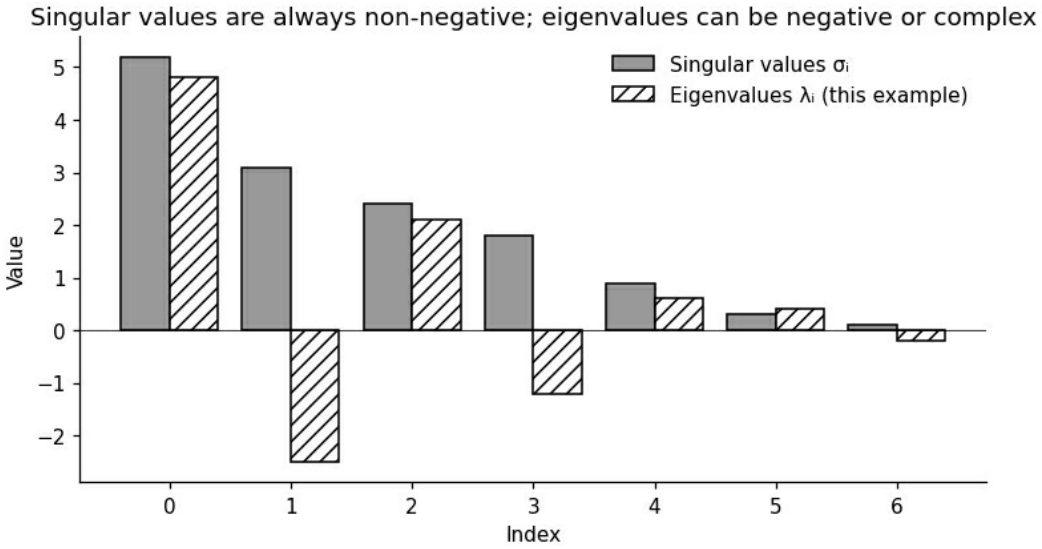


Figure 3.5: Singular values are always non-negative; eigenvalues can be negative or complex.

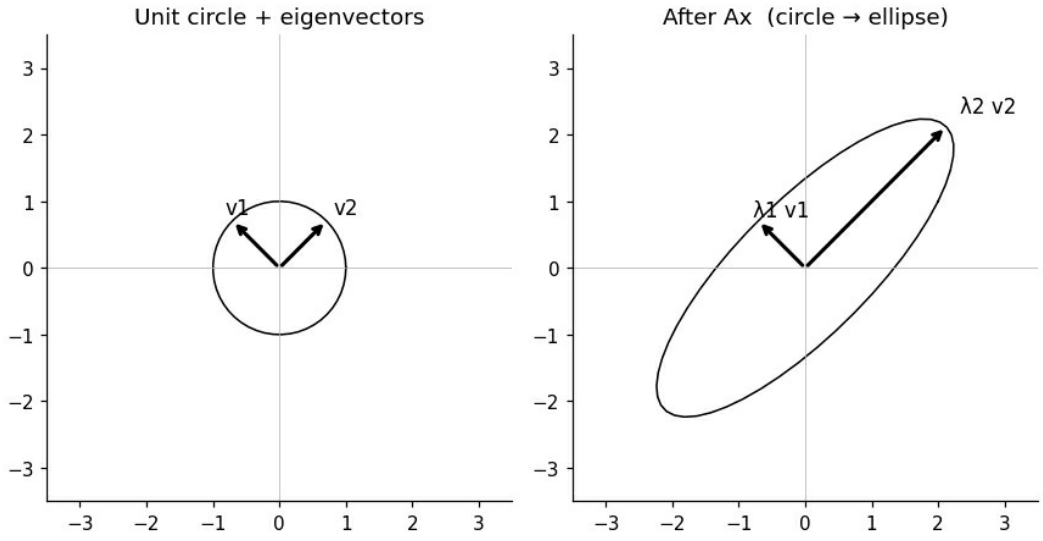


Figure 3.3: Eigenvectors are directions preserved by A; eigenvalues are the scaling factors.

Eigenvalues in ML

Some places where eigenvalues control machine learning systems:

- The Hessian of the loss function, whose eigenvalues determine the local curvature and the conditioning of optimization.
- The recurrence matrix in an RNN, whose spectral radius determines whether gradients explode or vanish.
- The covariance matrix of input features, whose eigenvalues are the variances along principal components.
- The graph Laplacian in a graph neural network, whose eigenvalues encode the structure of the graph.

3.4 Positive Definite Matrices

A symmetric matrix A is positive definite if $x^T Ax > 0$ for every nonzero vector x. Equivalently, all of its eigenvalues are positive. If we relax the strict inequality to ≥, we get positive semidefinite. These matrices are the well-behaved subset of symmetric matrices, and they are everywhere in machine learning.

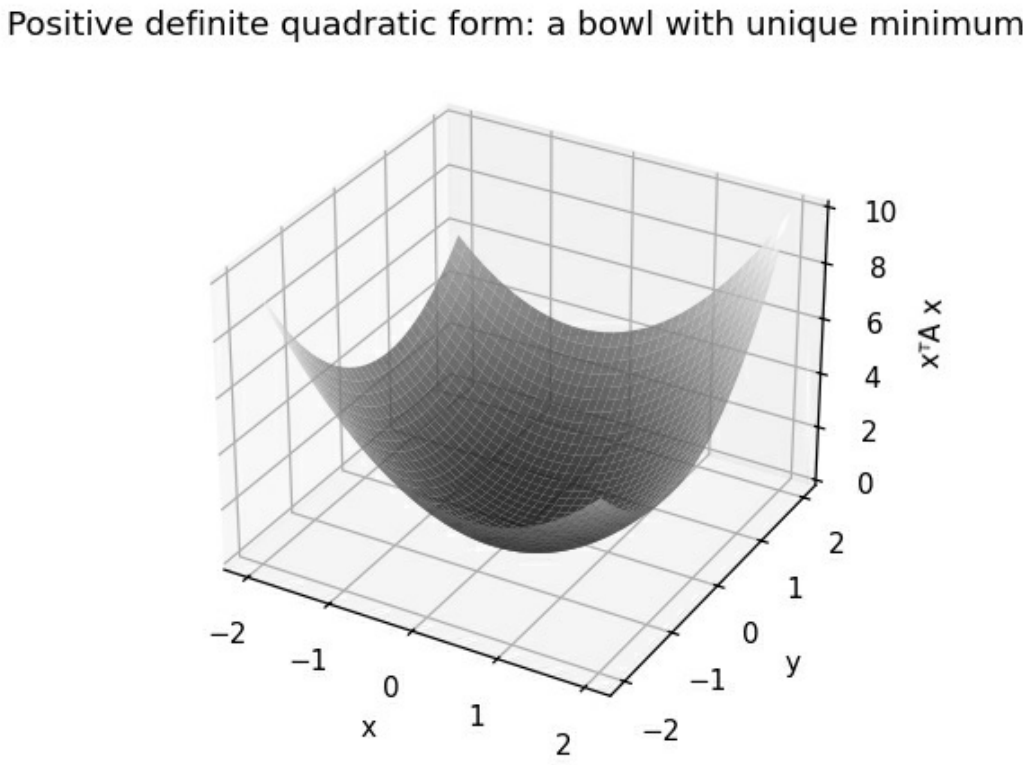


Figure 3.4: A positive definite quadratic form defines a bowl with a unique minimum.

Where they appear

- Covariance matrices are always positive semidefinite.
- Hessian matrices at a local minimum are positive semidefinite.
- Gram matrices $X^T X$ are always positive semidefinite.
- Kernels in kernel methods produce positive semidefinite matrices by construction.

Properties that matter numerically

Positive definite matrices admit the Cholesky decomposition: $A = L\,L^T$, where L is lower triangular. Cholesky is roughly twice as efficient as general LU decomposition and is more numerically stable. Whenever we know a matrix is positive definite, we should use Cholesky rather than general factorization.

They also have well-defined square roots: there is a unique positive definite matrix A^{1/2} such that A^{1/2} A^{1/2} equals A. Square roots come up in whitening, in the natural gradient, and in second-order optimization methods. Computing them stably is a nontrivial numerical problem, usually handled via the eigendecomposition or the Newton iteration on the matrix sign function.

3.5 Why Numerical Linear Algebra Is Not Textbook Linear Algebra

Standard linear algebra courses present matrix operations as if they were exact arithmetic on real numbers. This is a useful abstraction, but it is also misleading. The behavior of matrix algorithms in finite precision is qualitatively different in several ways.

Inverses are forbidden

Textbooks compute the inverse of A and then write A inverse times b. Working numerical code almost never computes an inverse, it solves the linear system Ax = b directly. The reasons are precision and efficiency: computing the inverse and then multiplying loses accuracy compared to solving directly, and the inverse takes more memory and computation than is needed.

Whenever you see a formula with a matrix inverse, mentally rewrite it as solving a system. The least-squares formula $\beta = (X^TX)^{-1} X^T y$ becomes solve $(X^TX)\ \beta = X^T y$. Better yet, solve it through QR or SVD without forming X^TX at all.

Determinants are also forbidden

Determinants are theoretically beautiful and numerically catastrophic. The determinant of an n by n matrix is, in general, the product of its singular values, which can be astronomically large or astronomically small. Use the log-determinant instead, computed from the diagonal of a triangular factorization. This is what every working library does.

Equality is conditional

Two matrices that are equal in pure linear algebra may differ at the level of machine epsilon when computed two different ways. Tests for equality should use a tolerance, never strict equality. Tests for properties, symmetry, positive definiteness, orthogonality, should also use tolerances and should be symmetric in their inputs.

Order matters

In pure mathematics, A(BC) equals (AB)C. In floating-point, the two are computed differently and may give different rounding errors. The order of operations affects both the cost (BC times A may be much cheaper than AB times C if the dimensions work out) and the precision.

The right factorizations

Numerical linear algebra has a small set of standard factorizations, LU, QR, Cholesky, SVD, eigendecomposition, and the art is choosing the right one for the problem. We will spend chapter 10 on this. For now: solve symmetric positive definite systems with Cholesky, solve general systems with LU (with partial pivoting), solve least squares with QR, and analyze rank or solve very ill-conditioned problems with SVD.

Key Takeaways

- The linear algebra of textbooks is exact; the linear algebra of computers is approximate.
- Orthogonal transformations preserve norms and condition; they are the safest tools in numerical linear algebra.
- Inverses and determinants are red flags — almost any problem stated with them has a better formulation.

• The choice of factorization (LU vs QR vs SVD) matters as much as the choice of algorithm.

• Treat matrix algorithms as numerical procedures, not as symbol manipulation.

Exercises

- [Basic] Compute the 1-norm, 2-norm, infinity-norm, and Frobenius norm of a 5×5 random matrix. Verify that the operator norms satisfy the inequality $\|A\|_2 \leq \sqrt{(\|A\|_1 \|A\|_\infty)}$.
- [Basic] Show that an orthogonal transformation preserves the 2-norm of any vector.
- [Intermediate] Construct a 4×4 symmetric matrix and compute its eigendecomposition. Verify that the eigenvectors are orthogonal.
- [Intermediate] Generate a positive definite matrix as $A^T A + 0.1\, I$ where A is random. Compute its Cholesky factor and verify $L\, L^T$ reproduces the matrix to machine precision.
- [Intermediate] For a poorly conditioned matrix, compare the accuracy of solving $Ax = b$ via (a) explicit inverse, (b) LU decomposition, (c) QR decomposition. Which is most accurate?

Part II

Core Optimization Methods

Optimization is the engine of machine learning. We pose learning as the minimization of a loss function, and the algorithms that perform that minimization determine how well, and how quickly, our models train. The next five chapters cover this terrain: the geometry of loss landscapes, the gradient methods that dominate practice, the second-order methods that promise faster convergence, and the constrained methods that handle the regularization and structure that real problems demand.

Chapter 4: The Optimization Landscape in Machine Learning

To choose an optimization algorithm intelligently, we have to understand what we are optimizing over. The shape of the loss function, its curvature, its critical points, the behavior far from the optimum, determines which algorithms work and which do not. This chapter is a tour of the landscape: convex and non-convex, low-dimensional and high-dimensional, well-conditioned and pathological.

4.1 Loss Functions and Their Geometry

A loss function takes the parameters of a model and returns a number that quantifies how well the model fits the data. Training is the search for parameters that make this number small. Different choices of loss function produce different landscapes, and the geometry of those landscapes matters at least as much as the algorithm we use to search them.

Squared loss

Squared loss, the sum of squared residuals, produces a quadratic landscape when the model is linear. The level sets are concentric ellipsoids, the unique minimum sits at the bottom, and there are no local minima other than the global one. This is the friendliest possible setting and the reason linear regression is solved analytically.

Cross-entropy

Cross-entropy loss for classification is convex when the model is linear (logistic regression), but non-convex once we wrap it in a deep network. The convex case has a unique minimum that is found efficiently; the non-convex case has many local minima and saddle points, and finding any of them is the best we can hope for in general.

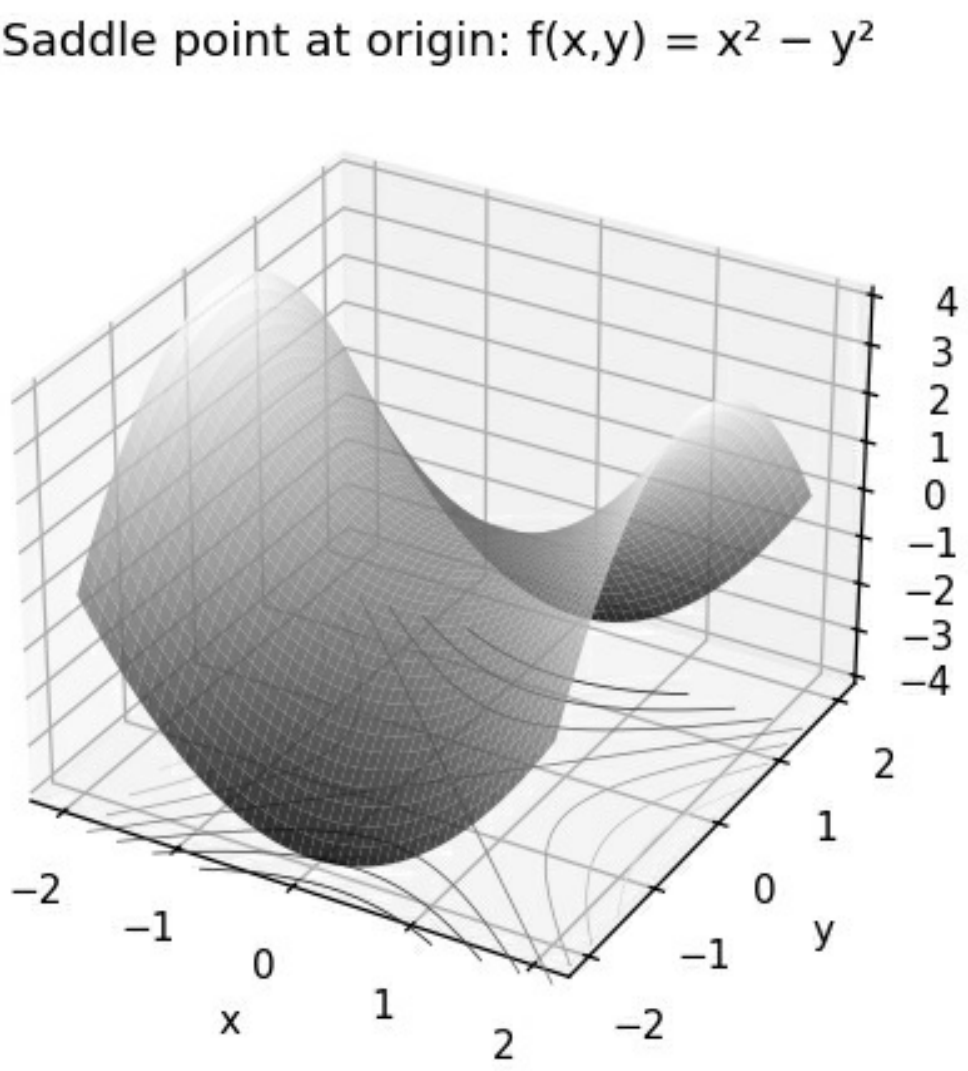


Figure 4.3: A saddle point: f(x, y) = x² − y². Gradient is zero at the origin, but the Hessian has both positive and negative eigenvalues.

Adversarial and minimax losses

Generative adversarial networks, robust optimization, and game-theoretic formulations of learning produce minimax losses, in which we want to minimize over one set of parameters while maximizing over another. These are not even guaranteed to have equilibria in the deep network case, and the optimization is far harder than ordinary minimization.

4.2 Convex vs Non-Convex Optimization

A function is convex if, for any two points in its domain, the function value at the midpoint is at most the average of the function values at the endpoints. Geometrically, the function lies below any chord drawn between two points on its graph. Convex functions have a single global minimum (or a flat valley of minima), no local minima other than the global one, and no saddle points.

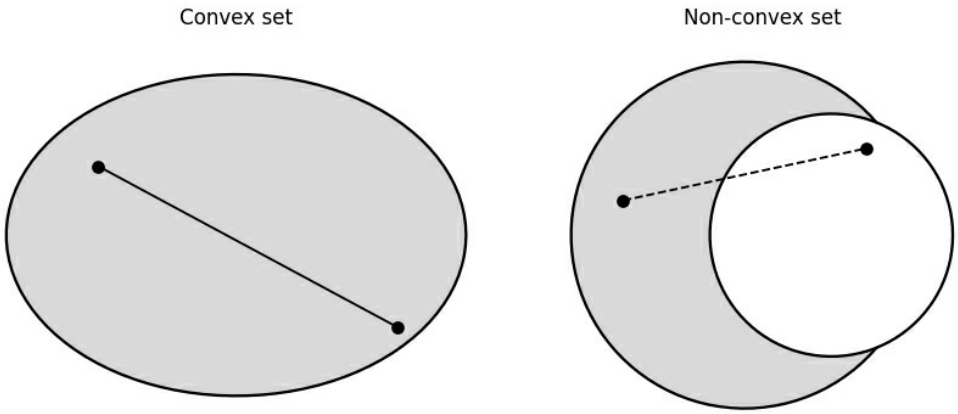


Figure 4.2: Convex set (left) contains the line segment between any two of its points. Non-convex set (right) does not.

Why convexity is a gift

Convex optimization is, for practical purposes, a solved problem. We have algorithms, gradient descent, interior point methods, ADMM, that converge to the global minimum at known rates and with known precision. For convex problems, the question "did the algorithm find the right answer?" almost always has the answer yes, given enough iterations.

Why machine learning is not convex

Modern neural networks are non-convex, fundamentally and irreversibly. The composition of a linear map and a nonlinear activation breaks convexity. Once you stack more than one layer of nonlinearity, the loss surface develops local minima, saddle points, and flat regions. Pretending otherwise leads to bad theoretical predictions and worse engineering decisions.

And yet deep learning works. The empirical fact that gradient-based methods reliably find good minima of high-dimensional non-convex losses, minima that generalize well, even, is one of the most important and least understood phenomena in modern machine learning. Several explanations have been proposed: that overparameterization smooths the landscape, that good minima are connected by low-loss paths, that the noise in stochastic gradient descent acts as an implicit regularizer. None of these is a complete theory, but together they account for much of what we observe.

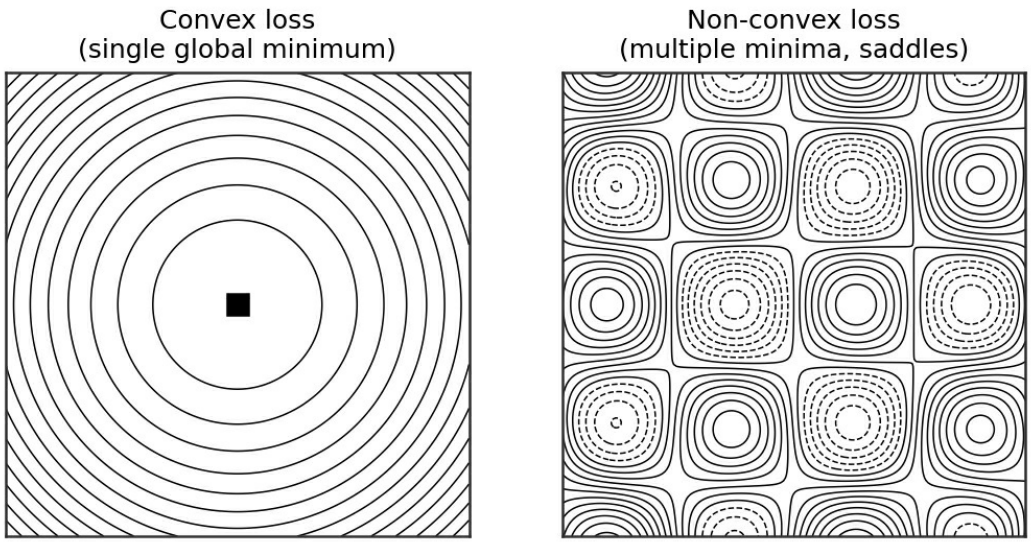


Figure 4.1: Convex loss (left) has a single basin; non-convex loss (right) has multiple minima and saddle points.

4.3 Saddle Points vs Minima

In low-dimensional optimization, we worry about local minima, points where the gradient is zero and the function curves up in every direction. In high-dimensional non-convex optimization, the more important obstacle is saddle points, points where the gradient is zero but the function curves up in some directions and down in others.

Why saddle points dominate in high dimensions

At a critical point of a smooth function, the Hessian has eigenvalues whose signs determine the local geometry. A local minimum has all positive eigenvalues; a local maximum has all negative; a saddle point has at least one of each. In high dimensions, the probability that all n eigenvalues are positive (or all negative) is astronomically small if the eigenvalues are roughly randomly distributed. The vast majority of critical points are saddles.

This is, paradoxically, good news. Saddle points are escapable, there is always a direction of descent, whereas local minima are not. Theoretical work has shown that, for many problems, gradient descent escapes saddle points relatively quickly. The bad news is that the escape can take many iterations and produces the long plateaus in training curves that look like the model has stopped learning.

4.4 Ill-Conditioning in Deep Learning

The condition number of the Hessian of the loss is one of the most important quantities controlling the difficulty of training. Recall that the condition number is the ratio of the largest to smallest eigenvalue. A well-conditioned Hessian has eigenvalues all of similar size; the loss surface looks roughly spherical, and gradient descent works well. An ill-conditioned Hessian has some eigenvalues much larger than others; the loss surface is a long, narrow valley, and gradient descent zigzags ineffectively along it.

Why deep networks are ill-conditioned

Deep networks are ill-conditioned for structural reasons. Different weights affect the loss through

chains of different lengths, and the effective scale of their contribution differs by orders of magnitude. A weight in the first layer affects the output through a chain of dozens of multiplications; a weight in the last layer affects it directly. The result is a Hessian whose eigenvalues span a huge range.

The condition number of a typical deep network can be 10^6 or worse. Plain gradient descent on such a problem would crawl. The interventions that make deep learning work, careful initialization, batch normalization, adaptive learning rates, residual connections, can all be understood, at least in part, as preconditioning techniques that reduce effective conditioning of the optimization problem.

Key Takeaways

- The ML optimization landscape is non-convex, high-dimensional, and ill-conditioned.
- Local minima are less of a problem than they appear; saddle points and pathological curvature are pervasive.
- Sharp minima generalize worse than flat ones — landscape geometry affects both convergence and generalization.
- Most algorithmic interventions in deep learning are responses to landscape pathology, not abstract optimization choices.

Exercises

- [Basic] Show that a quadratic function $f(x) = (1/2)\, x^T A\, x + b^T x$ is convex if and only if A is positive semidefinite.
- [Intermediate] Construct a 2D non-convex function with a saddle point. Plot its level sets and verify that gradient descent from a generic starting point passes near the saddle.
- [Basic] Compute the eigenvalues of the Hessian of the loss for a small linear regression problem. Relate the condition number to the eigenvalues of X^TX.

Chapter 5: Gradient-Based Methods

Gradient descent is the foundation of machine learning optimization. Almost every model trained today uses some variant of it. Despite its simplicity, at every step, move in the direction opposite to the gradient, its behavior on real loss surfaces is surprisingly subtle. This chapter is a careful examination of the algorithm in its three principal forms (batch, stochastic, and mini-batch), the question of how to choose a learning rate, and the specific failures that affect deep networks.

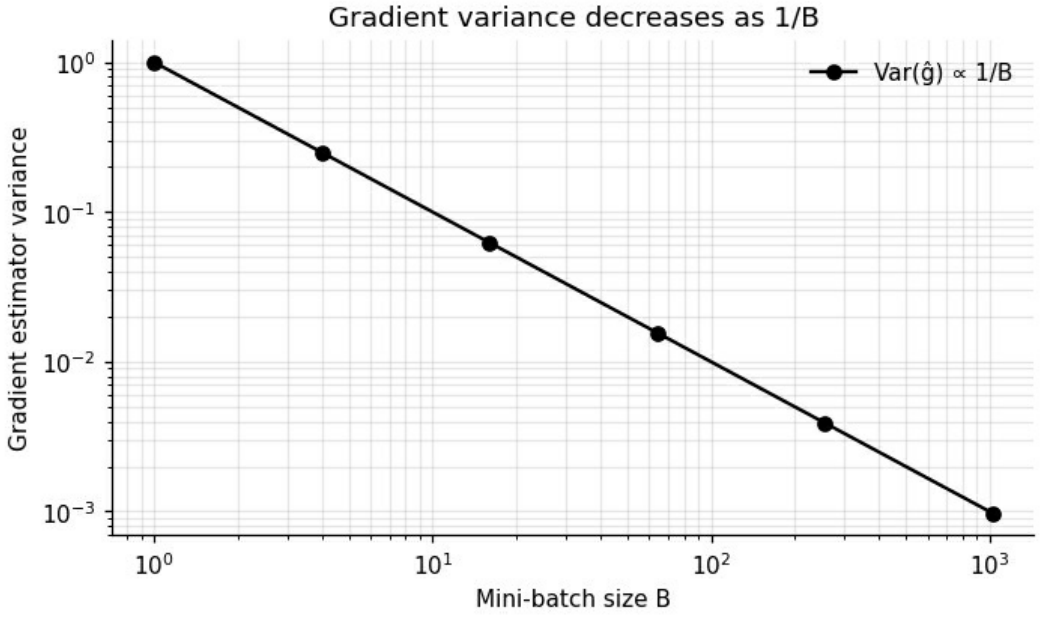


Figure 5.3: Gradient-estimator variance decreases as 1/B in batch size B.

5.1 The Update Rule

The gradient descent update is the simplest possible iterative optimization step. At iteration k, given the current parameters x_k, compute the gradient of the loss, and take a step in the opposite direction:

$$x_{k+1} = x_k - \eta \nabla f(x_k)$$

The scalar η is the learning rate, also called the step size. The algorithm is essentially a discrete approximation to the continuous gradient flow that descends the loss surface in the steepest direction. Everything else in this chapter is a refinement of this idea.

What gradient descent assumes

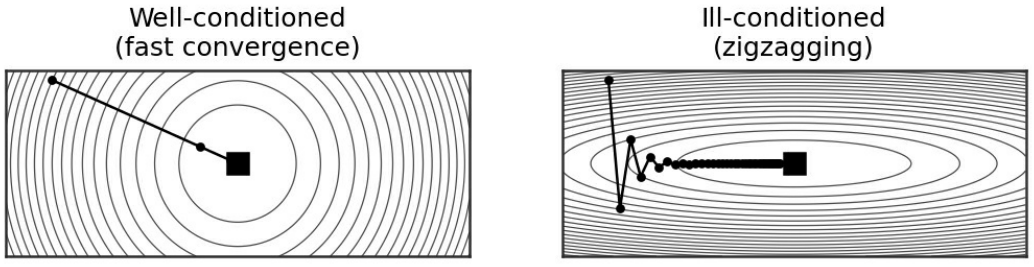


Figure 5.1: Gradient descent in a narrow valley zigzags across the long axis instead of moving directly toward the minimum.

The update rule treats the loss surface as locally linear: it follows the slope at the current point and trusts that this direction leads to lower loss for some distance. This works when the loss is smooth and the step size is small enough. It fails when the surface is sharply curved (we overshoot), when the gradient is noisy (we wander), or when the gradient is zero at a non-optimum (saddle points).

5.2 Batch, Stochastic, and Mini-Batch

In machine learning, the loss is typically a sum (or average) over training examples. The gradient is correspondingly a sum over examples. How many examples we use to compute each step gives us three variants of gradient descent.

Batch gradient descent

Batch gradient descent uses the gradient computed over all training examples at every step. This gives the most accurate estimate of the true gradient, but it is computationally prohibitive for any reasonable-sized dataset. Each step requires a full pass through the data, and modern training sets have millions of examples. Batch gradient descent is the textbook algorithm and almost never used in practice for deep learning.

Stochastic gradient descent

Stochastic gradient descent (SGD), in its purest form, uses a single training example at each step. The gradient computed from one example is an unbiased but high-variance estimate of the true gradient. The trajectory of SGD therefore looks noisy compared to batch gradient descent, but each step is fast, and over many steps the noise averages out.

SGD has properties that batch gradient descent lacks. The noise helps escape saddle points and shallow local minima. The fast iteration allows much more progress per unit of compute. And, perhaps most importantly, the noise acts as an implicit regularizer that helps generalization. There is a growing body of work suggesting that SGD's noise is not just tolerable but essential to why deep networks generalize.

Mini-batch gradient descent

Mini-batch gradient descent, the workhorse of practical deep learning, uses a batch of size between, say, 32 and a few thousand examples per step. This balances the variance reduction of larger batches against the per-step cost. Modern hardware (GPUs, TPUs) is designed to process mini-batches in parallel, so the per-step cost is roughly constant in batch size up to the hardware's parallel capacity.

The choice of batch size is more consequential than it appears. Larger batches give more accurate gradients but reduce the number of update steps for a given compute budget. Smaller batches give more steps but each step is noisier. Recent work suggests there is an optimal batch size that depends on the problem, with the loss landscape's noise scale playing a key role.

5.3 Choosing the Learning Rate

The learning rate is the most important hyperparameter in deep learning. A learning rate that is too small wastes time; a learning rate that is too large causes divergence. Between these extremes lies a narrow range where training is both stable and fast, and finding that range is the central practical challenge of optimization.

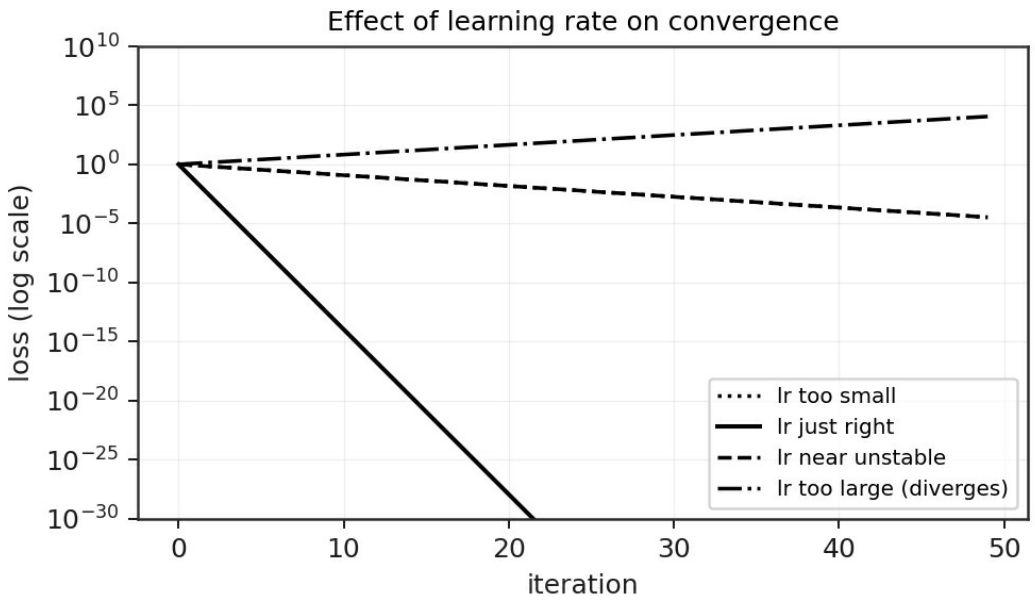


Figure 5.2: Effect of learning rate. Too small converges slowly; too large diverges; the sweet spot decreases loss rapidly without overshooting.

The mathematical bound

For a convex loss with Lipschitz gradient (the gradient does not change too quickly), gradient descent converges to the global minimum if the learning rate is at most 2/L, where L is the Lipschitz constant. The optimal rate is 1/L, and at this rate, the convergence is linear with rate 1 - μ/L, where μ is the strong convexity constant.

For non-convex problems, no such clean bound exists, but the same intuition applies: the learning rate should be inversely proportional to the largest eigenvalue of the Hessian. In deep learning, the largest Hessian eigenvalue can change dramatically during training, which is why fixed learning rates rarely work well.

Practical recipes

- Learning rate range test: train briefly across a range of learning rates and pick one near the largest rate that keeps the loss decreasing.

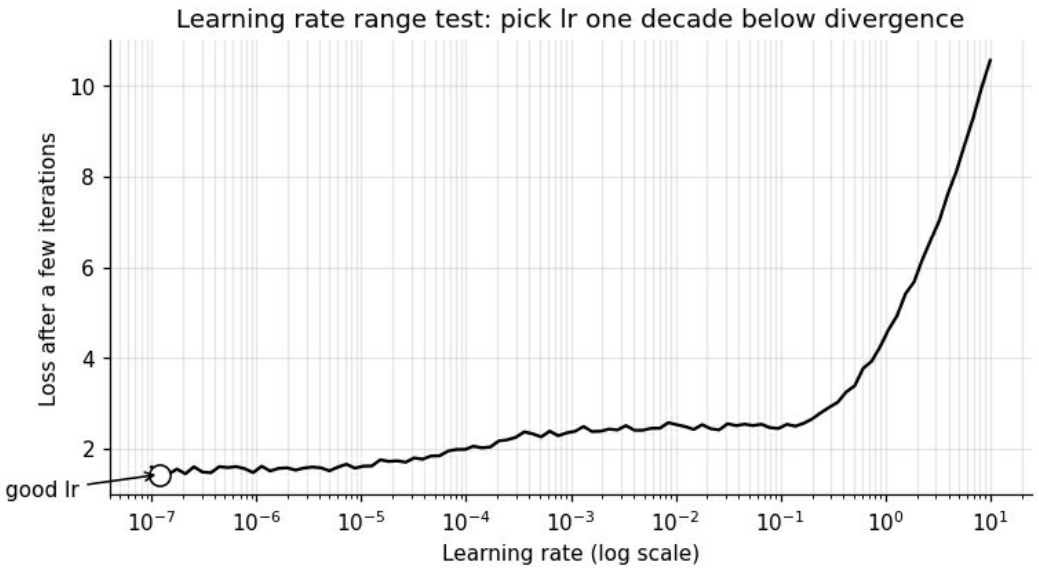


Figure 5.5: Learning rate range test. Pick lr one decade below where loss diverges.

- Warm-up: start with a small learning rate and gradually increase it over the first thousand or so steps. This gives early-training instabilities time to settle.
- Decay: reduce the learning rate over training. Common schedules include step decay, exponential decay, cosine annealing, and decay-on-plateau.
- Cyclical schedules: vary the learning rate up and down during training, on the theory that occasional high learning rates help escape sharp minima.

5.4 Convergence Intuition

Gradient descent on a quadratic provides the cleanest possible setting for understanding convergence. If the loss is $f(x) = (1/2)\ x^T\ A\ x$ with A symmetric positive definite, the eigendecomposition of A diagonalizes the problem: in eigenvector coordinates, gradient descent is independent in each direction, and the convergence rate in direction i is $1 - \eta\ \lambda_i$.

For convergence to occur, we need $0 < \eta\ \lambda_i < 2$ for every eigenvalue λ_i. This means η must be at most $2/\lambda_max$. For fastest convergence overall, we want to be as aggressive as possible in slow-converging directions while staying stable in fast-converging directions. The optimal compromise is $\eta = 2/(\lambda_max + \lambda_min)$, and the resulting convergence rate is $(\kappa - 1)/(\kappa + 1)$, where κ is the condition number of A.

This is a beautiful result with a brutal implication: convergence rate is controlled by the condition number. A condition number of 1000, modest by deep learning standards, gives convergence rate (1000-1)/(1000+1) ≈ 0.998 per step. Reducing the loss by a factor of e takes about 500 steps. Reducing it by 10^6 takes 7000 steps. Now imagine condition numbers of a million. This is why preconditioning, in any of its forms, is essential.

5.5 Vanishing and Exploding Gradients

In deep networks, the gradient is computed by backpropagation, which is essentially a chain of matrix products. If the Jacobian at each layer has a particular spectral character, the gradient grows or shrinks exponentially with depth.

Vanishing gradients

If the Jacobians at each layer have spectral radius less than one, as happens with sigmoid activations in their saturated regime, the gradient shrinks geometrically as it propagates backward. By the time it reaches the early layers, it may be 10^-30 or smaller, indistinguishable from zero in float32 and giving no signal for training. The early layers stop learning.

This was the dominant problem in deep learning before about 2015. The fixes that resolved it, ReLU activations (which have unit Jacobian in the active region), careful initialization, batch normalization, residual connections, are now standard, but understanding why they work requires understanding the vanishing gradient problem they solve.

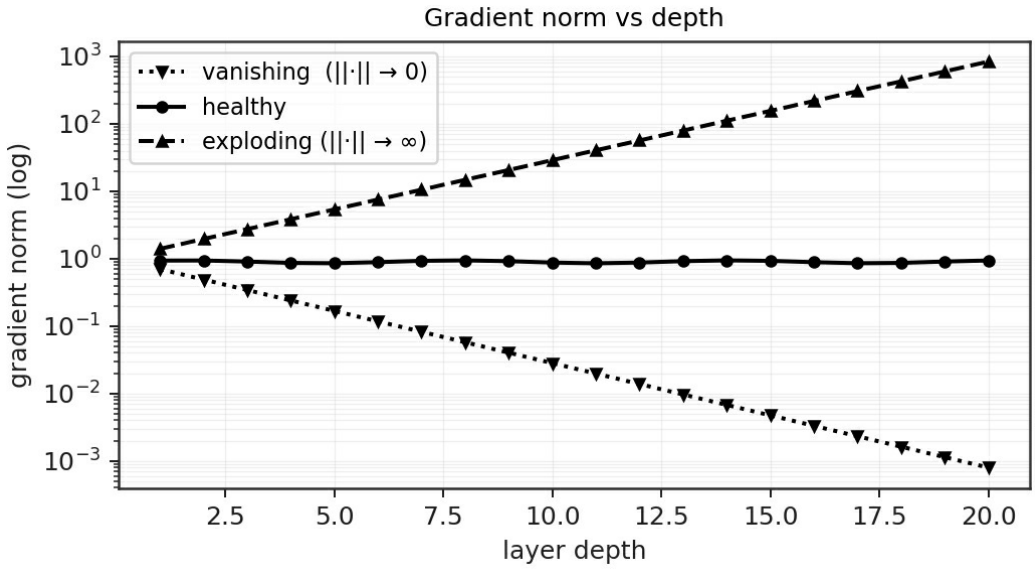


Figure 5.4: Gradient magnitude across layers in deep networks. Without proper initialization, gradients vanish or explode in early layers.

Exploding gradients

The mirror image: if the layer Jacobians have spectral radius greater than one, the gradient grows geometrically. We saw this in chapter 1 in the context of RNNs. The gradient overflows; the weights update to garbage; training dies.

The fix is gradient clipping: scale the gradient down to a maximum norm before applying the update. This sacrifices the magnitude information but preserves the direction, and in practice it suffices to keep training alive in regimes where exploding gradients would otherwise kill it.

```
# Gradient clipping in PyTorch
loss.backward()
torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
optimizer.step()
```

The structural fix: residual connections

Residual connections, introduced in ResNet, change the geometry of the gradient flow. Instead of computing y = f(x), each layer computes y = x + f(x). The Jacobian becomes I + Jf, which has eigenvalues clustered near 1 unless f's Jacobian is large. Gradients propagate through identity-plus-perturbation rather than through products of arbitrary matrices, and both vanishing and exploding behaviors are dramatically suppressed.

The success of residual connections is one of the great empirical lessons of deep learning. Architectural choices have consequences for optimization that no amount of clever learning rate scheduling can replicate.

Key Takeaways

- Gradient descent is a one-line algorithm with a lifetime of subtleties.
- Batch size trades variance for steps; learning rate trades stability for speed.
- Conditioning sets fundamental limits on convergence rate that no learning-rate tuning can escape.
- The interventions in deep learning — initialization, normalization, residual connections — are largely responses to the conditioning problem.
- Always run a learning-rate range test before committing to a fixed schedule.

Exercises

- [Intermediate] Implement gradient descent on a 2D quadratic with eigenvalues 1 and 100. Plot the trajectory. Verify that with the optimal learning rate, the trajectory zigzags in characteristic fashion.
- [Intermediate] Run mini-batch SGD on a logistic regression problem with batch sizes 1, 32, and 1024. Compare convergence behavior.
- [Intermediate] Implement a learning rate range test. Apply it to a small neural network and identify the optimal learning rate range.
- [Advanced] Train a deep MLP without residual connections, with Sigmoid activations and depth 30. Observe gradient magnitudes by layer. Then add ReLU and residual connections and compare.
- [Basic] Derive the optimal step size $2/(\lambda_{max} + \lambda_{min})$ for gradient descent on a quadratic, and compute the resulting convergence rate.

Chapter 6: Advanced Gradient Methods

Plain gradient descent has well-known failure modes: it crawls in ill-conditioned directions, it gets stuck on saddle points, and it requires careful learning-rate tuning. The advanced gradient methods in this chapter, momentum, Nesterov acceleration, AdaGrad, RMSProp, and Adam, address these failures in different ways. Each is the dominant choice in some regime; understanding when each is appropriate is one of the most useful skills an ML practitioner can develop.

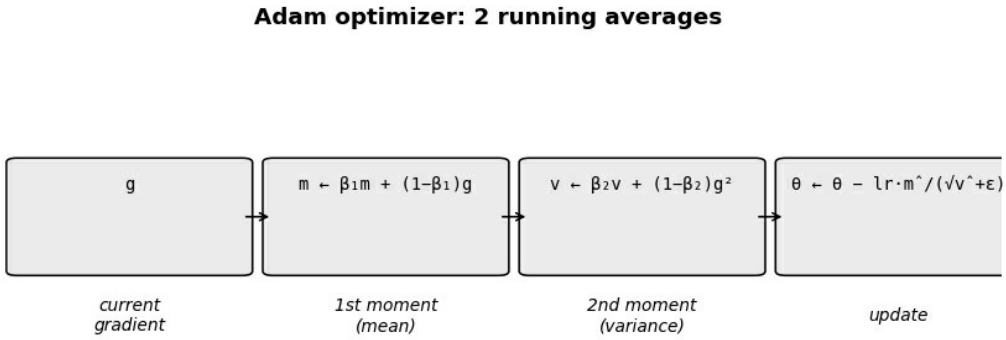


Figure 6.2: Adam maintains two running averages: first moment and second moment of the gradient.

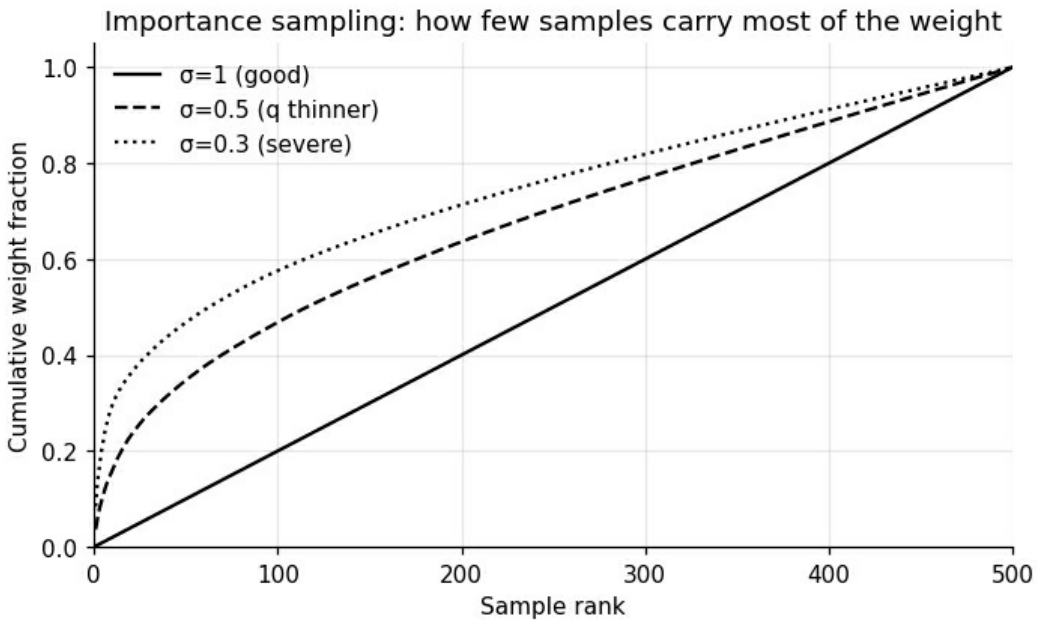


Figure 6.3: Convergence on an ill-conditioned quadratic. Gradient descent converges at rate (κ−1)/(κ+1); momentum and Nesterov achieve O(√κ).

6.1 Momentum

Momentum is the simplest and oldest of the accelerated methods. The idea is intuitive: instead of using the gradient as the direction of update, accumulate past gradients into a velocity vector and update in the direction of velocity. The update rules are:

$$v_{k+1} = \beta\, v_k + \nabla f(x_k)$$
$$x_{k+1} = x_k - \eta\, v_{k+1}$$

The parameter β is the momentum coefficient, typically between 0.9 and 0.99. When β is zero, momentum reduces to plain gradient descent. As β approaches one, the velocity becomes a moving average of all past gradients.

Why it works

Momentum has two effects, both useful. First, in narrow valleys where consecutive gradients point in similar directions, the velocity accumulates and the effective step size grows. This accelerates progress along the valley floor. Second, in directions where the gradient oscillates (the high-curvature directions of an ill-conditioned problem), the alternating gradients partially cancel in the velocity, damping the oscillation.

The mathematical analysis on quadratics shows that momentum can change the convergence rate from (κ-1)/(κ+1) to (√κ-1)/(√κ+1), which is much better for ill-conditioned problems. Where plain gradient descent on a problem with condition number 10^6 takes about 7 million steps to reduce the loss by a factor of e, momentum takes about 7 thousand.

6.2 Nesterov Acceleration

Nesterov accelerated gradient (NAG) is a refinement of momentum proposed by Yurii Nesterov in 1983. The idea is small but mathematically deep: instead of computing the gradient at the current point and then moving in the velocity direction, look ahead in the velocity direction first and compute the gradient there.

$$v_{k+1} = \beta\, v_k + \nabla f(x_k - \eta\, \beta\, v_k)$$
$$x_{k+1} = x_k - \eta\, v_{k+1}$$

This look-ahead gives the gradient a chance to correct the trajectory before committing to the next step. If the velocity is taking us toward a region where the gradient changes direction, we see that change one step earlier than ordinary momentum would.

Nesterov's analysis shows that NAG achieves the optimal convergence rate among all first-order methods: it cannot, in general, be beaten by any algorithm using only gradient information. For convex problems, this is a complete theory. For non-convex deep learning, the theoretical guarantee does not hold, but NAG remains an attractive choice in practice and is the foundation of Nesterov-style optimizers.

6.3 Adaptive Methods: AdaGrad, RMSProp, and Adam

The methods so far use a single learning rate for all parameters. Adaptive methods give each parameter its own effective learning rate, computed from the history of its gradients. The intuition is that parameters whose gradients have been consistently large should have small learning rates (so they do not take excessive steps) and parameters whose gradients have been consistently small should have large learning rates (to make up the difference).

AdaGrad

AdaGrad, introduced by Duchi, Hazan, and Singer in 2011, was the first widely used adaptive method. It maintains, for each parameter, the sum of squared gradients seen so far, and divides the learning rate by the square root of this sum:

$$G_k = G_{k-1} + (\nabla f(x_k))^2$$
$$x_{k+1} = x_k - \eta \, \nabla f(x_k) / (\sqrt{G_k + \varepsilon})$$

AdaGrad's strength is that it works well on sparse problems, parameters that rarely receive gradient signal get larger effective learning rates and update efficiently when they do. Its weakness is that the accumulator only grows; eventually all effective learning rates shrink to zero, and learning stops. This is fatal for deep learning, where we want to keep training for many epochs.

RMSProp

RMSProp, proposed by Hinton in a Coursera lecture, fixes AdaGrad's monotonically shrinking learning rate by replacing the sum with an exponential moving average. The accumulator forgets old gradients, so it can grow or shrink as appropriate:

$$G_k = \rho \, G_{k-1} + (1 - \rho) \, (\nabla f(x_k))^2$$
$$x_{k+1} = x_k - \eta \, \nabla f(x_k) / (\sqrt{G_k + \varepsilon})$$

RMSProp was a major improvement and was widely used in early deep learning. It is still a reasonable default for some applications, particularly those with non-stationary objectives.

Adam

Adam (Kingma and Ba, 2014) combines momentum with RMSProp-style adaptive learning rates. It maintains both an exponential moving average of the gradient (momentum) and an exponential moving average of the squared gradient (RMSProp), with bias correction for the first few steps when the moving averages are still warming up:

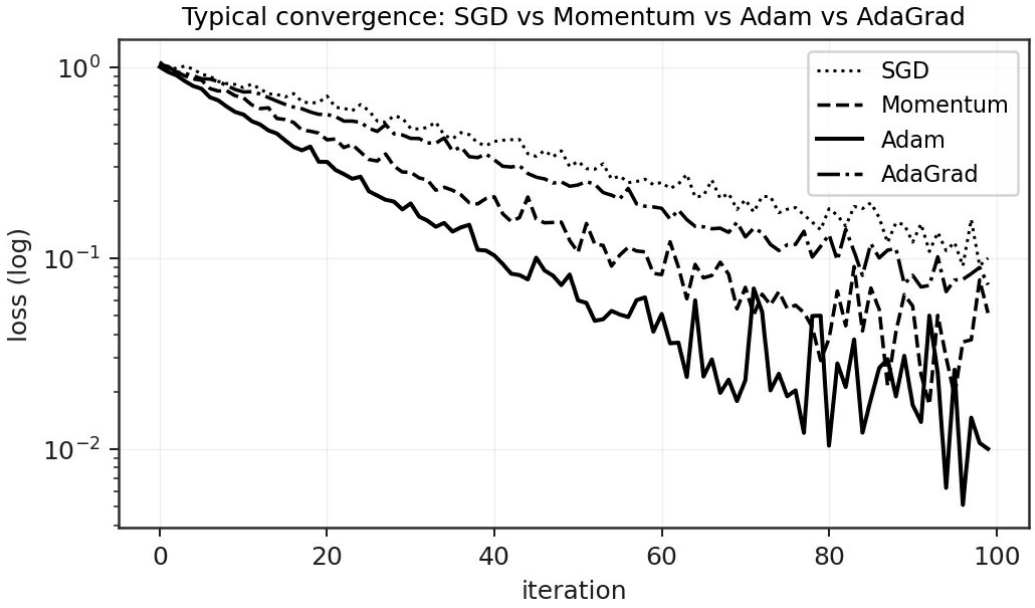


Figure 6.1: Convergence trajectories of SGD, momentum, RMSProp, and Adam on a representative loss surface.

$$m_k = \beta_1 \, m_{k-1} + (1-\beta_1) \, \nabla f(x_k)$$
$$v_k = \beta_2 \, v_{k-1} + (1-\beta_2) \, (\nabla f(x_k))^2$$
$$\hat{m} = m_k / (1 - \beta_1^k), \quad \hat{v} = v_k / (1 - \beta_2^k)$$
$$x_{k+1} = x_k - \eta \, \hat{m} / (\sqrt{\hat{v}} + \varepsilon)$$

Adam has become the default optimizer for deep learning, and for good reason. It works well across a wide range of problems with minimal tuning, it handles non-stationary objectives, and its adaptive learning rates respond to the per-parameter scale of gradients.

AdamW

A subtlety: the original Adam paper combined adaptive gradients with L2 regularization in a way that turns out to be incorrect for adaptive methods. The fix, AdamW (Loshchilov and Hutter, 2019), decouples weight decay from the gradient and applies it as a separate update step. AdamW is now preferred to Adam in most training pipelines that use weight decay.

6.4 When Adaptive Methods Fail

Despite its dominance, Adam is not always the right choice. There are documented cases where SGD with momentum outperforms Adam, sometimes substantially, and the difference appears not to be just a matter of tuning.

Generalization gap

Adam often achieves lower training loss than SGD but worse test accuracy. The gap appears in image classification, where well-tuned SGD with momentum is the standard for state-of-the-art models. The conjectured explanation is that the adaptive learning rates encourage convergence to sharper minima, which generalize less well.

Pathological learning rate behavior

In some problems, the second-moment estimate v can become very small for a while, leading to very large effective learning rates and divergence. The original Adam paper missed a convergence issue that was later fixed in AMSGrad (Reddi et al., 2018), which uses the maximum of past v values rather than the moving average.

The practitioner's guidance

- For most deep learning problems, AdamW with default hyperparameters is a strong starting point.
- For state-of-the-art image classification, SGD with Nesterov momentum and a careful schedule remains competitive or better.
- For training large language models, Adam variants are the standard, often with specific schedules and warm-ups.
- For convex problems, plain SGD with a 1/k learning rate has the strongest theoretical guarantees and is often hard to beat in practice.
- When in doubt, try both Adam and SGD-with-momentum; the choice matters more often than people expect.

Key Takeaways

• Momentum accumulates past gradients into a velocity, accelerating along narrow valleys.

• Nesterov acceleration adds look-ahead and achieves the optimal first-order convergence rate.

• Adaptive methods (AdaGrad, RMSProp, Adam) give each parameter its own effective learning rate.

• Adam is the modern default, but well-tuned SGD with momentum often outperforms it on

convex or near-convex problems.

• Always use bias-corrected versions of adaptive methods at warmup.

Exercises

- [Intermediate] Implement momentum and compare it with plain gradient descent on a 2D quadratic with high condition number. Verify the predicted speedup.
- [Intermediate] Implement Nesterov accelerated gradient and compare it with momentum. Construct a problem where Nesterov is noticeably faster.
- [Intermediate] Implement AdaGrad, RMSProp, and Adam from scratch and verify them against PyTorch's implementations on a small problem.
- [Intermediate] Find a problem where SGD with momentum generalizes better than Adam. (Hint: ResNet on CIFAR-10 is a classic example.)
- [Research-Level] Read the AMSGrad paper and explain in your own words the convergence issue it addresses.

Chapter 7: Second-Order Methods

First-order methods use only the gradient: information about the slope of the loss at the current point. Second-order methods also use the Hessian: information about the curvature. With curvature information, an optimizer can take better-shaped steps, long in flat directions, short in steep directions, and in principle achieve much faster convergence. The catch is that the Hessian is expensive to compute and store. The history of second-order optimization is the history of clever approximations that capture the benefits without paying the full cost.

7.1 Newton's Method

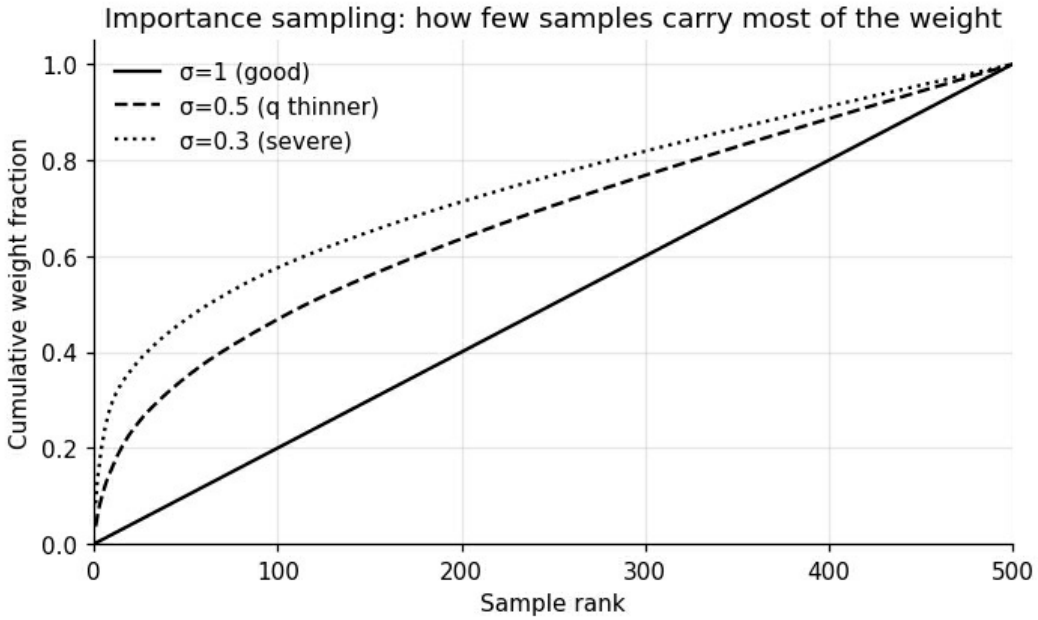


Figure 7.1: Convergence comparison. Newton's method exhibits quadratic convergence near the optimum; gradient descent is linear.

Newton's method is the canonical second-order optimizer. It applies the second-order Taylor expansion of the loss around the current point, finds the minimum of that quadratic approximation, and moves there:

$$x_{k+1} = x_k - H(x_k)^{-1} \nabla f(x_k)$$

Geometrically, Newton's method models the loss as a quadratic and jumps to the minimum of that quadratic in a single step. When the actual loss is quadratic, Newton's method converges in one step. When the loss is locally well-approximated by a quadratic, Newton's method converges quadratically, the number of correct digits doubles per iteration.

Why we cannot simply use Newton's method

In machine learning, Newton's method is rarely usable directly, for three reasons.

First, the Hessian of a deep network has size n by n, where n is the number of parameters. For a model with a billion parameters, the Hessian has 10^18 entries, far too many to store. Even computing one row of the Hessian-vector product requires a backward pass through the network.

Second, the Hessian may not be positive definite. If it has negative eigenvalues, as it does at

saddle points, Newton's method moves toward the saddle, not away from it. We need to modify the algorithm to handle indefinite Hessians.

Third, even when the Hessian is well-defined and positive, computing its inverse is expensive, cubic in n. This is more expensive than the gradient itself by a factor of n, which is prohibitive for large models.

Modifications that work

Several modifications make Newton's method tractable. Damped Newton replaces H with H + λI for some λ > 0, ensuring positive definiteness. Trust region methods restrict the step size to a region where the quadratic model is trusted, and adjust the region adaptively. Saddle-free Newton (Dauphin et al., 2014) uses the absolute values of the Hessian's eigenvalues, transforming saddle points into minima from the algorithm's perspective.

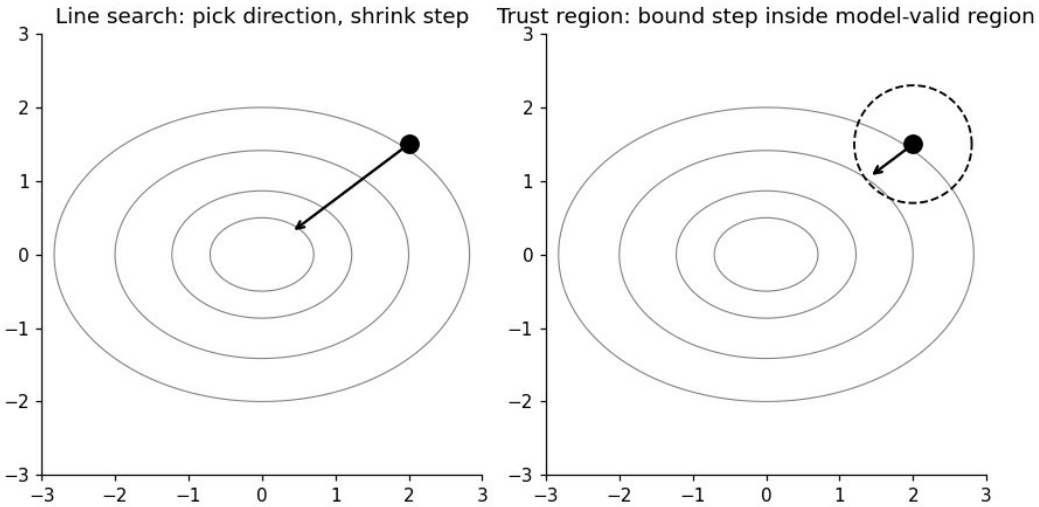


Figure 7.2: Line search vs trust region. Different ways to control step size.

None of these resolves the size problem for deep networks. For that, we need approximations to the Hessian itself.

7.2 Quasi-Newton Methods: BFGS and L-BFGS

Quasi-Newton methods build up an approximation to the inverse Hessian using only gradient information. They get most of the benefit of Newton's method without ever computing the actual Hessian. The most successful family is BFGS, named after its inventors Broyden, Fletcher, Goldfarb, and Shanno.

BFGS

BFGS maintains a matrix B_k that approximates the inverse Hessian. At each step, it updates B_k using the change in position and the change in gradient between iterations:

Let s_k = x_{k+1} - x_k and y_k = ∇f(x_{k+1}) - ∇f(x_k). Then B_{k+1} is updated by a rank-2 modification that ensures the secant condition: B_{k+1} y_k = s_k. This is the same condition the true inverse Hessian would satisfy in the linear case.

The mathematical details are technical, but the result is striking: BFGS often converges nearly as fast as Newton's method on convex problems, while requiring only gradient evaluations and O(n^2) storage. For decades it was the standard choice for medium-scale convex optimization.

L-BFGS

Even O(n^2) storage is prohibitive for large models. Limited-memory BFGS (L-BFGS) addresses this by storing only the last m updates (m typically between 5 and 20) rather than the full B matrix. The full inverse-Hessian approximation is reconstructed implicitly when applied to a vector.

L-BFGS has O(mn) memory and O(mn) per-step compute, comparable to a few gradient evaluations. It works well for smooth, convex, deterministic objectives. It is the standard optimizer for, say, fitting logistic regression on a fixed batch of data, or for solving the optimization problems inside SVM training.

Why L-BFGS is rarely used in deep learning

L-BFGS works less well in deep learning, for two main reasons. First, it relies on relatively accurate gradients, which we do not have when we use mini-batches. The line search and Hessian update both assume we are computing the same function repeatedly, which is not true for stochastic objectives. Second, the Hessian of a deep network changes rapidly during training, so the historical updates that L-BFGS uses become stale.

There is research on stochastic quasi-Newton methods, but for now, mainstream deep learning has settled on adaptive first-order methods like Adam.

7.3 Hessian Approximation Techniques

Even when we cannot afford the full Hessian, we can often afford useful approximations to it. These approximations underlie a range of practical methods that capture some of the benefit of curvature information.

Diagonal approximations

The simplest Hessian approximation is its diagonal. For some objectives, the off-diagonal terms are small relative to the diagonal, and the diagonal is enough to give well-scaled updates. Adam can be viewed as using a diagonal preconditioner, where each diagonal entry is the square root of the second-moment estimate of the gradient.

Hessian-vector products

We do not always need the Hessian itself; sometimes we only need products Hv for chosen vectors v. These can be computed in O(n) time using a single backward pass on the gradient, without ever forming H. Methods that use only Hv products are called Hessian-free; they include the Hessian-free optimization of Martens (2010) and the conjugate gradient subspace methods of Schraudolph and others.

Gauss-Newton and natural gradient

For least-squares and probabilistic models, the Gauss-Newton matrix and the Fisher information matrix are positive semidefinite approximations to the Hessian that capture much of the curvature information at lower cost. The natural gradient, using the inverse Fisher as preconditioner, is the basis for several modern methods, most notably K-FAC (Kronecker-Factored Approximate Curvature).

7.4 Trade-offs: Cost vs Convergence Speed

The choice of optimizer is fundamentally a trade-off between per-step cost and convergence rate. Each method has a different curve in this trade-off space.

Plain SGD has the cheapest steps but requires the most steps. Adam is slightly more expensive per step (constant factor) and typically converges in fewer iterations on deep learning problems. Quasi-Newton methods are 5 to 20 times more expensive per step but converge in dramatically fewer iterations on smooth, deterministic problems. Full Newton is even more expensive and reaches even higher convergence rates, but is only feasible for small models.

The deep learning calculus

In deep learning, per-step cost is dominated by the forward and backward passes through the network. Anything that requires a Hessian-vector product roughly doubles the per-step cost. This is rarely worth it unless the convergence rate improves by more than a factor of two.

This is why first-order methods dominate. The convergence-rate advantage of second-order methods is largely consumed by the per-step cost difference, especially at the scales where deep learning operates. The exceptions are small, smooth, well-conditioned problems, the regime where L-BFGS is still the right choice.

Key Takeaways

- Second-order methods promise faster convergence by using curvature, but pay for it with higher per-step cost.
- Newton's method is rarely usable directly on deep learning problems due to Hessian cost and indefiniteness.
- Quasi-Newton methods (BFGS, L-BFGS) provide a practical compromise for smooth convex problems.
- For deep learning, prefer diagonal approximations or Hessian-free methods over full second-order steps.

Exercises

- [Intermediate] Implement Newton's method on a simple convex function. Verify quadratic convergence near the optimum.
- [Intermediate] Construct a non-convex function with a saddle point. Show that plain Newton's method gets stuck and saddle-free Newton escapes.
- [Intermediate] Implement L-BFGS with memory size 10 and apply it to logistic regression. Compare convergence with SGD.
- [Basic] Compute a Hessian-vector product Hv using only backward passes. Verify it matches the explicit H times v.

Chapter 8: Constrained Optimization

So far we have considered only unconstrained optimization, where the parameters can take any value. In practice, machine learning is full of constraints: weights that must be non-negative, distributions that must sum to one, parameters confined to a sphere or a simplex. The mathematics of constrained optimization gives us the tools to handle these cases, and the language of Lagrangian duality is also the foundation of much of regularization theory.

8.1 The Setting

Constrained optimization replaces the unconstrained problem of minimizing $f(x)$ with the constrained problem of minimizing $f(x)$ subject to $g_i(x) \le 0$ for inequality constraints and $h_j(x) = 0$ for equality constraints. The set of points satisfying all constraints is the feasible set; the goal is the minimum of f over this set.

Examples in machine learning

- L1-constrained regression: minimize the squared error subject to the constraint that the sum of absolute values of weights is at most some t.
- Probability simplex: optimize over a probability distribution, where the values sum to one and are non-negative.
- Norm balls: weights confined to the unit ball, as in spectral normalization or robust optimization.
- Equality constraints: weight tying, parameter sharing, or constraints that some statistic equals a target value.

8.2 Lagrange Multipliers

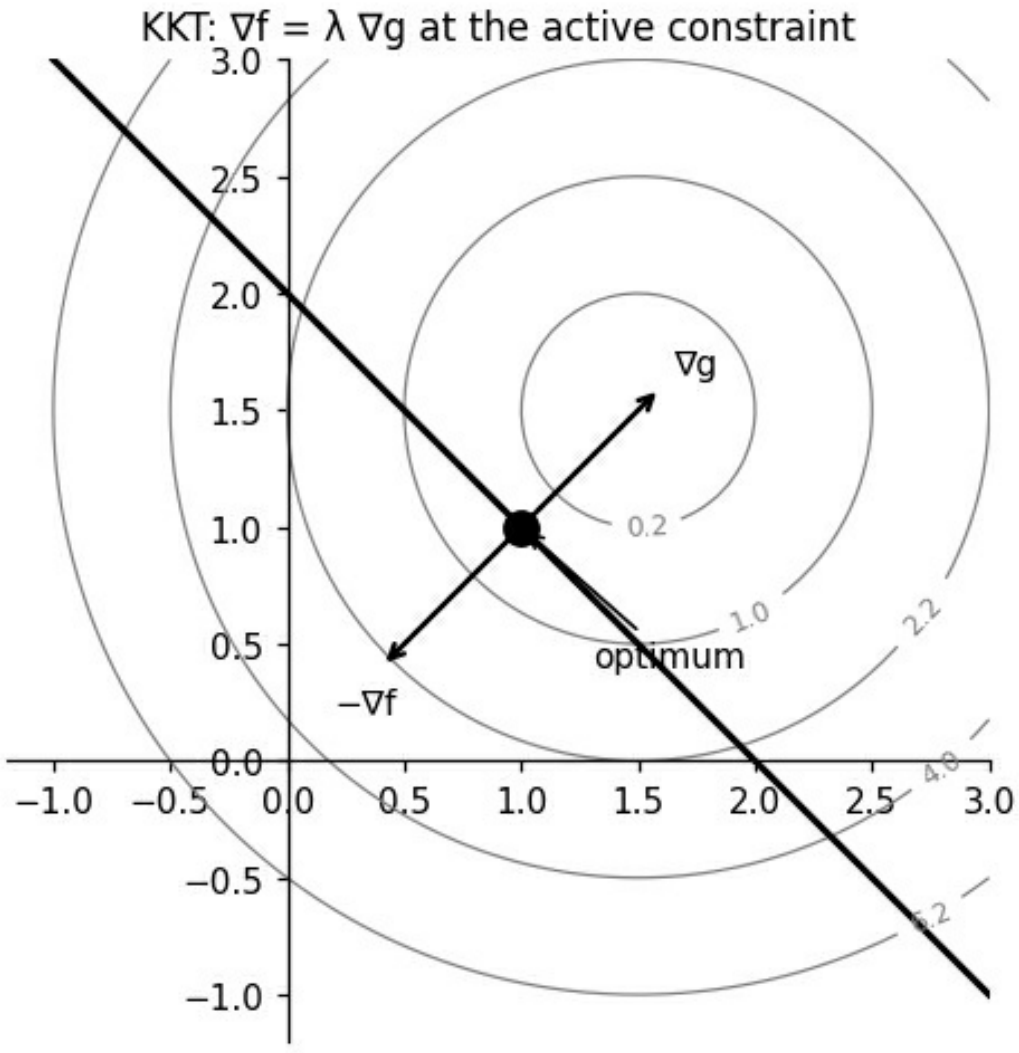


Figure 8.1: Karush-Kuhn-Tucker geometry. At a constrained optimum, the negative gradient of the objective is parallel to the gradient of the active constraint.

The classical method for handling equality constraints is Lagrange multipliers. To minimize f(x) subject to h(x) = 0, form the Lagrangian:

$$L(x, \lambda) = f(x) + \lambda\, h(x)$$

The condition for an optimum is that the gradient of L with respect to x is zero, and that h(x) is also zero. The multiplier λ measures how much the optimal value of f would change if we perturbed the constraint slightly. Multipliers have direct interpretations as shadow prices, sensitivities, or trade-off rates.

Geometric intuition

At a constrained optimum, the gradient of f and the gradient of h are parallel, there is no direction we can move that both decreases f and stays on the constraint surface. The Lagrange multiplier λ is the proportionality constant between them.

8.3 KKT Conditions

The Karush-Kuhn-Tucker (KKT) conditions generalize Lagrange multipliers to inequality constraints. They are the cornerstone of constrained optimization theory and appear throughout machine learning, particularly in support vector machines.

The conditions

For the constrained problem with inequalities $g_i(x) \le 0$ and equalities $h_j(x) = 0$, a point x^* is a candidate optimum if there exist multipliers $\lambda_i \ge 0$ and μ_j such that:

- Stationarity: $\nabla f(x^*) + \Sigma \lambda_i \nabla g_i(x^*) + \Sigma \mu_j \nabla h_j(x^*) = 0$
- Primal feasibility: $g_i(x^*) \le 0$ and $h_j(x^*) = 0$
- Dual feasibility: $\lambda_i \ge 0$
- Complementary slackness: $\lambda_i g_i(x^*) = 0$ for each i

The complementary slackness condition is particularly informative. It says that for each inequality constraint, either the constraint is active ($g_i = 0$) or its multiplier is zero. Active constraints determine the optimum; inactive ones do not affect it.

KKT in support vector machines

The dual formulation of the SVM is a direct application of KKT. The support vectors are the training examples whose Lagrange multipliers are nonzero, equivalently, the examples whose margin constraints are active. All other training examples could be removed without changing the solution. This sparsity property comes directly from complementary slackness.

8.4 Projected Gradient Descent

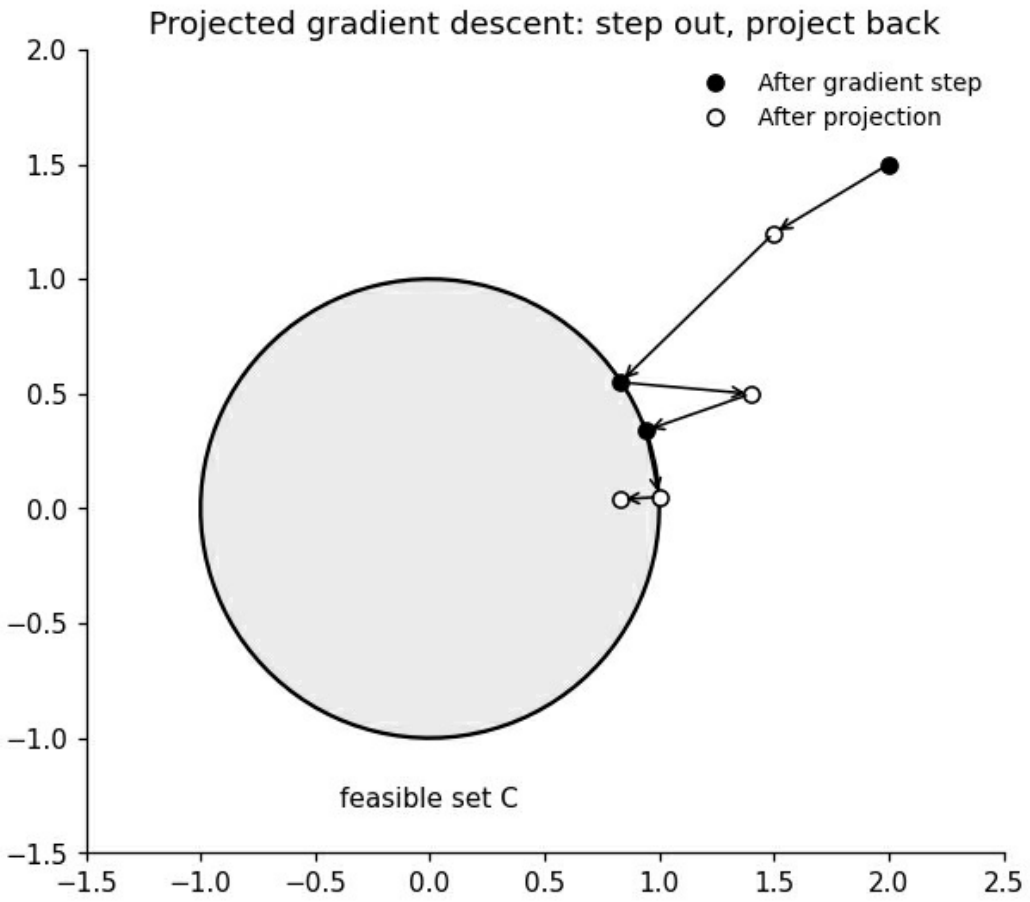


Figure 8.2: Projected gradient descent: take an unconstrained step, then project onto the feasible set.

The simplest practical algorithm for constrained optimization is projected gradient descent. It alternates two steps: take an unconstrained gradient step, then project the result back onto the feasible set.

$$x_{k+1} = \Pi_C(x_k - \eta \nabla f(x_k))$$

Here Π_C denotes projection onto the constraint set C. For simple constraint sets, the projection has a closed form. For the non-negative orthant, projection is just clipping at zero. For an L2-ball, projection is rescaling. For the simplex, projection is a slightly more involved but still cheap algorithm.

Why it works

Projected gradient descent inherits the convergence properties of unconstrained gradient descent on convex problems. For non-convex problems, it converges to a stationary point of the constrained problem, satisfying the KKT conditions. The per-step cost is the cost of a gradient step plus the cost of the projection, which is usually small.

When the projection is hard

Projected gradient descent works only when projection onto the constraint set is cheap. For some constraints, the cone of positive semidefinite matrices, intersections of multiple complicated sets, projection is itself an optimization problem. In those cases, we use other methods: penalty methods, interior point methods, or augmented Lagrangian methods.

8.5 Applications in Regularization

Many regularization techniques can be understood as constrained optimization in disguise. The duality between constraint-form and penalty-form is a deep and useful connection.

Constraint vs penalty

Consider L2-regularized regression. We can pose it two ways: minimize the squared error plus λ times the squared norm of the weights (penalty form), or minimize the squared error subject to the squared norm being at most t (constraint form). For appropriate values of λ and t, these problems have the same optimum. The Lagrange multiplier of the constrained problem is exactly λ in the penalty form.

Both formulations are useful. The penalty form is easier to optimize because it removes the constraint. The constraint form is easier to interpret, t has a clear meaning as a model complexity budget. In practice, we usually use the penalty form for efficiency and tune λ for the desired effect.

L1 regularization and sparsity

L1 regularization is a particularly important case. Penalizing the L1 norm of the weights (rather than the L2 norm) drives many weights exactly to zero, producing sparse solutions. This is interpretable as a constraint: the L1 ball has corners on the coordinate axes, and constraining solutions to lie in or on the L1 ball makes corner-solutions natural.

L1 regularization is used in feature selection (Lasso), in compressed sensing, and in pruning of neural networks. The optimization is somewhat trickier than L2, the L1 norm is non-differentiable at zero, but proximal gradient methods handle it cleanly.

Constrained optimization in modern ML

- Adversarial training: an inner constrained problem (find the worst perturbation in a small ball) inside an outer unconstrained problem (minimize the worst-case loss).
- Trust region policy optimization: a constraint on the KL divergence between successive policies stabilizes RL training.
- Wasserstein GANs: a Lipschitz constraint on the discriminator, often enforced via spectral normalization or gradient penalty.
- Fairness-constrained learning: parity or equalized-odds constraints on the predictor's behavior across groups.

Key Takeaways

- Constraints turn unconstrained optimization into a richer problem requiring Lagrange multipliers and KKT conditions.
- Projected gradient, penalty methods, and proximal methods are the practical workhorses for constrained problems.
- Lagrangian duality often reveals algorithmic structure invisible in the primal formulation.
- Much of regularization is constrained optimization in disguise — the L1/L2 penalties have constraint-set interpretations.

Exercises

- [Basic] Use Lagrange multipliers to find the point on the line $x + y = 1$ closest to the origin.
- [Intermediate] Write down the KKT conditions for the SVM dual and identify the support vectors.
- [Intermediate] Implement projected gradient descent for non-negative least squares. Test it on a small example.
- [Basic] Show that the projection of a vector onto the probability simplex can be computed in $O(n \log n)$ time. Implement it.
- [Intermediate] For L2-regularized regression, derive the relationship between the regularization strength λ and the constraint budget t.

Part III

Numerical Linear Algebra for Machine Learning

Linear algebra is the lingua franca of machine learning. Every dense layer, every convolution, every attention head, every batch operation reduces to matrix arithmetic. The next four chapters cover this terrain from the numerical perspective: solving linear systems stably, choosing the right factorization, computing eigenvalues at scale, and handling the size and structure of modern data.

Chapter 9: Solving Linear Systems

The fundamental task of numerical linear algebra is solving Ax = b: given a matrix A and a vector b, find the vector x. It sounds elementary, but it is the engine inside countless machine learning algorithms, least squares, Gaussian processes, Newton's method, Kalman filters, and more. Doing it well, at scale, with controlled precision, is the foundation everything else stands on.

9.1 Direct Methods: Gaussian Elimination and LU

Direct methods solve Ax = b in a fixed number of arithmetic operations, producing an exact answer in the absence of rounding error. The classical direct method is Gaussian elimination, which uses elementary row operations to reduce A to triangular form.

LU decomposition

Gaussian elimination's output, organized properly, is the LU decomposition: A = LU, where L is lower triangular with ones on the diagonal and U is upper triangular. With this decomposition in hand, solving Ax = b becomes two triangular solves: first Ly = b for y, then Ux = y for x. Each triangular solve is O(n^2), much cheaper than the O(n^3) cost of computing the decomposition itself.

The point of LU is that the cost of factoring A is paid once. After that, solving Ax = b for any number of right-hand sides b is cheap. This matters when we need to solve many systems with the same matrix, as in iterative refinement, sensitivity analysis, or repeated kernel evaluations.

Pivoting

Naive Gaussian elimination can fail catastrophically: if a diagonal element of A is zero or very small, dividing by it explodes the rounding error. The fix is pivoting, at each step, swap rows so that the divisor is as large as possible. This is partial pivoting, and it makes LU stable in practice.

Full pivoting (swapping both rows and columns) gives slightly better stability at higher cost. Most production libraries use partial pivoting because the stability benefit of full pivoting is rarely worth the extra time.

Cost and limits

LU is O(n^3) work and O(n^2) storage. For dense matrices up to a few thousand rows, this is fast and accurate, and LU is the default. For larger problems, the cubic cost dominates and we need iterative methods.

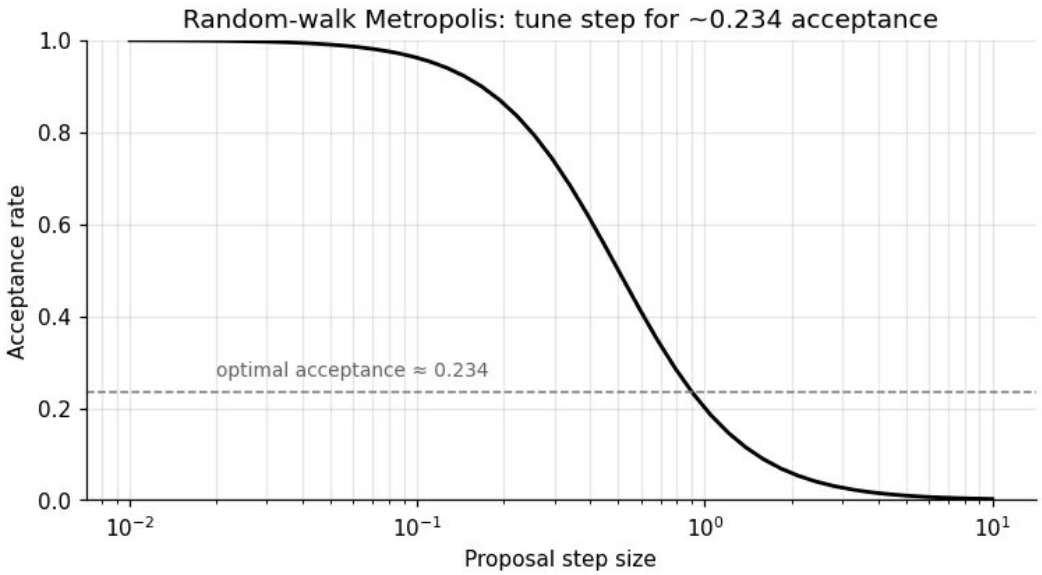


Figure 9.3: Direct vs iterative solvers. Cost crosses around n = 10⁴.

9.2 Iterative Methods: Jacobi and Gauss-Seidel

Iterative methods produce a sequence of approximations x_0, x_1, x_2, ... that converge to the solution. They never give an exact answer in finite work, but they can give an answer accurate to any specified tolerance, and for large sparse matrices they are dramatically faster than direct methods.

Jacobi iteration

Jacobi splits A into its diagonal D and off-diagonal R = A - D, and iterates:

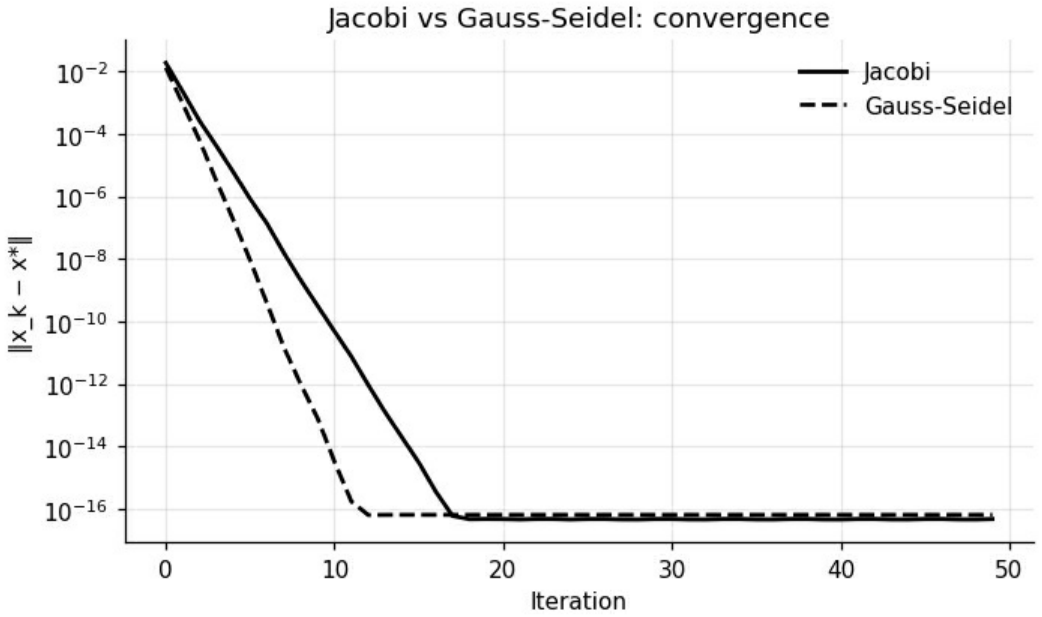


Figure 9.2: Gauss-Seidel typically converges faster than Jacobi because it uses already-updated values within the same iteration.

$$x_{k+1} = D^{-1}(b - R\,x_k)$$

In words: solve the diagonal system using the off-diagonal terms from the previous iterate. Jacobi converges if A is diagonally dominant, each diagonal element is larger than the sum of absolute values of off-diagonal elements in its row. It is simple and parallelizable but converges slowly.

Gauss-Seidel

Gauss-Seidel is similar to Jacobi but uses updated values as soon as they become available, rather than waiting for the next iteration. The update rule is:

$$x_i^{k+1} = (b_i - \Sigma_{j<i} a_{ij} x_j^{k+1} - \Sigma_{j>i} a_{ij} x_j^k) / a_{ii}$$

Gauss-Seidel converges faster than Jacobi for the same problems but is harder to parallelize because of the sequential dependence. Successive over-relaxation (SOR) is an extension that often converges much faster, by combining each Gauss-Seidel update with an extrapolation.

9.3 Conjugate Gradient

The conjugate gradient method (CG) is the workhorse iterative solver for symmetric positive definite systems. For problems where the matrix A has good structure, sparsity, clustered eigenvalues, or fast matrix-vector products, CG can solve linear systems with millions of unknowns in a small number of iterations.

How it works

CG generates a sequence of search directions that are mutually conjugate with respect to A: that is, p_i^T A p_j = 0 for i ≠ j. At each iteration, it minimizes the quadratic 1/2 x^T A x - b^T x along the new search direction, which gives one extra dimension of the answer. After n iterations, in exact arithmetic, the answer is exact.

In finite precision, CG often converges in far fewer than n iterations. The number of iterations needed for a given accuracy depends on the eigenvalue distribution of A: clustered eigenvalues give fast convergence; spread-out eigenvalues give slow convergence. The bound is roughly proportional to the square root of the condition number.

Why CG is essential for ML

CG is the engine inside many ML algorithms. Gaussian process regression at scale uses CG to solve kernel systems. Hessian-free optimization uses CG to compute Newton steps without forming the Hessian. Neural tangent kernel methods rely on CG. Anywhere we have a large symmetric positive definite system, CG is the algorithm of choice.

Preconditioning

CG is dramatically faster on a well-conditioned matrix than on an ill-conditioned one. Preconditioning replaces Ax = b with M^{-1}Ax = M^{-1}b for some matrix M chosen so that M^{-1}A has better-clustered eigenvalues. Common preconditioners include the diagonal of A (Jacobi preconditioning), an incomplete Cholesky factor, or, for ML, application-specific structure such as the Fisher information matrix.

```
# Conjugate gradient in scipy
from scipy.sparse.linalg import cg
x, info = cg(A, b, rtol=1e-6, maxiter=1000)
# info == 0 means convergence; positive means not converged
# negative means illegal input
```

9.4 Choosing the Right Solver

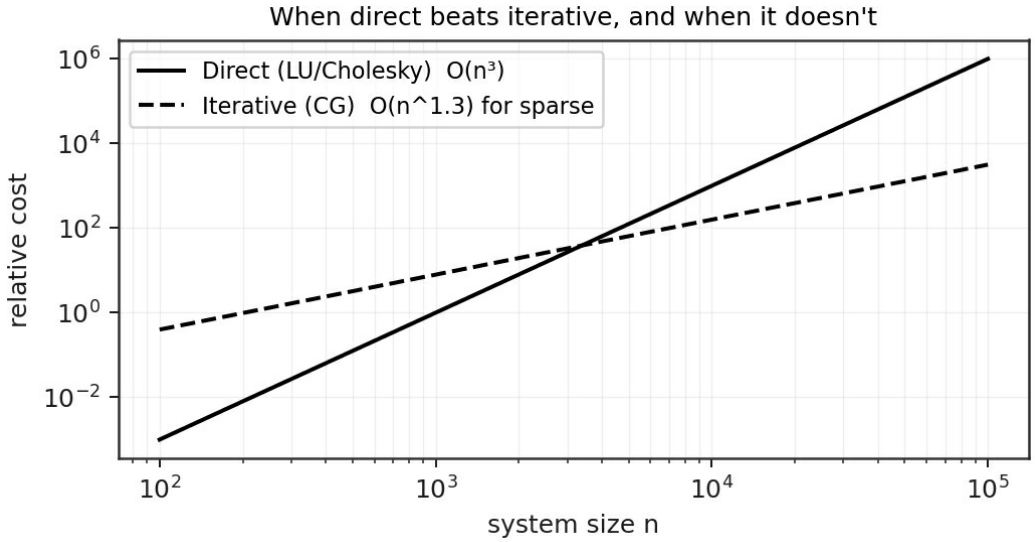


Figure 9.1: Decision flow for choosing a linear-system solver based on matrix size, sparsity, and structure.

Practical guidance for choosing between methods:

- Small dense matrices (n up to a few thousand): use LU with partial pivoting.
- Symmetric positive definite, dense: use Cholesky.
- Large, sparse, symmetric positive definite: use conjugate gradient with appropriate preconditioning.
- Large, sparse, general: use GMRES or BiCGSTAB.
- Least squares: solve via QR decomposition rather than the normal equations.
- Least squares with rank deficiency: solve via SVD with a tolerance.

Key Takeaways

• Solving Ax = b is the hidden engine inside many ML algorithms — never as visible as it should be.

• Direct methods (LU, Cholesky) are best for dense problems up to moderate size ($n \lesssim 10^4$).

• Iterative methods (especially conjugate gradient) handle large sparse problems where direct methods are infeasible.

• Preconditioning is almost always the difference between an iterative method that converges quickly and one that does not.

• Never explicitly form the inverse — solve the system instead.

Exercises

- [Intermediate] Implement LU decomposition with partial pivoting and verify on a 5×5 matrix.
- [Intermediate] Implement Jacobi and Gauss-Seidel iterations and compare convergence on a diagonally dominant matrix.
- [Intermediate] Use scipy's CG to solve a 1000×1000 sparse positive definite system.

Compare with a direct solve.

- [Intermediate] Construct an ill-conditioned positive definite matrix and apply CG with and without diagonal preconditioning. Compare iteration counts.

Chapter 10: Matrix Factorizations

Linear algebra in practice means matrix factorizations. We rarely compute with raw matrices; we compute with their factored forms, where the structure of the factorization makes specific operations cheap. The art of numerical linear algebra is choosing which factorization to use for which problem. This chapter covers the four factorizations every ML practitioner should know: LU, QR, Cholesky, and SVD.

10.1 LU Decomposition

We have already met LU: A = LU with L lower triangular and U upper triangular. LU is the factorization for solving general linear systems. With partial pivoting, the decomposition becomes PA = LU, where P is a permutation matrix that records the row swaps.

LU's defining feature is generality. It works for any invertible matrix, regardless of structure. Its weakness is that it does not exploit symmetry, if A is symmetric, LU computes both factors when one would suffice. For symmetric matrices, prefer Cholesky.

10.2 QR Decomposition

QR decomposition factors a matrix A as A = QR, where Q is orthogonal and R is upper triangular. Unlike LU, QR exists for any matrix A, not just square, not just invertible. It is the canonical tool for least squares, orthogonalization, and many eigenvalue computations.

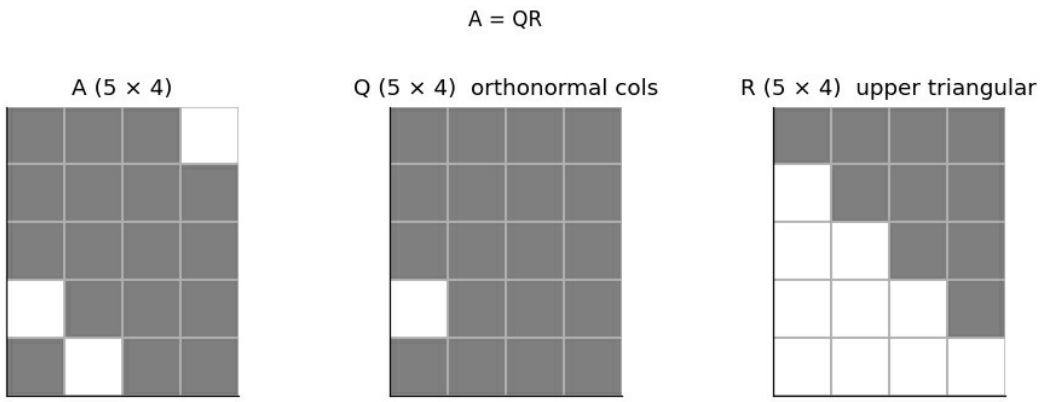


Figure 10.1: QR factorization. Q has orthonormal columns; R is upper triangular. Same sparsity structure but different conditioning properties.

Three ways to compute QR

There are three classical algorithms for QR decomposition, each with different stability properties.

- Gram-Schmidt: orthogonalize the columns of A one at a time. Simple but numerically unstable in finite precision.
- Modified Gram-Schmidt: a small change in the order of operations that improves stability substantially. Often used in Krylov subspace methods.
- Householder reflections: use a sequence of orthogonal reflections to zero out subdiagonal elements. The most stable and the standard choice for general QR.
- Givens rotations: use a sequence of plane rotations. More expensive than Householder

for full QR but useful for updating an existing factorization or for sparse matrices.

QR for least squares

To solve the least-squares problem min ||Ax - b||, factor A = QR and solve Rx = Q^T b. This is more accurate than the normal equations approach (solving A^T A x = A^T b) because it avoids forming A^T A, which can square the condition number.

10.3 Cholesky Decomposition

For a symmetric positive definite matrix A, Cholesky decomposition factors it as A = LL^T, where L is lower triangular. Cholesky is roughly twice as efficient as general LU and is more numerically stable, it cannot fail catastrophically the way LU without pivoting can.

Where it appears in ML

- Gaussian process regression: covariance matrices are positive definite, and Cholesky is the standard solver.
- Multivariate Gaussian sampling: factor the covariance, multiply by standard normal samples.
- Quadratic programming: positive definite quadratic terms factored once for repeated solves.
- Whitening transformations: dividing by the Cholesky factor of the covariance produces uncorrelated unit-variance features.

Cholesky as a positive-definiteness test

Cholesky decomposition succeeds if and only if A is positive definite. Many implementations of "is this matrix positive definite" simply attempt Cholesky and check whether it raises an error. This is faster than computing eigenvalues and gives the answer with the same arithmetic as solving.

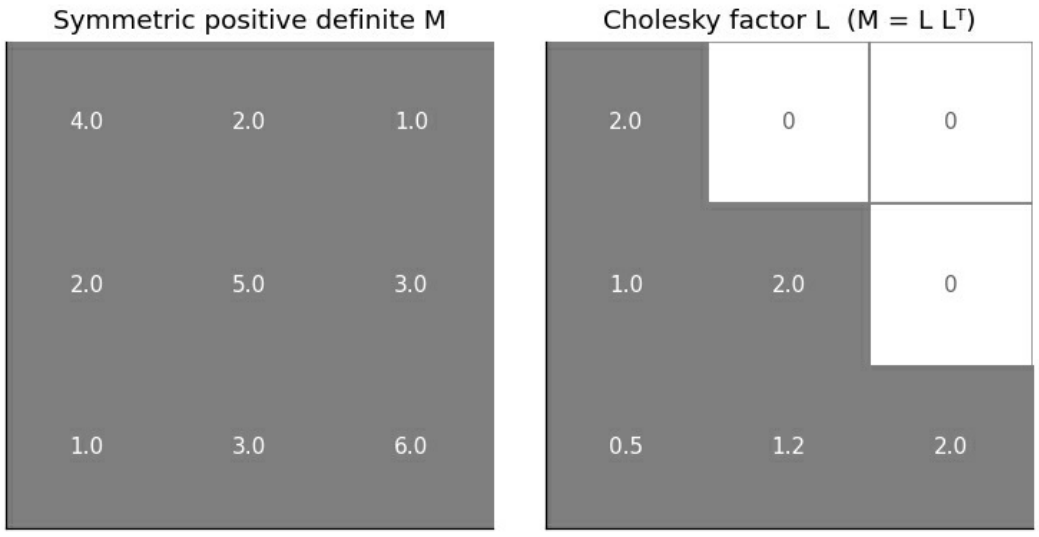


Figure 10.2: Cholesky factorization of a symmetric positive definite matrix. The factor L is lower triangular with positive diagonal.

10.4 SVD: The Most Powerful Factorization

Singular value decomposition (SVD) is the most general and most useful matrix factorization. For any matrix A (rectangular, rank-deficient, anything), SVD gives A = UΣV^T where U and V are orthogonal and Σ is diagonal with non-negative entries called singular values.

The singular values are arranged in decreasing order. The number of nonzero singular values is the rank of the matrix. The largest singular value is the operator 2-norm; the ratio of largest to smallest nonzero is the condition number.

What SVD tells you

SVD lets us read off the most important properties of a matrix immediately. The rank is the count of nonzero singular values. The numerical rank, given a tolerance, is the count of singular values exceeding the tolerance. The condition number is the ratio of largest to smallest. The best low-rank approximation in the 2-norm or Frobenius norm is given by truncating the SVD.

Cost

SVD is the most expensive of the standard factorizations: about 21 n^3 / 2 flops for an n by n matrix using the standard algorithm. For large matrices, this is prohibitive, and we use truncated or randomized variants. We will see these in chapter 11.

10.5 Stability Considerations

Different factorizations have different stability properties, and the choice can matter enormously when the problem is ill-conditioned.

LU stability

LU with partial pivoting is stable for almost all matrices encountered in practice. There exist contrived examples where it fails (the Wilkinson matrix), but they essentially never arise in real applications.

QR stability

QR via Householder reflections is unconditionally stable. The QR factor R has the same condition number as A, so QR-based solves do not amplify error. This is why QR is the gold standard for least squares.

Cholesky stability

Cholesky is stable for positive definite matrices. For matrices that are mathematically positive definite but numerically near-singular, the algorithm may fail at the last step when the diagonal entry of L would need to be the square root of a small negative number due to rounding. The fix is regularization: add a small multiple of the identity before factoring.

SVD stability

SVD is among the most stable algorithms in linear algebra. Modern implementations (the Golub-Reinsch algorithm and variants) are accurate to within a small multiple of machine epsilon. SVD is the algorithm of choice when accuracy matters more than speed.

10.6 When to Use Which Method

The decision tree in practice:

- Solving Ax = b for general invertible A: LU.
- Solving Ax = b for symmetric positive definite A: Cholesky.
- Solving least squares with full-rank A: QR.
- Solving least squares with rank-deficient A or determining numerical rank: SVD.
- Computing the spectral norm or condition number: SVD.
- Low-rank approximation, dimensionality reduction: SVD.
- Eigenvalues of a symmetric matrix: symmetric eigendecomposition (related to but different from SVD).

Key Takeaways

- Factorize once, operate on the factors many times — this is the central efficiency trick of numerical linear algebra.
- LU for general systems, Cholesky for positive definite, QR for least squares, SVD for rank and condition.
- Cost varies by an order of magnitude across factorizations, so choose deliberately.
- Cholesky is twice as fast as LU on PD matrices — use it when you can.
- SVD is the most expensive but most informative factorization; reach for it when you need rank, condition, or low-rank structure.

Exercises

- [Intermediate] Implement modified Gram-Schmidt for QR. Compare its accuracy with classical Gram-Schmidt on a near-rank-deficient matrix.
- [Basic] Compute the Cholesky factor of a 5x5 random positive definite matrix. Verify L L^T equals the original matrix.
- [Basic] Compute the SVD of a 100x50 random matrix. Verify the orthogonality of U and V.
- [Intermediate] Solve a least-squares problem three ways: normal equations, QR, SVD. Compare accuracy on a problem with high condition number.

Chapter 11: Eigenvalue Problems and PCA

Eigenvalues capture the directions in which a matrix acts as pure scaling. They appear throughout machine learning: in PCA, in spectral methods, in stability analysis, in graph algorithms. This chapter covers the practical computation of eigenvalues and eigenvectors, the singular value decomposition that generalizes it to non-square matrices, and the application that brings these techniques into mainstream ML, principal component analysis.

11.1 Power Iteration

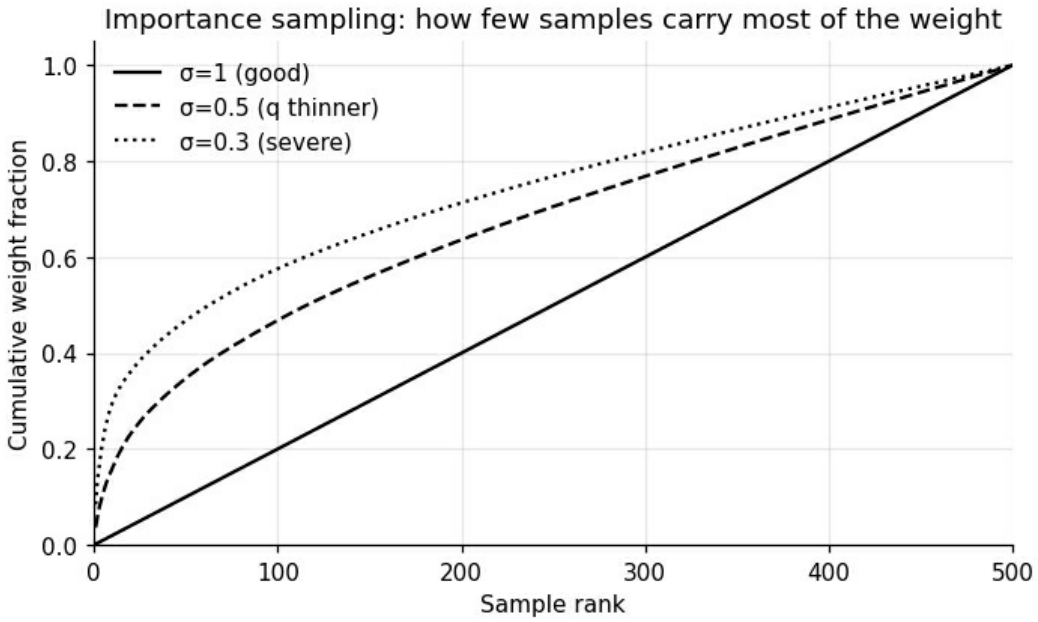


Figure 11.2: Power iteration convergence. Error decreases geometrically at rate $|\lambda_2/\lambda_1|$.

The simplest eigenvalue algorithm is power iteration. To find the dominant eigenvalue and eigenvector, repeatedly multiply a vector by the matrix and normalize:

$$v_{k+1} = A\, v_k / ||A\, v_k||$$

After many iterations, v_k aligns with the eigenvector corresponding to the largest-magnitude eigenvalue. The eigenvalue itself can be recovered as the Rayleigh quotient v^T A v.

Convergence rate

Power iteration converges at rate |λ_2| / |λ_1|, the ratio of the second-largest to largest eigenvalue magnitudes. If the eigenvalues are well-separated, convergence is fast. If they are close, λ_1 ≈ λ_2, convergence is glacial.

The pathological case is when |λ_1| = |λ_2| but they are different (perhaps λ_1 = 1 and λ_2 = -1). Power iteration oscillates without converging. The fix is to apply power iteration to a shifted matrix A - σI or to A^2.

Why power iteration matters

Power iteration is the conceptual foundation of more sophisticated methods. The Lanczos and Arnoldi algorithms generalize it to compute multiple eigenvalues. PageRank is essentially power iteration on the web graph's transition matrix. Spectral clustering uses power iteration to find the

leading eigenvectors of a graph Laplacian.

11.2 The QR Algorithm

For computing the full eigendecomposition of a small to medium dense matrix, the QR algorithm is the standard. Despite the name overlap, it is different from QR decomposition, it uses QR decomposition repeatedly, but the algorithm itself iterates.

The basic idea: at each step, compute the QR decomposition of the current matrix, then form a new matrix by reversing the multiplication: A_new = R Q. This iteration, with appropriate refinements (shifts, deflation, Hessenberg reduction), converges to a triangular matrix whose diagonal entries are the eigenvalues.

The QR algorithm in its modern form is one of the most important algorithms of the twentieth century. It is the foundation of LAPACK's eigenvalue routines and is used inside virtually every linear algebra library.

11.3 Singular Value Decomposition

The SVD of A factors it as A = UΣV^T. The columns of V are right singular vectors, the columns of U are left singular vectors, and the diagonal of Σ contains the singular values. SVD generalizes the eigendecomposition to non-square matrices and gives the cleanest possible picture of what a matrix does.

Connection to eigenvalues

The singular values of A are the square roots of the eigenvalues of A^T A (or AA^T, which has the same nonzero eigenvalues). The right singular vectors are eigenvectors of A^T A; the left singular vectors are eigenvectors of AA^T. So SVD is, in principle, an eigenvalue computation in disguise.

In practice, computing SVD by forming A^T A is a bad idea. Squaring A doubles the condition number, small singular values become very small and may underflow. Modern SVD algorithms work directly on A using techniques like bidiagonalization and the Golub-Kahan algorithm, never forming A^T A.

11.4 PCA from a Numerical Perspective

Principal component analysis is the most widely used SVD application in machine learning. Given a data matrix X with samples in rows and features in columns, PCA finds the directions of maximum variance, the principal components, and projects the data onto a small number of them for dimensionality reduction or visualization.

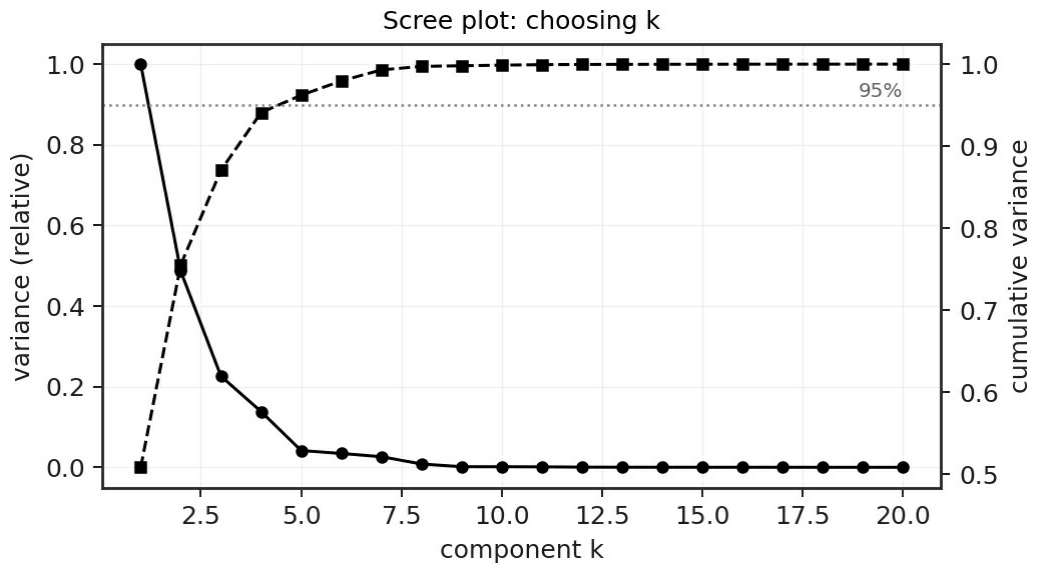


Figure 11.1: A scree plot showing how variance is distributed across principal components.

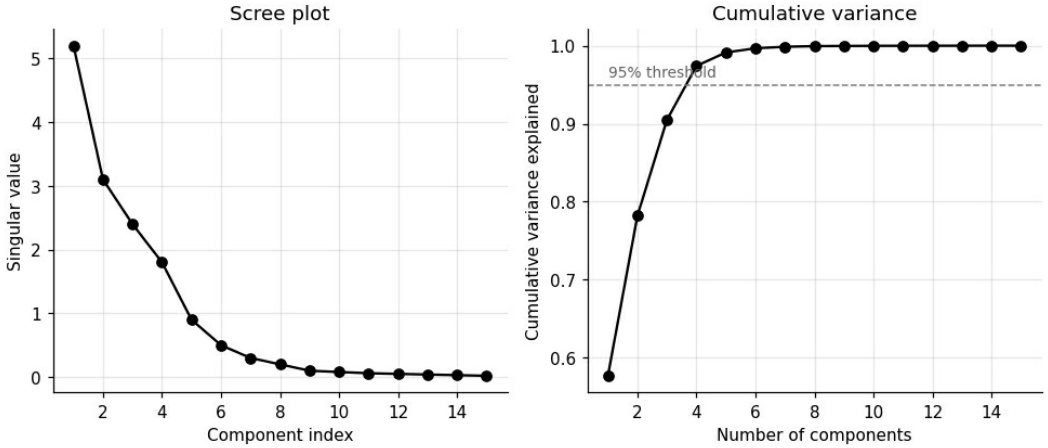


Figure 11.3: Scree plot (left) and cumulative variance (right).

The recipe

- Center the data: subtract the mean of each feature.
- Compute the SVD: X_centered = UΣV^T.
- The columns of V are the principal directions; the columns of UΣ are the projections of the data onto them.
- To reduce to k dimensions, keep only the first k columns of V and the first k columns of UΣ.

Numerical pitfalls

The naive approach to PCA, compute the covariance matrix as X^T X / (n-1), then find its eigenvectors, has the same problem as the naive approach to SVD. Forming X^T X squares the condition number and amplifies rounding error. For data with disparate feature scales or near-collinear features, the SVD-of-X approach is much more accurate.

Another pitfall: PCA is sensitive to feature scaling. Features with large variance will dominate

the principal components, regardless of their importance. Standard practice is to standardize features (subtract mean, divide by standard deviation) before PCA, so each feature contributes on equal footing. Whether to standardize or not is a modeling choice that depends on the problem.

```
import numpy as np

def pca(X, n_components):
    X_centered = X - X.mean(axis=0)
    U, S, Vt = np.linalg.svd(X_centered, full_matrices=False)
    components = Vt[:n_components]
    projected = U[:, :n_components] * S[:n_components]
    return projected, components, S
```

11.5 Dimensionality Reduction at Scale

Standard PCA scales poorly. Computing the SVD of a matrix with d features and n samples costs roughly O(min(n^2 d, n d^2)). For modern problems with millions of samples and thousands of features, this is prohibitive. Several techniques address the scaling problem.

Truncated SVD

If we only need the top k singular vectors, we do not need the full SVD. Iterative methods like ARPACK or the LOBPCG algorithm can compute the top k pairs in O(nd k) per iteration, with the number of iterations modest if the singular values are well-separated.

Randomized SVD

Randomized SVD (Halko, Martinsson, and Tropp, 2011) is the modern method of choice for very large matrices. The idea is to project the matrix onto a random low-dimensional subspace, compute the SVD of the much smaller projected matrix, and use it to approximate the SVD of the original. With appropriate oversampling, the approximation is very good in practice.

```
from sklearn.utils.extmath import randomized_svd
U, S, Vt = randomized_svd(X, n_components=50, n_iter=5)
```

Incremental PCA

When data does not fit in memory, incremental PCA processes it in chunks, updating the principal directions as each chunk arrives. Several algorithms exist; the practical ones in scikit-learn handle this gracefully for streaming data.

PCA via Hebbian learning

Oja's rule and its descendants compute principal components via online learning, one sample at a time. These methods are slower per sample than batch SVD but use constant memory and can adapt to non-stationary data.

Key Takeaways

- Eigenvalues and singular values reveal the inner structure of matrices.
- Power iteration finds the dominant eigenvector; the QR algorithm finds all eigenvalues.
- PCA is SVD applied to centered data — the most widely used dimensionality reduction technique.
- Truncated and randomized SVD make PCA scale to modern data sizes.
- Always plot the scree (singular-value spectrum) before choosing the number of components.

Exercises

- [Intermediate] Implement power iteration. Apply it to a 10x10 random symmetric matrix and verify convergence to the dominant eigenvalue.
- [Basic] Compute the SVD of a centered MNIST data matrix (28x28 images flattened to 784 features). Visualize the first few principal directions as images.
- [Intermediate] Compare full SVD with randomized SVD for a 10000x500 random matrix. Verify the approximation quality.
- [Intermediate] Implement PCA two ways: via SVD and via eigendecomposition of the covariance matrix. Compare accuracy on data with widely disparate feature scales.

Chapter 12: Large-Scale Linear Algebra

The matrices in modern machine learning have grown enormous. A transformer's attention matrix at sequence length 8192 has 67 million entries per head per layer. The Hessian of a billion-parameter model has 10^18 entries. The covariance matrix of a single layer's activations can occupy dozens of gigabytes. At these scales, the algorithms we have studied need to be reformulated to exploit structure, sparsity, and parallelism, or they simply do not work.

12.1 Sparse Matrices

A matrix is sparse if most of its entries are zero. Sparsity is everywhere in modern data: word-document matrices, graph adjacency matrices, recommendation systems, certain kinds of attention masks. Storing and operating on sparse matrices efficiently requires representations that record only the nonzero entries.

Sparse storage formats

- COO (coordinate format): a list of (row, column, value) triples. Easy to build but inefficient for arithmetic.
- CSR (compressed sparse row): row pointers plus column indices and values. Efficient for matrix-vector multiplication and row operations.
- CSC (compressed sparse column): the column-oriented analog of CSR.
- Block sparse formats: chunks of contiguous nonzeros, useful for hardware-accelerated computation.

Sparse matrix-vector multiplication

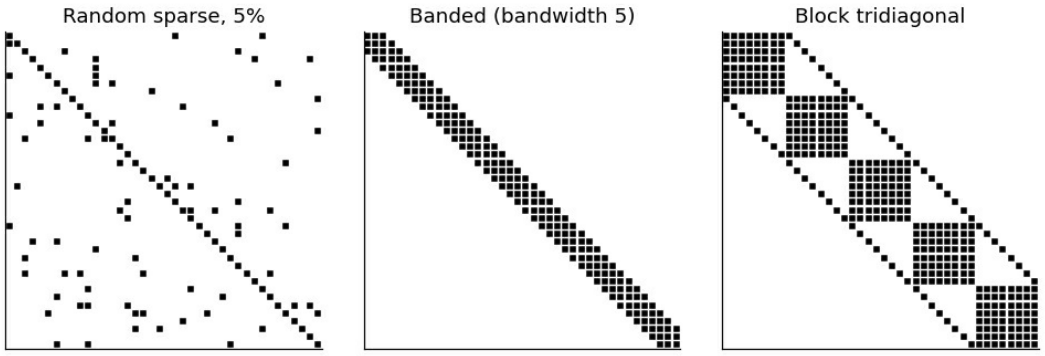


Figure 12.1: Sparse matrix patterns. Random sparse (left), banded (middle), block tridiagonal (right). Storage and algorithm choice depend on the pattern.

The fundamental operation on sparse matrices is matrix-vector multiplication. For a sparse matrix with nnz nonzeros, this costs O(nnz) work, independent of the matrix's size. For a graph adjacency matrix with n nodes and m edges, where m is much less than n^2, this is the difference between O(m) and O(n^2) per multiplication.

Sparse matrix-matrix multiplication is harder. The result of multiplying two sparse matrices may have many more nonzeros than either factor, the so-called fill-in problem. Specialized algorithms minimize fill-in, but in general, sparse-times-sparse is much trickier than sparse-times-dense.

12.2 Memory-Efficient Computation

Beyond sparsity, large-scale linear algebra requires algorithms that respect the memory hierarchy. Modern hardware has fast caches near the processor and slow main memory; an algorithm that fits in cache runs orders of magnitude faster than one that does not.

Blocked algorithms

The dominant technique is blocking. Instead of computing one element at a time, we divide matrices into blocks small enough to fit in cache and operate on them in cache-resident chunks. Blocked LU, blocked Cholesky, blocked QR, every standard algorithm has a blocked version. The blocked versions are not just faster; they may be 100 times faster on the same hardware.

In-place computation

Where possible, modern algorithms overwrite their input rather than allocating new storage. LU decomposition writes L and U into the storage of A, separated by the implicit unit diagonal. In-place computation halves memory usage and improves cache behavior, at the cost of destroying the original matrix.

Streaming computation

For very large data, streaming algorithms process the data in passes, never holding all of it in memory. Online PCA, incremental matrix factorization, and streaming least squares all fall into this category. The trade-off is between memory (small) and the number of passes over the data (potentially many).

12.3 Iterative Solvers for Big Data

For truly large problems, even sparse direct methods are infeasible. Sparse Cholesky on a million-by-million system can fill in to dozens of times the original storage. The only way to solve such problems is iteratively, with conjugate gradient or its relatives.

Krylov subspace methods

CG, GMRES, BiCGSTAB, and their relatives are all Krylov subspace methods: they construct a sequence of approximations from the subspace spanned by b, Ab, A^2 b, ... The genius of these methods is that they only require matrix-vector products with A, not the matrix itself. So if there is a fast way to compute Av without storing A explicitly, the methods work.

This matters in machine learning. The Hessian of a neural network can be applied to a vector in the time of two backward passes, even though storing the Hessian is impossible. Hessian-free optimization exploits exactly this fact.

Preconditioning at scale

Preconditioning is even more important at scale than at smaller sizes, because the iteration count of an unpreconditioned solver may be larger than feasible. Multigrid preconditioners, incomplete factorizations, and domain decomposition are the classical workhorses. For ML, application-specific preconditioners, such as a Kronecker-factored approximation to the Fisher information, can be dramatically more effective than general-purpose ones.

12.4 GPU Considerations

The hardware story for linear algebra has been transformed by GPUs. A modern GPU can perform tens of trillions of floating-point operations per second on dense matrix multiplications,

an order of magnitude more than even the best CPUs. The challenge is structuring algorithms to use that throughput.

What GPUs love

- Dense matrix multiplications. The GEMM (general matrix multiply) operation is GPU-optimal.
- Operations that can be parallelized across thousands of threads.
- Workloads with regular memory access patterns.
- Reduced precision: float16 and bfloat16 give 2-4x throughput improvements over float32 on modern hardware.

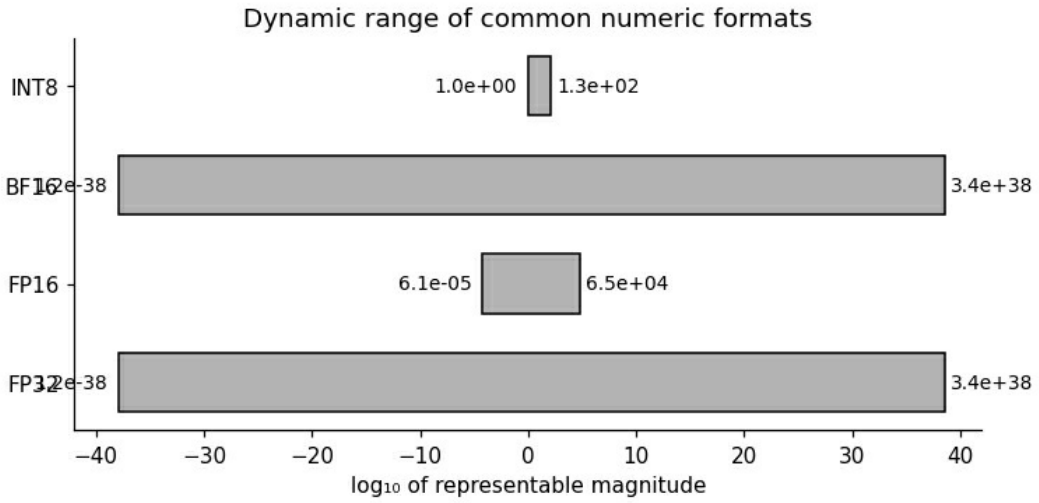


Figure 12.2: Dynamic range of common numeric formats. BF16 keeps FP32's range but loses precision; FP16 loses both. INT8 has a narrow range and requires careful calibration.

What GPUs hate

- Sparse operations with irregular structure.
- Algorithms with sequential dependencies that limit parallelism.
- Frequent host-to-device data transfers.
- Workloads that don't fill the GPU's parallel capacity.

Mixed-precision training

Modern deep learning routinely uses mixed precision: forward passes in float16 or bfloat16 for speed, with a master copy of weights in float32 for accuracy. The arithmetic of mixed precision is delicate. Loss scaling, multiplying the loss by a large constant to bring small gradient values into the float16 range, is a standard technique. Bfloat16, with the same exponent range as float32 but fewer mantissa bits, has emerged as a particularly good compromise: it does not need loss scaling, and it converges as well as float32 on most problems.

```
# PyTorch automatic mixed precision
from torch.cuda.amp import autocast, GradScaler
scaler = GradScaler()
for x, y in loader:
   with autocast():
      loss = model(x, y)
```

```
scaler.scale(loss).backward()
scaler.step(optimizer)
scaler.update()
optimizer.zero_grad()
```

Distributed linear algebra

When even a single GPU is not enough, we distribute matrices across multiple devices. Data parallelism splits the batch; model parallelism splits the parameters. Tensor parallelism splits individual matrices across devices and uses specialized communication patterns to compute matrix multiplications collectively. The implementation details are intricate but the underlying linear algebra is straightforward.

Key Takeaways

- At scale, algorithms must respect the memory hierarchy, exploit sparsity, and run on parallel hardware.
- Iterative methods needing only matrix-vector products are often the only feasible approach.
- Mixed-precision computation is now standard; bfloat16 is the emerging default for deep learning.
- Communication cost dominates compute cost at multi-node scale — design algorithms with this in mind.
- Knowing what your hardware loves and hates is the difference between a model that trains in days vs months.

Exercises

- [Intermediate] Construct a 1000x1000 sparse matrix with about 1% nonzeros. Compare CSR and dense storage size and the cost of matrix-vector multiplication.
- [Intermediate] Implement blocked matrix multiplication with various block sizes. Find the optimal block size for your CPU's cache.
- [Intermediate] Run the same neural network in float32 and bfloat16. Compare training time and final accuracy.
- [Research-Level] Read the FlashAttention paper. Explain how it exploits the memory hierarchy of GPUs.

Part IV

Numerical Methods in Deep Learning

Deep learning is where numerical issues are most acute and where their resolution is most consequential. The next four chapters apply the foundational ideas, gradient computation, conditioning, optimization, regularization, to the specific challenges of training deep networks.

Chapter 13: Backpropagation as a Numerical Algorithm

Backpropagation is often presented as a clever application of the chain rule, and it is, but it is also a numerical algorithm running in finite precision, and its numerical behavior matters as much as its mathematical correctness. This chapter examines backpropagation from the perspective of the computer that actually performs it.

13.1 The Chain Rule as a Computation Graph

To compute a gradient, we trace through the operations that produced the output, from end to beginning, accumulating partial derivatives along the way. The structure of this trace is a directed acyclic graph, the computation graph, with operations as nodes and intermediate values flowing along edges.

Forward and backward

The forward pass evaluates the function: each node computes its output from its inputs. The backward pass computes gradients: starting from the output and working backward, each node receives a gradient with respect to its output and produces gradients with respect to its inputs.

The chain rule says that if y = g(x) and z = f(y), then dz/dx = (dz/dy)(dy/dx). For each operation in the graph, we know the local Jacobian (dy/dx); the backward pass propagates the upstream gradient (dz/dy) and produces the downstream gradient (dz/dx) by chain-rule multiplication.

Static vs dynamic graphs

Some frameworks (TensorFlow 1.x, JAX with jit) build the graph once and reuse it; others (PyTorch, TensorFlow eager) build it dynamically as code runs. The dynamic approach is more flexible, control flow, recursion, and data-dependent shapes are natural, at some cost in performance. Modern frameworks have largely converged on dynamic graphs with optional compilation for performance-critical code.

13.2 Reverse-Mode Differentiation

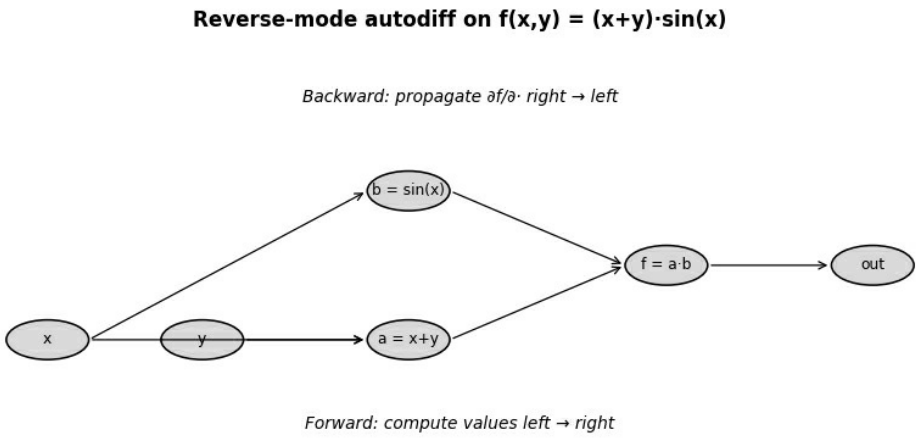


Figure 13.2: Reverse-mode autodiff. Forward computes values; backward propagates gradients.

Backpropagation is a special case of reverse-mode automatic differentiation. The key insight is that, for a function from R^n to R, reverse-mode computes the gradient at the cost of one forward pass and one backward pass, independent of n. This is what makes training a billion-parameter network feasible.

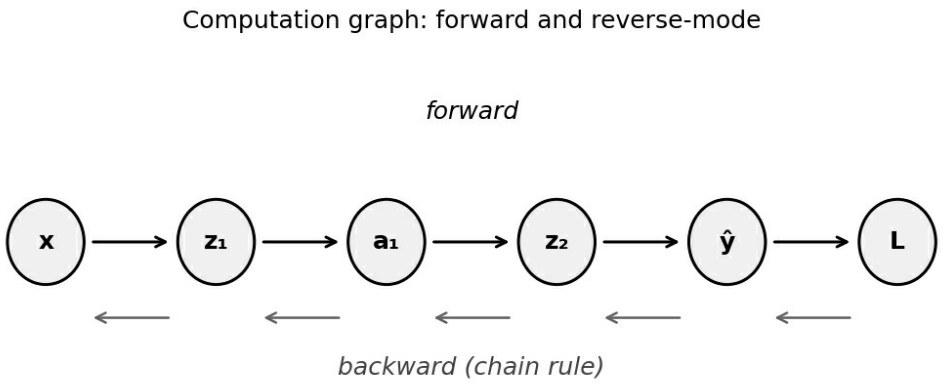


Figure 13.1: A simple computation graph. Forward pass computes node values; backward pass propagates gradients in reverse topological order.

Why reverse mode wins for ML

Reverse-mode is the right choice when the input dimension is much larger than the output dimension. In neural networks, the input is the parameter vector (millions or billions of dimensions) and the output is the scalar loss. Reverse-mode pays for itself many times over.

Forward-mode automatic differentiation is the dual: efficient when the input has fewer dimensions than the output. It is rarely the right choice for ML, but it appears in some specialized contexts, computing Hessian-vector products, for instance, can use a combination of forward and reverse modes.

Memory: the price of reverse mode

Reverse-mode requires storing intermediate activations from the forward pass, since they are needed in the backward pass. For a deep network, this means memory usage proportional to depth times the activation size. For very deep networks or very long sequences, this becomes prohibitive.

Gradient checkpointing, recomputing some activations during the backward pass instead of storing them, trades compute for memory. Set the checkpoints carefully and the memory savings can be enormous, at the cost of one or two extra forward passes.

13.3 Numerical Instability in Backpropagation

Backpropagation is mathematically a chain of multiplications. Numerically, this is precisely where things go wrong. The product of many factors can either explode or vanish, and either failure produces useless gradients.

The matrix-product picture

In a deep feedforward network, the gradient of the loss with respect to layer i's input is a product of Jacobians of all subsequent layers, multiplied by the gradient of the loss with respect to the

output. If those Jacobians have spectral norm above one, the product grows exponentially with depth (gradient explosion). If they have spectral norm below one, it shrinks exponentially (gradient vanishing).

Both modes appear in practice. Gradient explosion classically afflicted RNNs with linear activations. Gradient vanishing afflicted deep networks with sigmoid or tanh activations. The remedies, ReLU activations, careful initialization, batch normalization, residual connections, all aim to keep the layer Jacobians close to having spectral norm one.

Gradient quantization

Modern deep learning often uses lower-precision arithmetic, float16 or bfloat16, for speed. Gradients in low precision can underflow to zero or overflow to infinity. The fix, loss scaling, multiplies the loss by a large constant before backprop, scales gradients up, performs the backward pass in low precision, then scales gradients back down before the optimizer step.

Numerical gradient checking

To verify a gradient implementation is correct, compare it to a numerical approximation:

$$\partial f/\partial x \approx (f(x + h) - f(x - h)) / (2h)$$

With h around the square root of machine epsilon (about 10^-8 for float64), the central difference is accurate to several digits. Disagreement with the analytical gradient by more than a few digits indicates a bug. This is one of the most useful debugging techniques in all of deep learning.

```
def gradient_check(f, x, h=1e-7):
    grad_analytic = analytic_gradient(f, x)
    grad_numeric = np.zeros_like(x)
    for i in range(len(x)):
        e = np.zeros_like(x); e[i] = h
        grad_numeric[i] = (f(x + e) - f(x - e)) / (2 * h)
    rel_err = np.abs(grad_analytic - grad_numeric) / (
        np.abs(grad_analytic) + np.abs(grad_numeric) + 1e-12)
    return rel_err.max()
```

Key Takeaways

- Backpropagation is mathematically the chain rule, but numerically it is a chain of matrix products that can explode, vanish, or be inaccurate.
- Reverse-mode autodiff is what makes computing gradients of a scalar loss with respect to many parameters efficient.
- Modern deep learning interventions (initialization, normalization, residual connections) are largely about keeping the chain numerically well-behaved.
- Gradient checking against finite differences is the cheapest defense against autograd bugs in custom layers.
- Memory cost of autograd is proportional to the activation footprint — gradient checkpointing trades compute for memory.

Exercises

- [Intermediate] Implement reverse-mode autodiff for scalar functions on a small computation graph by hand.
- [Intermediate] Implement gradient checking and verify a hand-coded gradient against PyTorch's autograd.

- [Intermediate] Train a 50-layer MLP with ReLU and verify gradient magnitudes at each layer. Replace ReLU with sigmoid and observe the change.
- [Intermediate] Use gradient checkpointing in PyTorch on a large model. Compare memory usage with and without.

Chapter 14: Initialization and Conditioning

The starting point of optimization matters. A neural network whose weights are initialized poorly may train slowly, fail to train at all, or produce gradients so extreme that float32 cannot represent them. The right initialization keeps activations and gradients in well-behaved ranges throughout the network, regardless of depth.

14.1 Why Initialization Matters

Consider a deep network whose weights are all initialized to zero. Every neuron in a layer computes the same thing; backprop produces identical gradients for them; they remain identical throughout training. The network has no symmetry-breaking, and effectively no learning happens. This is the simplest illustration that initialization is not arbitrary.

More subtly, even random initialization can be wrong. If weights are too large, activations grow with depth and saturate nonlinearities. If they are too small, activations shrink to zero and gradients vanish. The right scale depends on the layer width, the activation function, and the architecture.

14.2 Weight Initialization Strategies

Xavier (Glorot) initialization

Xavier Glorot and Yoshua Bengio (2010) proposed initializing weights from a distribution with variance 2 / (fan_in + fan_out). The derivation: for a linear layer with sigmoid or tanh activations, this variance keeps the variance of activations roughly constant from layer to layer, and similarly for the gradients in the backward pass.

He initialization

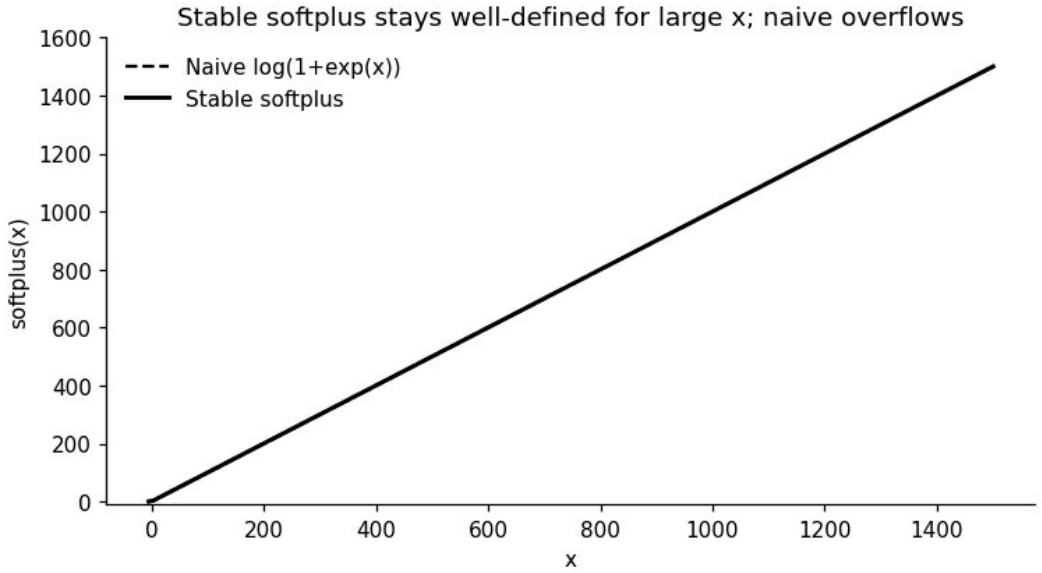


Figure 14.2: He initialization gives larger weight variance than Xavier; calibrated for ReLU.

Kaiming He et al. (2015) showed that Xavier underestimates the right variance for ReLU networks, because ReLU zeros out half the activations. The He initialization uses variance 2 / fan_in, which restores the correct variance for ReLU networks. This is the default for most

modern architectures.

Orthogonal initialization

For very deep networks, even He initialization can produce gradients that grow or shrink with depth. Orthogonal initialization sets the weight matrix to a random orthogonal matrix, scaled by the He factor. Orthogonal matrices preserve norms exactly, so gradient magnitudes are stable through arbitrary depth.

```
import torch.nn as nn

# Common initializations in PyTorch
nn.init.xavier_uniform_(layer.weight)
nn.init.kaiming_normal_(layer.weight, nonlinearity='relu')
nn.init.orthogonal_(layer.weight)
```

14.3 Batch Normalization

Batch normalization, introduced by Ioffe and Szegedy (2015), normalizes the activations of each layer to have zero mean and unit variance, computed over the mini-batch. The normalization is followed by a learned scale and shift, so the network can recover any desired distribution of activations.

What batch norm fixes

Without batch norm, each layer's input distribution shifts during training as upstream weights update, the so-called internal covariate shift. Batch norm fixes this by enforcing a stable distribution at each layer. The downstream effect: gradients are better conditioned, training can use larger learning rates, and the network is less sensitive to initialization.

Inference complications

Batch norm uses batch statistics during training but cannot use them at inference (when there may be only one example). Instead, running averages of the mean and variance are tracked during training and used at inference. The discrepancy between training and inference statistics can cause subtle issues, particularly when batch size is small or when the input distribution at inference differs from training.

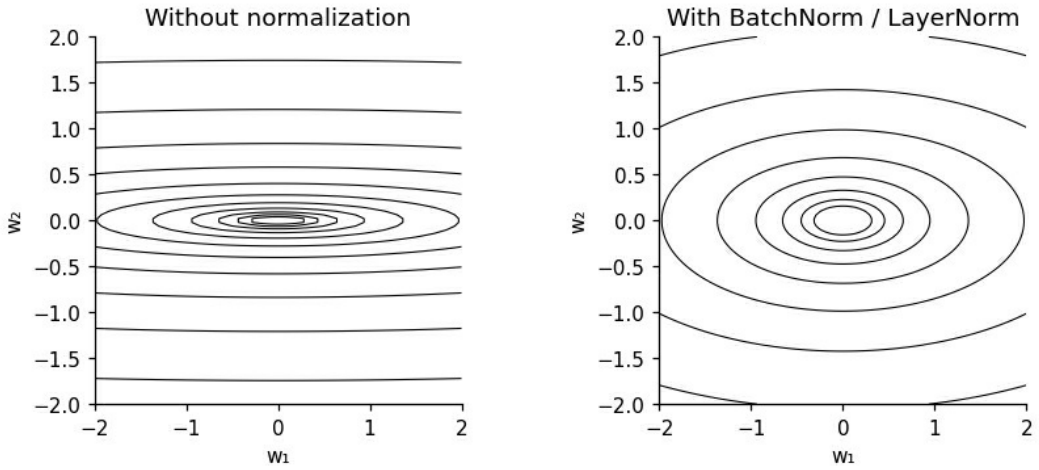


Figure 14.1: Effect of normalization on the loss landscape. Without normalization (left), the landscape is highly anisotropic; with BatchNorm or LayerNorm (right), it becomes more isotropic and easier to optimize.

14.4 Layer Normalization

Layer normalization, introduced by Ba, Kiros, and Hinton (2016), normalizes across features within each example, rather than across examples within each batch. Unlike batch norm, it does not depend on batch size and behaves identically at training and inference.

Where each is used

Batch norm dominates in convolutional networks for vision, where batch sizes are large and the activation statistics are stable. Layer norm dominates in transformers and language models, where sequence-position-dependent statistics make batch norm unsuitable. The choice is architectural and matters; using the wrong one can prevent training entirely.

Other normalizations

- Group normalization: normalize over groups of channels. Useful when batch size is small.
- Instance normalization: normalize each example separately. Useful in style transfer.
- RMS normalization: simpler than layer norm, scales by the RMS of activations. Used in modern LLMs.
- Weight normalization: reparameterize weights as direction times magnitude. An alternative to activation normalization.

14.5 Conditioning of Deep Networks

All of these techniques can be understood as conditioning interventions. They reshape the loss landscape so that gradient methods can navigate it efficiently.

The conditioning of a deep network

Deep networks have Hessians with eigenvalue spreads of many orders of magnitude. The largest eigenvalues are typically associated with the last layer's weights (which directly affect the output), the smallest with the first layer's (which affect the output through long chains).

Without intervention, gradient descent on such an ill-conditioned problem would crawl. The interventions, careful initialization, batch normalization, residual connections, adaptive optimizers like Adam, work together to compress the eigenvalue spectrum and make the problem tractable.

Residual connections as conditioning

Residual connections deserve special mention. By making the layer Jacobian I + Jf rather than Jf, they ensure that, in the limit of small Jf, gradients propagate as identity. Eigenvalues of I + Jf cluster near 1, regardless of the eigenvalues of Jf. This stabilizes the conditioning of arbitrarily deep networks.

ResNet's success at scaling to over 1000 layers is direct evidence of this. Without residual connections, networks beyond about 30 layers cannot be trained even with all the modern initialization tricks. With them, depth ceases to be an obstacle.

Key Takeaways

- Initialization sets the starting conditioning of optimization — bad initialization can make a network untrainable.

- He initialization works for ReLU networks; Xavier for tanh/sigmoid; orthogonal initialization for RNN-like architectures.
- Batch normalization and layer normalization stabilize activations during training, indirectly reducing conditioning.
- Residual connections preserve gradient magnitudes through arbitrary depth — they are why networks of hundreds of layers became feasible.
- Test your initialization by checking activation statistics layer-by-layer on the first forward pass.

Exercises

- [Advanced] Train a 30-layer MLP with three initialization schemes (zeros, Xavier, He) and observe which one trains.
- [Intermediate] Train the same network with and without batch normalization. Compare training curves.
- [Advanced] Train a deep network with and without residual connections at depth 50, 100, and 200. Find the depth at which non-residual networks fail.

Chapter 15: Regularization from a Numerical View

Regularization is usually presented as a way to control model complexity and prevent overfitting. It is also, from another perspective, a way to control the conditioning of optimization. The same techniques that help generalization often help numerical stability, and understanding both perspectives gives a deeper picture of why regularization works.

15.1 L2 Regularization

L2 regularization adds a penalty proportional to the squared Euclidean norm of the parameters: λ ||w||^2 / 2. Equivalently, the gradient gains a term λ w pulling weights toward zero. In the linear regression setting, this is ridge regression, with the closed-form solution w = (X^T X + λI)^{-1} X^T y.

Numerical effect: conditioning

The matrix X^T X + λI has eigenvalues bounded below by λ. So the condition number of the regularized problem is at most (λ + λ_max) / λ. If λ_max is, say, 10^8 and λ is 10^-2, the condition number is 10^10. Without regularization (λ = 0), it would be infinity if X^T X is singular. Regularization quite literally prevents the system from being uninvertible.

This is why ridge regression works in practice on problems where ordinary least squares does not. Multicollinearity in features can make X^T X numerically singular; ridge regression's regularization parameter is the floor of its eigenvalues.

Statistical effect: bias-variance tradeoff

From a statistical perspective, L2 regularization trades increased bias for decreased variance. The estimator is biased toward zero, it shrinks coefficients, but its sampling variance is smaller. For finite-sample problems, the smaller variance often wins, and the regularized estimator has lower mean squared error than the unbiased one.

15.2 L1 Regularization

L1 regularization adds a penalty proportional to the sum of absolute values of parameters: λ ||w||_1. Unlike L2, this penalty is non-differentiable at zero, and its effect on the optimum is qualitatively different: many weights become exactly zero, producing sparse solutions.

Why L1 produces sparsity

Geometrically, the L1 ball is a polytope with corners on the coordinate axes. The L2 ball is a sphere. When the squared-error contours are tangent to the constraint surface, they typically touch corners on the L1 ball (where some coordinates are zero) and arbitrary points on the L2 ball. Hence L1 sparsity, L2 shrinkage.

Numerical aspects

L1 regularization is non-smooth, which complicates optimization. Subgradient methods are slow; proximal gradient methods (also called ISTA) are the modern choice. Proximal methods handle L1 cleanly: the proximal operator of the L1 norm is soft-thresholding, which sets small values to zero exactly.

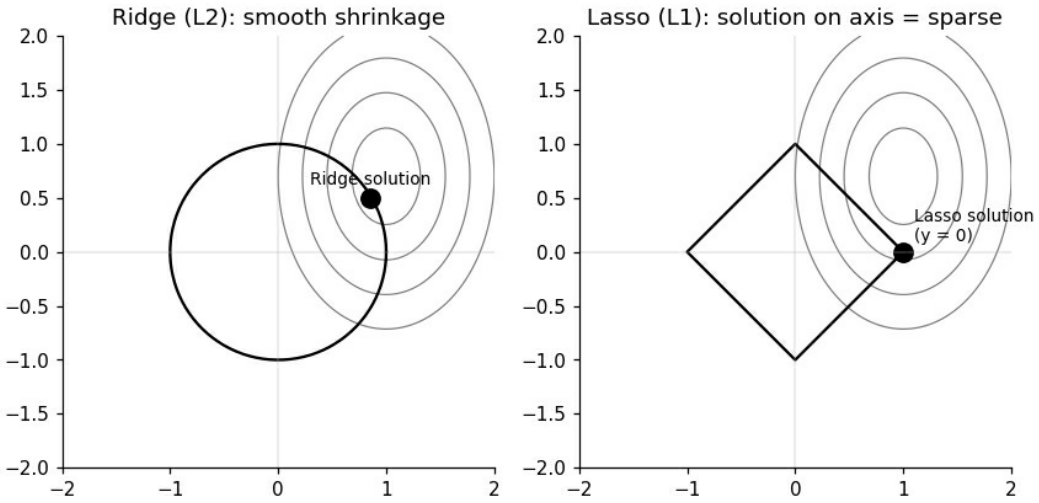


Figure 15.2: Ridge (L2) shrinks coefficients smoothly. Lasso (L1) produces sparse solutions.

```
def soft_threshold(x, lam):
    return np.sign(x) * np.maximum(np.abs(x) - lam, 0)

# Proximal gradient (ISTA) step for Lasso
for k in range(num_iter):
    grad = X.T @ (X @ w - y)
    w = soft_threshold(w - lr * grad, lr * lam)
```

15.3 Dropout as Stochastic Approximation

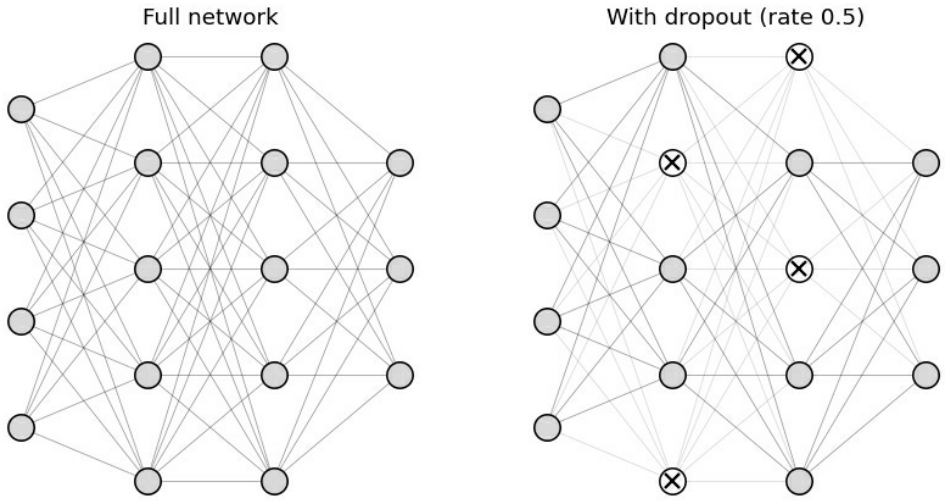


Figure 15.1: Dropout. During training, a random subset of units is zeroed each step, creating an implicit ensemble of subnetworks.

Dropout, introduced by Srivastava et al. (2014), randomly zeros out a fraction of activations at each forward pass during training. At test time, all activations are kept and scaled appropriately. Dropout has been one of the most successful regularization techniques in deep learning.

What dropout does numerically

Dropout introduces noise into each forward pass. From a Bayesian perspective, training with dropout approximates inference in a model where weights have a particular prior; this is the

variational interpretation of dropout. From a practical perspective, the noise prevents co-adaptation of neurons, a unit cannot rely on any specific other unit being present, since any of them might be dropped.

Dropout's effect on the loss landscape is to smooth it, in roughly the same way that adding noise to gradients smooths the optimization. This makes the landscape easier to navigate and produces minima that generalize better.

Inference and dropout

At inference, we want the expected output rather than a sample. Inverted dropout, scaling activations during training so the expectation is preserved without test-time scaling, is the standard implementation. PyTorch's Dropout module handles this automatically.

15.4 Early Stopping as Numerical Control

Early stopping halts training when validation performance stops improving. It is one of the simplest and most effective regularization techniques, and it has a precise relationship to L2 regularization.

The connection to L2

For linear regression with gradient descent and small step size, training for k iterations produces approximately the same regularization effect as L2 regularization with parameter inversely proportional to k. Early stopping is implicit L2.

For non-linear models the correspondence is no longer exact, but the intuition remains: stopping early prevents the optimizer from overfitting to noise in the training data, just as a regularization penalty does. Early stopping has the practical advantage of not requiring a hyperparameter search over λ, instead it requires a validation set and a stopping criterion.

Beyond early stopping: dynamic regularization

More sophisticated forms include cosine annealing of the learning rate (which has implicit regularization effects), stochastic weight averaging (which finds wider minima), and lookahead optimizers (which interpolate between fast and slow weight trajectories). Each of these can be understood as a form of regularization that emerges from how training proceeds, not from an explicit penalty term.

Key Takeaways

- Regularization simultaneously controls model complexity (statistical view) and conditioning of optimization (numerical view).
- L2 (ridge) regularization prevents singular matrices and shrinks coefficients smoothly.
- L1 (lasso) regularization produces sparse solutions by hitting axes of the coefficient simplex.
- Dropout introduces noise that smooths the loss landscape; early stopping is implicit regularization.
- Each regularizer addresses both statistics and numerics — they often combine well in practice.

Exercises

- [Intermediate] Compare ridge regression and ordinary least squares on a problem with multicollinear features. Verify that ridge produces stable solutions when OLS fails.

- [Intermediate] Implement Lasso via proximal gradient descent and verify that solutions become sparse for large λ.
- [Intermediate] Train a network with and without dropout. Compare training and validation loss.
- [Intermediate] Train a model and use early stopping. Experiment with different patience values for the stopping criterion.

Chapter 16: Optimization Pathologies

Even with all the modern interventions, deep learning optimization can go wrong in characteristic ways. This chapter is a tour of the pathologies, what they look like, why they occur, and what to do about them.

16.1 Vanishing Gradients

The vanishing gradient problem is the failure mode where, in a deep network, gradients become so small as to be indistinguishable from zero. Training stalls. Loss does not decrease. The network does not learn.

Diagnosing it

Vanishing gradients are visible in gradient magnitudes by layer. Plot the norm of the gradient at each layer; if it decreases by orders of magnitude from output to input, gradients are vanishing. Float32 underflows around 10^-38, but gradients much larger than that, say, below 10^-7, are effectively zero relative to typical weight magnitudes and produce no learning.

Causes

- Saturating activations: sigmoid and tanh saturate near zero gradients in their tails.
- Poor initialization: weights too small, activations shrinking with depth.
- Lack of normalization: activation distributions drift to extreme regions.
- Lack of skip connections: Jacobians multiplied with no rescue from identity components.

Fixes

ReLU activations, He initialization, batch or layer normalization, and residual connections are the standard cure. Each addresses one of the causes; together they essentially eliminate vanishing gradients in feedforward and convolutional networks. Modern architectures ship with these built in.

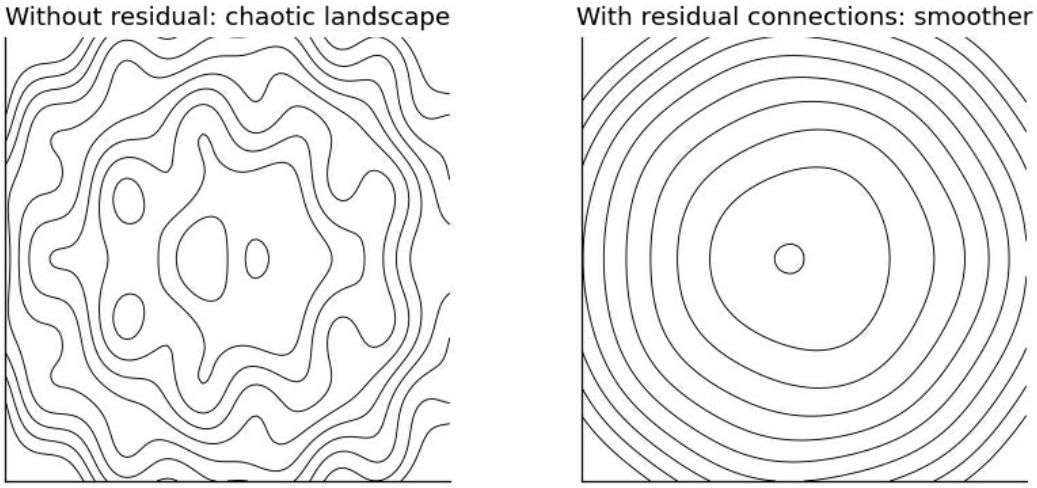


Figure 16.3: Residual connections smooth the loss landscape.

16.2 Exploding Gradients

The mirror image: gradients grow unbounded with depth or sequence length, eventually overflowing into infinity or NaN. We discussed this in chapter 1 as our case study; here we summarize the practical responses.

Gradient clipping

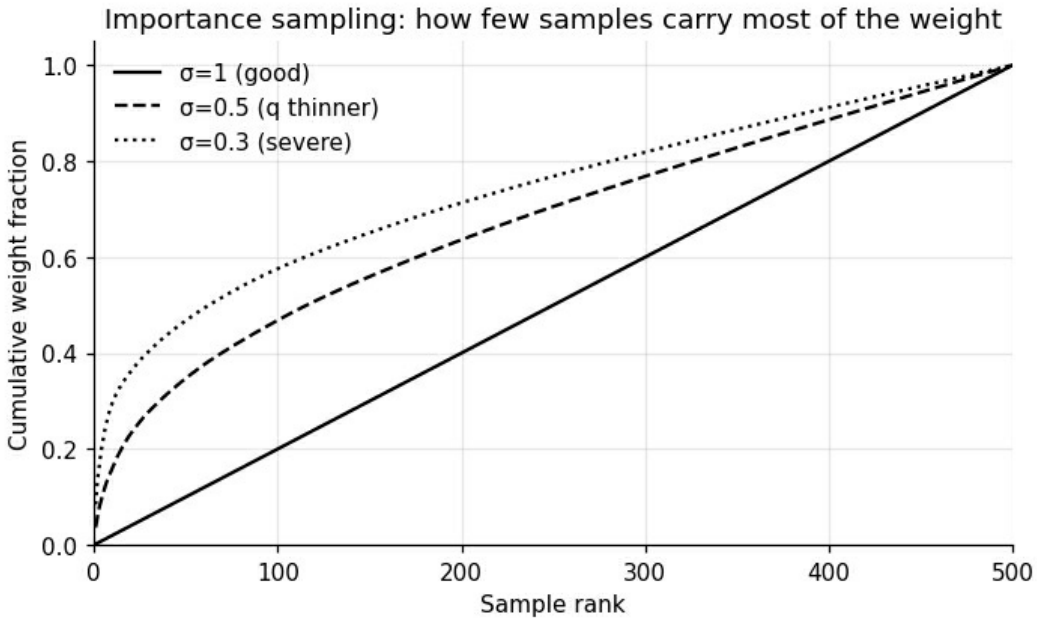


Figure 16.2: Gradient clipping bounds rare large updates that would otherwise destabilize training.

The standard remedy is gradient clipping. Compute the total gradient norm; if it exceeds a threshold, scale the gradient down to that threshold. Direction is preserved; magnitude is bounded. Clipping at a value like 1.0 is the default in many transformer training recipes.

Element-wise clipping (capping each gradient coordinate independently) is also used but is less principled, it can change the gradient direction in arbitrary ways. Norm clipping is preferred.

Architectural fixes

Long short-term memory (LSTM) and gated recurrent units (GRU) have gating mechanisms that bound the effective recurrence and prevent gradients from exploding through long sequences. The same principle drives the success of transformers, which replace recurrence with attention and avoid long chains of multiplications entirely.

16.3 Saddle Point Plateaus

In high-dimensional non-convex optimization, saddle points dominate. At a saddle, the gradient is small and the loss is roughly constant in a neighborhood, training looks stuck. But unlike a local minimum, there is a direction of descent if we can find it.

Diagnosing a plateau

Loss is decreasing very slowly. Gradient magnitudes are small but nonzero. Validation accuracy is not improving. This pattern, lasting many iterations, often indicates a saddle plateau.

Escaping

Stochastic gradient descent escapes saddles eventually, because the noise in mini-batch gradients perturbs the iterate off the saddle. The escape rate depends on the negative curvature direction's magnitude and the noise level. In some cases, increasing the batch size helps (more accurate gradients direct the iterate more efficiently); in others, decreasing it helps (more noise, faster escape).

Saddle-aware methods, like the saddle-free Newton of Dauphin et al. (2014), use second-order information to actively escape saddles. They are not common in practice, the cost of Hessian information typically does not justify itself, but they illustrate that saddles are an algorithmic obstacle that can be addressed.

16.4 Flat vs Sharp Minima

A growing body of work suggests that flat minima generalize better than sharp ones. Intuitively, a flat minimum has a wide basin where the loss stays low even if weights are slightly perturbed; this corresponds to a robust solution that doesn't depend on exact weight values. A sharp minimum has a narrow valley where small perturbations cause large loss increases.

Why optimizers find flatter minima

SGD, with its inherent noise, tends to find flatter minima, the noise prevents the iterate from settling in a sharp narrow valley. Larger learning rates have a similar effect. This is one explanation for why SGD often generalizes better than methods like Adam, which find lower training loss but in regions of the loss surface that are sharper.

Sharpness-aware minimization

Foret et al. (2020) introduced sharpness-aware minimization (SAM), which explicitly biases optimization toward flat minima by minimizing the maximum loss in a neighborhood of each point, not the loss at the point itself. SAM has shown improved generalization on various benchmarks. The cost is an extra forward-backward pass per step, which is expensive but has proven worth it for some applications.

Key Takeaways

- Deep learning optimization fails in characteristic ways: vanishing gradients, exploding gradients, saddle plateaus, sharp minima.
- Each pathology has a known fix: initialization, normalization, residual connections, gradient clipping, careful architecture, or noise injection.
- Diagnose first (which pathology is hitting you?), then intervene (which fix is the right one?).
- Gradient clipping is a band-aid that works; residual connections are a structural fix that usually works better.
- Sharpness-aware minimization is a recent technique that targets the flatness of minima directly.

Exercises

- [Advanced] Train a deep network and visualize gradient norms by layer over training. Identify any vanishing or exploding behavior.
- [Intermediate] Train an RNN on a sequence task at length 200 with and without gradient clipping. Compare training stability.
- [Intermediate] Construct a problem with a known saddle point. Show that gradient

descent stalls there but SGD with noise escapes.

- [Research-Level] Read the SAM paper. Implement it and test whether it improves generalization on a small classification benchmark.

Part V

Probabilistic & Statistical Numerical Methods

Probability is woven through machine learning. Every classifier outputs a distribution. Every Bayesian model demands integration over high-dimensional posteriors. Every variational autoencoder, every diffusion model, every Monte Carlo estimator depends on numerical methods that approximate integrals, sample from intractable distributions, and optimize stochastic objectives.

The numerical challenges in probabilistic computing differ from those in deterministic optimization. We are no longer trying to find a single point that minimizes a function, we are trying to characterize an entire distribution. The notions of error, convergence, and stability all take on stochastic flavors. A Monte Carlo estimate has variance as well as bias. An MCMC chain has burn-in, autocorrelation, and effective sample size. A variational approximation has both the bias of the family and the optimization error of the fit.

Part V develops the numerical tools that bring probability theory into computational reach: sampling methods, Monte Carlo integration, maximum likelihood and variational inference, and Markov chain Monte Carlo. Throughout, we keep one foot in the algorithm and one foot in the numerics, because the gap between a probabilistic model that works on paper and one that works on a GPU is filled with floating-point gotchas.

Chapter 17: Numerical Methods in Probability

Probability theory tells us what to compute. Numerical methods tell us how. The bridge between the two is sampling: drawing pseudo-random numbers from distributions to approximate expectations, integrals, and other quantities that would otherwise be intractable. This chapter develops the foundational sampling toolkit and the Monte Carlo principle that turns randomness into computation.

17.1 Sampling Methods

All sampling begins with the uniform distribution on the unit interval. Modern pseudorandom number generators produce sequences of numbers that pass statistical tests for uniformity and independence, even though they are deterministic functions of an internal state. From this primitive, we construct samples from arbitrary distributions.

Inverse transform sampling

If F is the cumulative distribution function of a random variable X, and U is uniform on (0,1), then $F^{-1}(U)$ is distributed as X. The method is exact and elegant when F^{-1} has a closed form, exponential, Cauchy, Pareto, but useless when it does not. The Gaussian, the workhorse of ML, has no closed-form CDF inverse, so we resort to other techniques.

Box-Muller and the Gaussian

Two independent uniforms can be transformed into two independent standard normals via the Box-Muller transform:

$$Z_1 = \sqrt{(-2 \ln U_1)} \cdot \cos(2\pi U_2), \quad Z_2 = \sqrt{(-2 \ln U_1)} \cdot \sin(2\pi U_2)$$

This is exact in exact arithmetic. In floating-point, U_1 near zero produces large negative logarithms and possible overflow; most production implementations use the polar variant, which avoids the trigonometric calls and is more numerically stable. NumPy, PyTorch, and modern libraries use ziggurat or related algorithms for speed, but the principle is the same: Gaussians come from carefully transformed uniforms.

Rejection sampling

When neither inverse transform nor a closed-form transformation applies, rejection sampling offers a general but sometimes wasteful approach. We sample from a proposal distribution q whose density dominates the target p (up to a constant M), accept with probability $p(x)/(M \cdot q(x))$, and reject otherwise. The accepted samples are exactly distributed as p.

The numerical pitfall is M. If the proposal is poorly matched to the target, M must be enormous to satisfy the dominance condition, and the acceptance rate becomes vanishingly small. In high dimensions this becomes catastrophic, rejection sampling is a 1D and 2D tool, not a 100D tool.

Importance sampling

Rather than rejecting samples, importance sampling reweights them. To estimate E_p[f(X)] we draw from q and compute:

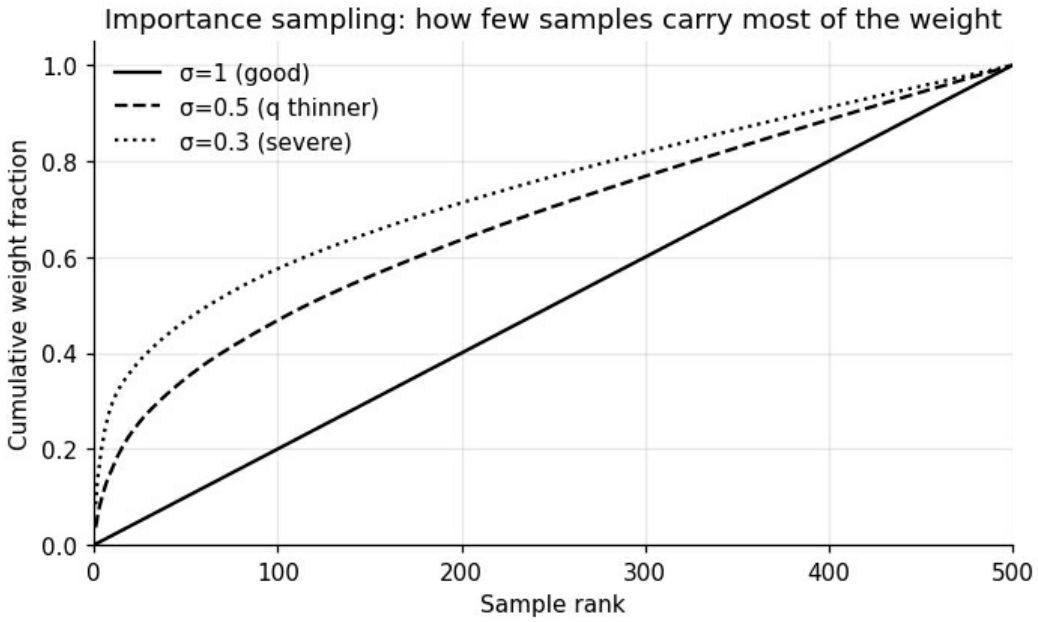


Figure 17.2: Importance sampling weight concentration. When the proposal q is mismatched with the target p, a small fraction of samples carries most of the cumulative weight, signaling effective sample size collapse.

$$\hat{E} = (1/N)\ \Sigma\ f(x_i) \cdot w(x_i), \quad w(x) = p(x)/q(x)$$

The estimator is unbiased, but its variance depends critically on the match between p and q. When q has thinner tails than p, a few samples receive enormous weights and the estimator becomes dominated by outliers, the effective sample size collapses. The diagnostic is the variance of the weights, or equivalently:

$$ESS = (\Sigma\ w_i)^2 / \Sigma\ w_i^2$$

If the ESS is much smaller than N, the estimator is unreliable regardless of how large N is.

17.2 Monte Carlo Integration

The defining application of sampling is integration. Monte Carlo replaces the deterministic quadrature rules of classical numerical analysis with a stochastic estimator that, remarkably, is dimension-independent in its convergence rate.

To estimate an integral we recognize it as an expectation:

$$\int f(x)\ p(x)\ dx = E_p[f(X)] \approx (1/N)\ \Sigma\ f(x_i), \quad x_i \sim p$$

The error of this estimate decreases as $O(1/\sqrt{N})$, regardless of the dimension of the integration domain. Compare this with the trapezoidal rule, which converges as O(N^(−2/d)) in d dimensions and becomes useless beyond d = 5 or so. The curse of dimensionality, which paralyzes deterministic quadrature, leaves Monte Carlo essentially unaffected.

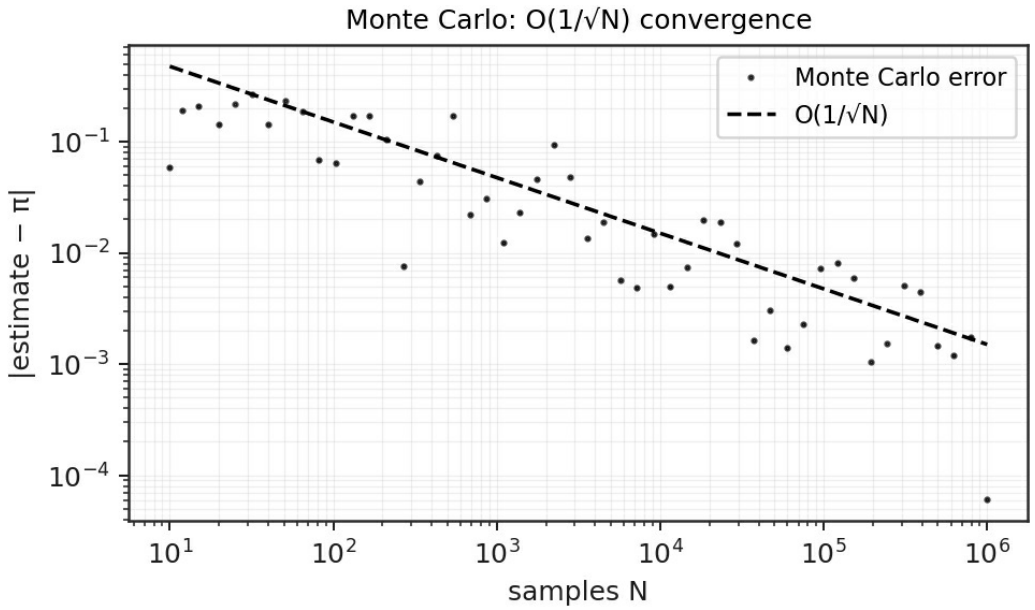


Figure 17.1: Monte Carlo estimator error decreases as 1/√N regardless of dimension.

Variance reduction

The price of dimension-independence is the slow O(1/√N) rate. To halve the standard error we need four times the samples. Variance reduction techniques attack this constant factor.

- Antithetic variates: pair each sample x with −x (or its reflection through the mean), exploiting negative correlation to halve the variance for symmetric integrands.
- Control variates: subtract a function with known integral that correlates with the integrand.
- Stratified sampling: partition the domain and sample within each stratum, reducing the within-cell variance.
- Quasi-Monte Carlo: replace pseudorandom sequences with low-discrepancy sequences (Sobol, Halton), achieving rates closer to O(1/N) for smooth integrands in moderate dimensions.

17.3 Law of Large Numbers in Practice

The strong law of large numbers guarantees that sample averages converge to expectations almost surely. The central limit theorem refines this: the deviation of the sample mean from the population mean is approximately Gaussian with standard error σ/√N. These theorems underpin every Monte Carlo estimate.

In practice, several caveats apply. First, the variance σ² must be finite. Distributions with heavy tails, Cauchy, certain power laws, have infinite variance, and the CLT fails. The sample mean does not converge to a Gaussian; it has its own non-trivial distribution, and confidence intervals built from sample standard deviations are misleading.

Second, finite-sample behavior can deviate sharply from asymptotic predictions. With small N, the sample standard deviation underestimates the true σ, and naive confidence intervals are too narrow. Bootstrap methods and Student's t corrections address this.

Third, in ML we frequently estimate expectations of unbounded or extremely concentrated functions, loss values that span many orders of magnitude, gradient norms that occasionally explode. A single outlier can dominate the average for many iterations. Defensive techniques include median-of-means estimators, trimmed means, and confidence-interval-aware stopping rules.

Code example: Monte Carlo estimate of π

```
import numpy as np

def estimate_pi(n_samples, rng=None):
    rng = rng or np.random.default_rng(0)
    x = rng.uniform(-1.0, 1.0, n_samples)
    y = rng.uniform(-1.0, 1.0, n_samples)
    inside = (x*x + y*y) <= 1.0
    p_hat = inside.mean()
    pi_hat = 4.0 * p_hat
    se = 4.0 * np.sqrt(p_hat * (1 - p_hat) / n_samples)
    return pi_hat, se

for n in [10**3, 10**4, 10**5, 10**6]:
    pi_hat, se = estimate_pi(n)
    print(f"N={n:>7}: pi ≈ {pi_hat:.5f} ± {se:.5f}")
```

Key Takeaways

- Monte Carlo trades guaranteed accuracy for dimension-independence — its $O(1/\sqrt{N})$ rate is slow but unaffected by problem dimension.
- Sample mean converges to the population mean at rate $1/\sqrt{N}$; always report a standard error alongside the estimate.
- Variance reduction techniques (control variates, antithetic sampling) can dramatically lower the constant in the $1/\sqrt{N}$ rate.
- Importance sampling reweights samples from a different distribution but suffers if the proposal is poorly matched to the target.

Exercises

- [Intermediate] Implement inverse transform sampling for the exponential distribution. Verify empirically that the sample mean converges to $1/\lambda$ at rate $1/\sqrt{N}$.
- [Intermediate] Construct an importance sampling estimator where p is N(0, 1) and q is $N(0, \sigma^2)$. Plot the variance of the estimator as a function of σ. Where is the optimum?
- [Advanced] Show by example that estimating E[X] when X is Cauchy distributed gives a sample mean that does not converge, run the average of 10^6 samples and observe the long-run instability.
- [Intermediate] Implement antithetic variates for estimating $E[(X-1)^2]$ where X is uniform on [0,1]. Compare variance with naive Monte Carlo.

Chapter 18: Optimization in Probabilistic Models

Probabilistic models give us a principled framework for inference, but they make every prediction expensive. Where a deterministic model returns a single output, a probabilistic model returns a distribution, and computing or summarizing that distribution typically reduces to optimization. Maximum likelihood estimation, expectation maximization, and variational inference all turn probabilistic questions into optimization problems, often nonconvex and high-dimensional.

18.1 Maximum Likelihood Estimation

Given data $D = \{x_1, ..., x_N\}$ and a parametric family $p(x \mid \theta)$, the maximum likelihood estimator chooses θ to maximize the likelihood:

$$\hat{\theta} = \operatorname{argmax}_\theta \sum_i \log p(x_i \mid \theta)$$

We work with log-likelihoods rather than likelihoods for both numerical and computational reasons. A product of small probabilities underflows to zero almost immediately; the sum of log-probabilities does not. Differentiation of a sum is also cheaper than differentiation of a product, and gradient methods naturally compose with the log-sum form.

Numerical issues in MLE

MLE for nontrivial models is rarely closed-form. We rely on the optimization machinery of Parts II and IV: gradient descent, Newton's method, quasi-Newton methods. The objective is often nonconvex, especially for latent-variable models, mixtures, and neural networks with stochastic outputs.

Common pitfalls include:

- Boundary degeneracies. The likelihood of a Gaussian mixture is unbounded above, a single component can collapse onto a data point with vanishing variance, sending the likelihood to infinity. MLE has no finite maximum, and gradient methods drift toward this singularity. Regularization or careful initialization is essential.
- Overflow in exponentials. Computing $\log p(x|\theta)$ for an exponential family requires care: the partition function may involve sums of exponentials of large quantities. The log-sum-exp trick (Chapter 20) is mandatory.
- Identifiability. Many probabilistic models have symmetric parameterizations, mixture component labels, rotational symmetry in factor models. The likelihood has multiple equivalent maxima, and gradient methods can stall at saddle points connecting them.

18.2 Gradient-Based Inference

For models with continuous parameters, gradient ascent on the log-likelihood is the standard tool. For exponential families and generalized linear models, the gradient has a particularly clean form:

$$\nabla_\theta \log p(x \mid \theta) = T(x) - E_\theta[T(X)]$$

where $T(x)$ is the sufficient statistic. The gradient measures the gap between the empirical sufficient statistics and their expectation under the current model, zero when the model fits perfectly. This is the moment-matching interpretation of MLE.

For models that are not in the exponential family, or for which the expectation is intractable, we estimate gradients with Monte Carlo. The score function estimator, also known as REINFORCE, expresses the gradient of an expectation as an expectation:

$$\nabla_\theta \, E_{p\theta}[f(X)] = E_{p\theta}[f(X) \, \nabla_\theta \log p\theta(X)]$$

This estimator is unbiased but high-variance. The variance often grows with the magnitude of f and with the dimension of x. Variance reduction is essential, and modern probabilistic ML invests heavily in techniques like reparameterization, control variates, and Rao-Blackwellization.

Reparameterization trick

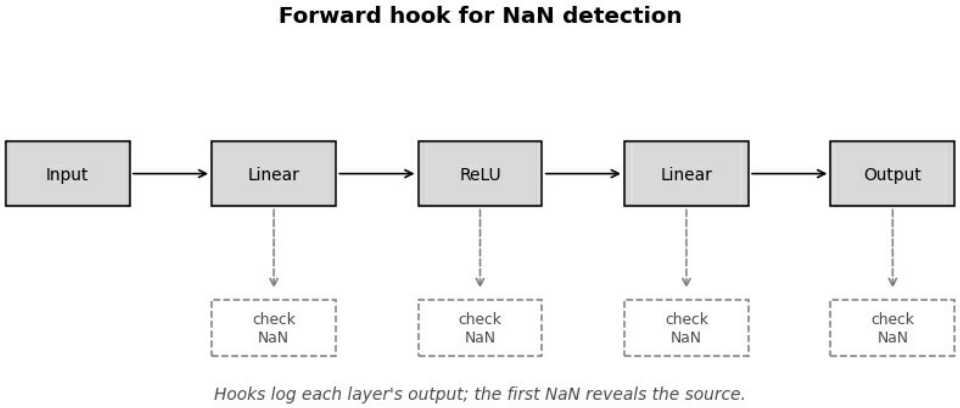


Figure 18.2: The reparameterization trick routes randomness through ε so gradients can flow through θ.

When the distribution can be written as a deterministic transformation of a parameter-free noise variable, we can move the gradient inside the expectation. For a Gaussian $X \sim N(\mu, \sigma^2)$, we write $X = \mu + \sigma \cdot \varepsilon$ with $\varepsilon \sim N(0, 1)$:

$$\nabla_{\mu,\sigma} \, E[f(X)] = E_\varepsilon[\nabla_{\mu,\sigma} \, f(\mu + \sigma \cdot \varepsilon)]$$

This estimator typically has dramatically lower variance than the score function estimator. It is the engine of variational autoencoders and most modern continuous-relaxation techniques.

18.3 Variational Inference Basics

Bayesian inference asks for the posterior $p(\theta \mid D)$, which is rarely tractable. Variational inference (VI) sidesteps the intractable computation by approximating the posterior with a simpler distribution $q(\theta; \varphi)$ chosen from a tractable family, and optimizing the variational parameters φ to make q as close as possible to the true posterior.

The objective is the Evidence Lower Bound, or ELBO:

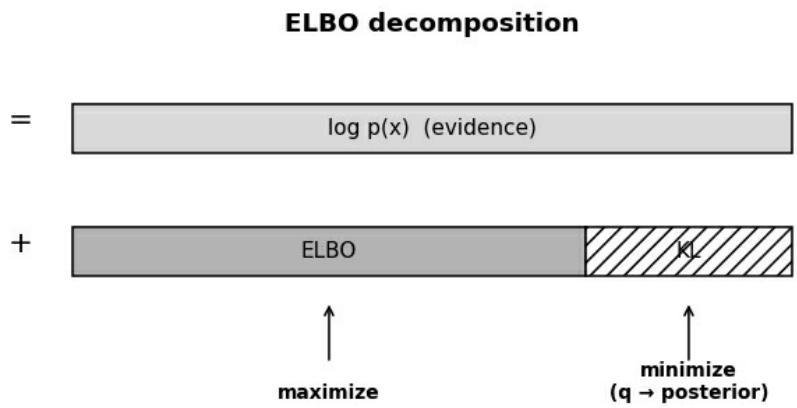


Figure 18.3: ELBO decomposition. Maximizing ELBO simultaneously minimizes KL.

$$ELBO(\varphi) = E_q[\log p(D, \theta)] - E_q[\log q(\theta; \varphi)]$$

Maximizing the ELBO is equivalent to minimizing KL(q ∥ p(·|D)). The first term encourages q to place mass where the joint is large, where the data is well-explained. The second term, the entropy of q, penalizes q from collapsing to a delta. The interplay produces a posterior approximation that is concentrated where the likelihood is informative but spread according to the prior in directions where the data is silent.

Mean-field VI

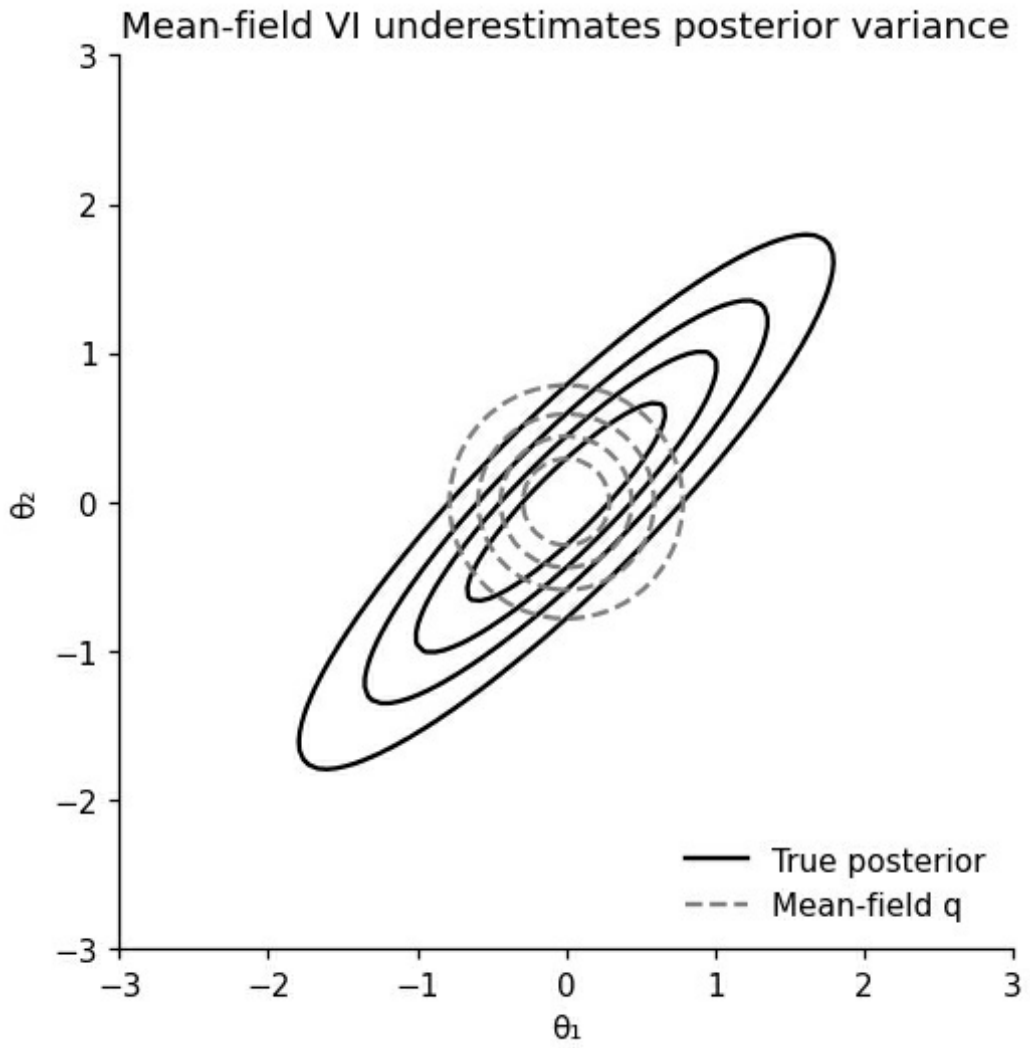


Figure 18.4: Mean-field VI underestimates posterior variance when the true posterior is correlated.

The simplest variational family factorizes the posterior across coordinates: $q(\theta) = \prod_j q_j(\theta_j)$. For conditionally conjugate models, coordinate ascent updates the optimal q_j in closed form. The CAVI algorithm cycles through coordinates, updating each in turn. It is reminiscent of Gauss-Seidel iteration and inherits both the convergence guarantees and the slowness of that method when coordinates are correlated.

Stochastic VI

For large datasets, even one pass through the data is expensive. Stochastic variational inference applies the same SGD principles from Part II: subsample the data, estimate a noisy ELBO gradient, and step. Combined with the reparameterization trick for continuous latent variables, this scales VI to datasets with millions of examples and high-dimensional latent spaces, the foundation of variational autoencoders, deep latent Gaussian models, and modern probabilistic deep learning.

Numerical pitfalls in VI

- ELBO tightness vs. posterior accuracy. A tight ELBO does not imply an accurate posterior. KL(q ∥ p) penalizes q for placing mass where p has none, but not the reverse.

Mean-field VI tends to underestimate posterior variance and miss multimodality.

- Variance of stochastic ELBO gradients. Score-function estimators can have variance that swamps the signal. Always prefer reparameterization when the latent distribution allows it.
- Optimization plateaus. The ELBO landscape often has long flat regions where many parameterizations of q give similar values. Adaptive learning rates (Adam, RMSProp) help; pure SGD often does not.
- Numerical underflow in entropy and log-probability terms. Compute log q(θ) directly from the parameterization, never as log of an exponentiated value.

Code example: VI for Bayesian regression

```
import torch
import torch.nn as nn

class VIGaussianRegression(nn.Module):
    def __init__(self, dim, prior_var=1.0):
        super().__init__()
        self.mu = nn.Parameter(torch.zeros(dim))
        self.log_sigma = nn.Parameter(torch.full((dim,), -2.0))
        self.prior_var = prior_var

    def sample_w(self, n=1):
        sigma = self.log_sigma.exp()
        eps = torch.randn(n, *self.mu.shape)
        return self.mu + sigma * eps  # reparameterization

    def elbo(self, X, y, sigma_noise=1.0, n_samples=1):
        w = self.sample_w(n_samples)
        pred = X @ w.T
        ll = -0.5 * ((y[:, None] - pred) ** 2 / sigma_noise**2).sum(0).mean()
        sigma2 = (2 * self.log_sigma).exp()
        kl = 0.5 * (
            (sigma2 + self.mu.pow(2)) / self.prior_var
            - 1.0 - sigma2.log()
            + torch.log(torch.tensor(self.prior_var))
        ).sum()
        return ll - kl
```

Key Takeaways

- Probabilistic optimization differs from deterministic in two ways: gradients are stochastic, and objectives involve intractable expectations.
- The reparameterization trick routes randomness through an auxiliary variable so gradients can flow through distribution parameters.
- The log-sum-exp trick and careful parameterization avoid overflow in probability computations.
- Mean-field VI is a workhorse but underestimates posterior variance — know its biases before trusting its uncertainties.
- ELBO maximization simultaneously minimizes KL divergence between the variational approximation and the true posterior.

Exercises

- [Basic] Derive the closed-form MLE for the parameters of a univariate Gaussian. Verify that the sample mean and sample variance maximize the log-likelihood.

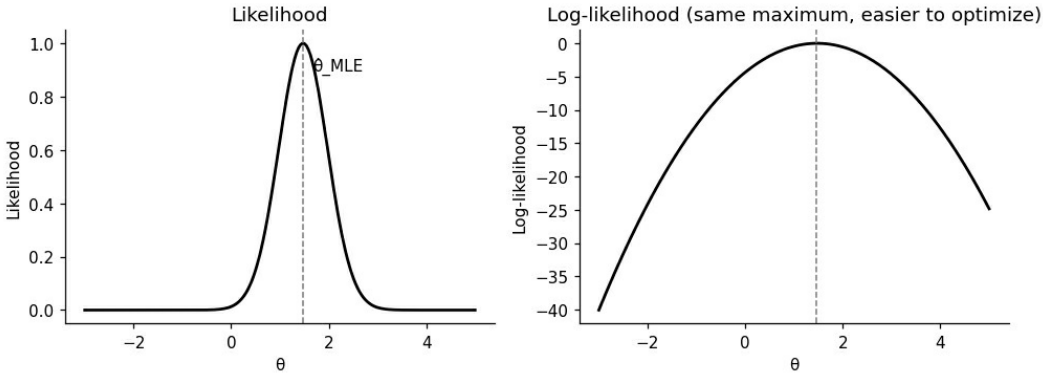


Figure 18.1: Likelihood and log-likelihood reach the same maximum; the log-form is easier to optimize.

- [Intermediate] Implement REINFORCE for a Bernoulli policy with parameter p. Compare gradient variance with the analytical gradient.
- [Intermediate] For a 1D mean-field VI approximation of a 2D Gaussian posterior with strong correlation, show that the optimal mean-field q underestimates the marginal variances. Quantify the underestimation.
- [Intermediate] Implement a variational autoencoder on MNIST with a 2D latent space. Plot the learned latent manifold.

Chapter 19: Bayesian Computation

Variational inference trades fidelity for tractability, a parametric q is fast but biased. Markov chain Monte Carlo (MCMC) takes the opposite trade-off: by constructing a Markov chain whose stationary distribution is the target posterior, we obtain samples that, in the limit, are exact. The cost is computation and a delicate set of convergence diagnostics.

19.1 Markov Chain Monte Carlo

Given an unnormalized target density $\pi(\theta) \propto p(D \mid \theta)\, p(\theta)$, MCMC constructs a transition kernel $T(\theta' \mid \theta)$ such that the resulting chain has π as its stationary distribution. Once the chain has burned in and reached stationarity, samples can be averaged to estimate posterior expectations:

$$E_\pi[f(\theta)] \approx (1/M)\, \Sigma_{m=1}^{M} f(\theta_m)$$

The samples are correlated, autocorrelation between successive iterates inflates the variance of the estimator beyond the iid Monte Carlo rate. The effective sample size, $ESS = M / (1 + 2\, \Sigma\, \rho_k)$, captures the actual statistical content of the chain.

Metropolis-Hastings

The Metropolis-Hastings algorithm proposes a new state $\theta' \sim q(\cdot|\theta)$ and accepts it with probability:

$$\alpha = \min(1, [\pi(\theta')\, q(\theta|\theta')] / [\pi(\theta)\, q(\theta'|\theta)])$$

If accepted, the chain moves to θ'; otherwise it stays at θ. The acceptance ratio cancels the unnormalized constant of π, allowing us to sample from posteriors whose normalizing constant (the marginal likelihood) is intractable.

The proposal q must be tuned. Too narrow, and the chain crawls; too wide, and most proposals are rejected. The optimal acceptance rate for high-dimensional Gaussian targets is approximately 0.234, a result from Roberts, Gelman, and Gilks. Adaptive MCMC tunes the proposal during a warm-up period to target this rate.

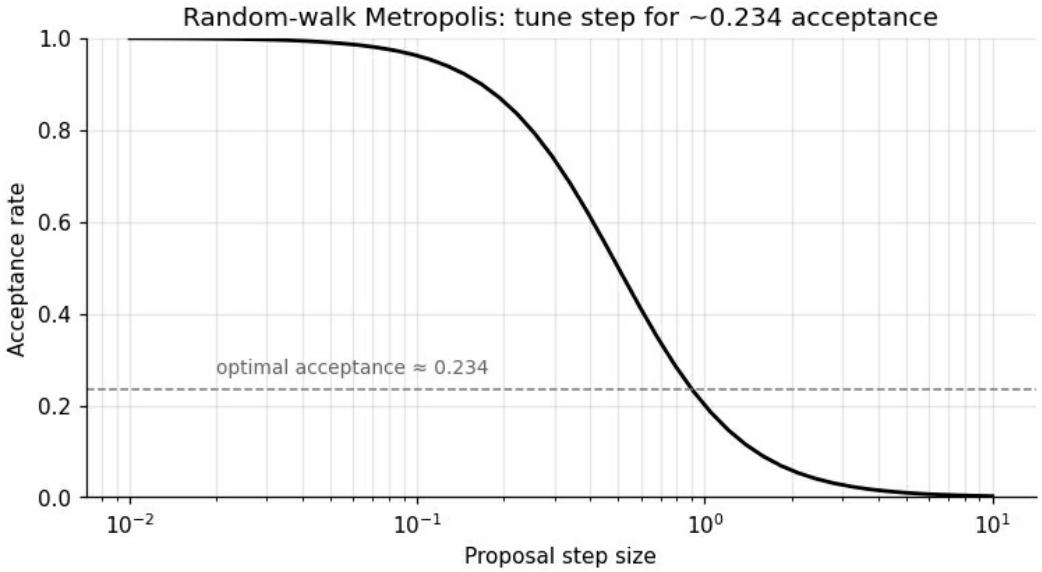


Figure 19.3: Random-walk Metropolis: tune proposal step for acceptance rate ~0.234.

Hamiltonian Monte Carlo

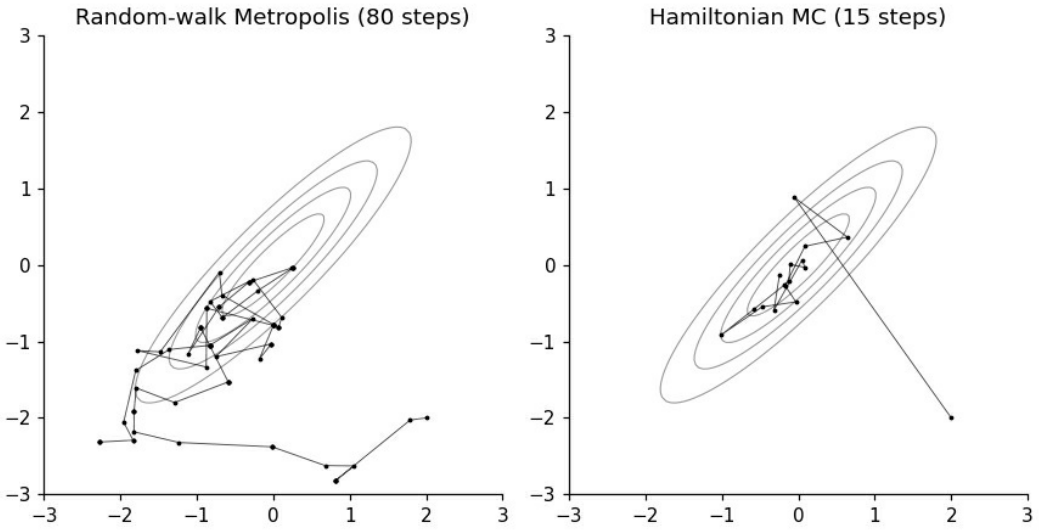


Figure 19.2: Random-walk Metropolis (left) explores the correlated posterior slowly; Hamiltonian Monte Carlo (right) takes long, well-directed steps.

For high-dimensional smooth posteriors, random-walk Metropolis becomes catastrophically slow, the chain explores by diffusion, taking O(d) steps to traverse a posterior of dimension d. Hamiltonian Monte Carlo (HMC) replaces the random walk with a deterministic dynamical system. By augmenting θ with momentum p, defining a Hamiltonian $H(\theta, p) = -\log \pi(\theta) + (1/2)p^{T}M^{-1}p$, and integrating Hamilton's equations with a leapfrog scheme, HMC produces proposals that traverse the posterior efficiently.

The gradient $\nabla \log \pi$ is required, HMC only works for differentiable posteriors. The leapfrog integrator is symplectic and time-reversible, properties essential for the validity of the Metropolis correction. Modern implementations (NUTS, the No-U-Turn Sampler, used in Stan, PyMC, and Pyro) automatically tune the trajectory length and step size, requiring no manual intervention from the user. NUTS is the gold standard for general-purpose Bayesian inference at moderate scale.

19.2 Gibbs Sampling

When the conditional distributions $p(\theta_j \mid \theta_{-j}, D)$ can be sampled directly, Gibbs sampling provides a special case of Metropolis-Hastings with always-accept proposals. The chain cycles through coordinates (or blocks), drawing each from its full conditional:

$$\theta_j^{(t+1)} \sim p(\theta_j \mid \theta_{-j}, D)$$

Gibbs sampling shines for conditionally conjugate models, Gaussian latent variables, Dirichlet-categorical models, hierarchical regressions, where the conditionals are standard distributions with closed-form sampling routines. It struggles when coordinates are highly correlated, since each step changes only one coordinate at a time.

Block Gibbs sampling updates groups of coordinates jointly. Choosing blocks well is an art: blocks must be small enough that the joint conditional is tractable, but large enough to capture the principal correlations of the posterior.

19.3 Numerical Challenges in Bayesian ML

Bayesian computation strains numerical precision in several characteristic ways.

Log-space arithmetic

Posterior densities span many orders of magnitude. A direct evaluation of p(D | θ) for moderate N immediately underflows. All computation occurs in log space: log-likelihoods are summed, ratios become differences, and the Metropolis ratio is computed as min(0, log α). Code that does not stay in log space breaks for any nontrivial dataset.

Mixing and convergence diagnostics

MCMC samples are useful only after the chain has converged. Diagnostics include:

- Trace plots: visual inspection of the parameter trajectories. Healthy chains look like fuzzy caterpillars; pathological chains drift, stick, or oscillate slowly.

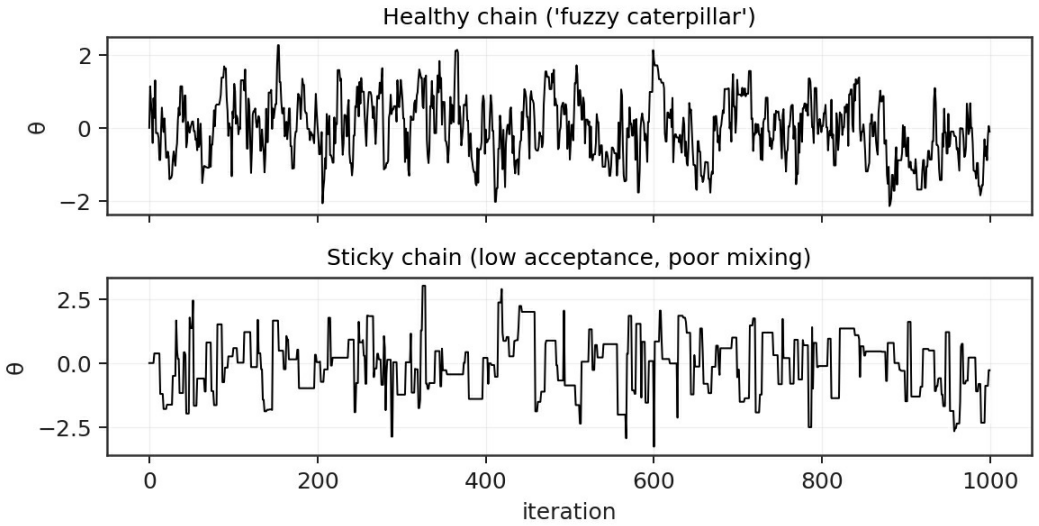


Figure 19.1: Healthy and pathological MCMC trace plots. A well-mixing chain (top) looks like a fuzzy caterpillar; a stuck chain (bottom) shows long correlations.

- R-hat (Gelman-Rubin): ratio of between-chain to within-chain variance for multiple chains run from dispersed initial conditions. Values above 1.01 suggest non-convergence.
- Effective sample size: dimension-by-dimension. Low ESS in a few coordinates often signals strong posterior correlation that the sampler cannot exploit.
- Divergent transitions in HMC: the leapfrog integrator can fail when the step size is too large for regions of high curvature. Stan and PyMC report divergences explicitly; ignoring them produces biased samples.

Multimodal posteriors

MCMC chains can become trapped in a single mode of a multimodal posterior, even when the chain is otherwise mixing well. The classic example is a mixture model with label-switching symmetry, where the posterior has K! equivalent modes corresponding to permutations of cluster labels. Methods to address multimodality include:

- Parallel tempering: run multiple chains at different temperatures and exchange states

across the temperature ladder.

- Replica exchange: structural variant of parallel tempering with explicit acceptance criterion.
- Sequential Monte Carlo: track an ensemble of particles through a sequence of distributions interpolating from prior to posterior.
- Reparameterization to break symmetry: enforce identifiability constraints (e.g., ordering of mixture means) so the posterior has a unique mode.

Scaling to big data

Vanilla MCMC requires evaluating the full likelihood at each step, expensive when N is large. Scalable variants include:

- Stochastic gradient HMC: noisy gradients from minibatches drive the dynamics, with appropriate friction to control variance.
- Stochastic gradient Langevin dynamics (SGLD): a one-step discretization of Langevin dynamics with injected noise, blending optimization and sampling.
- Subsampled Metropolis: estimate the acceptance ratio from a subsample of data, with a correction term to control bias.

All of these methods make trade-offs between exactness and scalability. SGLD, in particular, is biased, the stationary distribution is not exactly the posterior, but the bias is often small enough to be acceptable for predictive tasks where calibrated uncertainties are not critical.

Code example: Random-walk Metropolis

```
import numpy as np

def metropolis(log_pi, theta0, n_iter, step_size, rng=None):
    rng = rng or np.random.default_rng(0)
    d = theta0.size
    samples = np.empty((n_iter, d))
    log_pi_curr = log_pi(theta0)
    theta = theta0.copy()
    n_accept = 0
    for t in range(n_iter):
        proposal = theta + step_size * rng.standard_normal(d)
        log_pi_prop = log_pi(proposal)
        log_alpha = log_pi_prop - log_pi_curr
        if np.log(rng.uniform()) < log_alpha:
            theta = proposal
            log_pi_curr = log_pi_prop
            n_accept += 1
        samples[t] = theta
    return samples, n_accept / n_iter
```

Key Takeaways

- MCMC is the right tool when you need exact (in the limit) posterior samples and can afford the computation.
- Always use log-space arithmetic for acceptance ratios — naive probability ratios overflow quickly.
- Run multiple chains and check R-hat (potential scale reduction factor); never trust a single-

chain sampler.

- Divergent transitions or low effective sample size are diagnostic of sampler failure; do not ignore them.
- For high-dimensional smooth posteriors, HMC/NUTS is dramatically more efficient than random-walk Metropolis.

Exercises

- [Intermediate] Implement Metropolis-Hastings for a 2D Gaussian posterior with strong correlation. Plot the trajectory and compute the autocorrelation of each coordinate.
- [Intermediate] Tune the proposal scale of a random-walk Metropolis sampler to achieve approximately 0.234 acceptance. How does the optimal scale relate to the posterior covariance?
- [Intermediate] Implement Gibbs sampling for a Bayesian linear regression with Normal-Inverse-Gamma prior. Verify against an analytical posterior.
- [Intermediate] Run NUTS (via PyMC or Stan) on a hierarchical model and inspect divergent transitions. What reparameterization eliminates them?

Part VI

Stability, Generalization & Practical Systems

By this point in the book we have walked the full landscape of numerical methods for ML, floating-point foundations, optimization, linear algebra, deep learning, probabilistic computation. Each topic carried its own slice of practical wisdom, but the wisdom was largely organized around what we were computing. Part VI organizes wisdom around how computation behaves in practice: which patterns are stable, which patterns generalize, how to debug when things go wrong, and how to scale when things go right.

These four chapters are the most directly actionable in the book. The first, numerical stability in ML pipelines, contains a small set of techniques (stable softmax, log-sum-exp, careful scaling) that prevent the majority of NaN-and-infinity failures in production code. The second connects numerical conditioning to statistical generalization, surveying the surprising ways that small numerical effects shape large statistical behavior. The third is a debugging chapter: a systematic approach to diagnosing the failures that even careful code occasionally produces. The fourth covers scaling, vectorization, parallelism, GPU/TPU arithmetic; because the methods that win in modern ML are almost always the ones that map well onto hardware.

Chapter 20: Numerical Stability in ML Pipelines

Numerical stability is the discipline of computing the right answer even when the intermediate quantities span many orders of magnitude. In machine learning pipelines, the demand is acute: probabilities can be vanishingly small, exponentials can overflow with arguments of only a few hundred, and the same code must work with logits that range from negligible to enormous. This chapter collects the small set of techniques that, applied consistently, eliminate the majority of NaN-and-infinity failures in production ML code.

20.1 Stable Softmax Computation

The softmax function maps a vector of real numbers to a probability distribution:

$$softmax(x)_i = exp(x_i) / \Sigma_j \, exp(x_j)$$

The mathematical definition is impeccable. The naive implementation is broken. With logits of order 1000, entirely common in unnormalized model outputs, exp(1000) overflows to infinity in float32 (the float32 maximum is approximately 3.4×10^{38}, and exp(89) already exceeds it). The denominator becomes inf, the numerator becomes inf, and the ratio becomes NaN. Training silently dies.

The fix is the shift-invariance of softmax: subtracting any constant c from every logit leaves the distribution unchanged.

$$softmax(x)_i = exp(x_i - c) / \Sigma_j \, exp(x_j - c)$$

Choosing $c = \max_j x_j$ guarantees that the largest exponent is zero, the largest exponential is one, and the sum is at least one. Overflow is impossible. Underflow remains possible, logits much smaller than the maximum produce exponentials that round to zero, but those terms contribute negligibly to the sum and to the resulting probabilities, which is what we want.

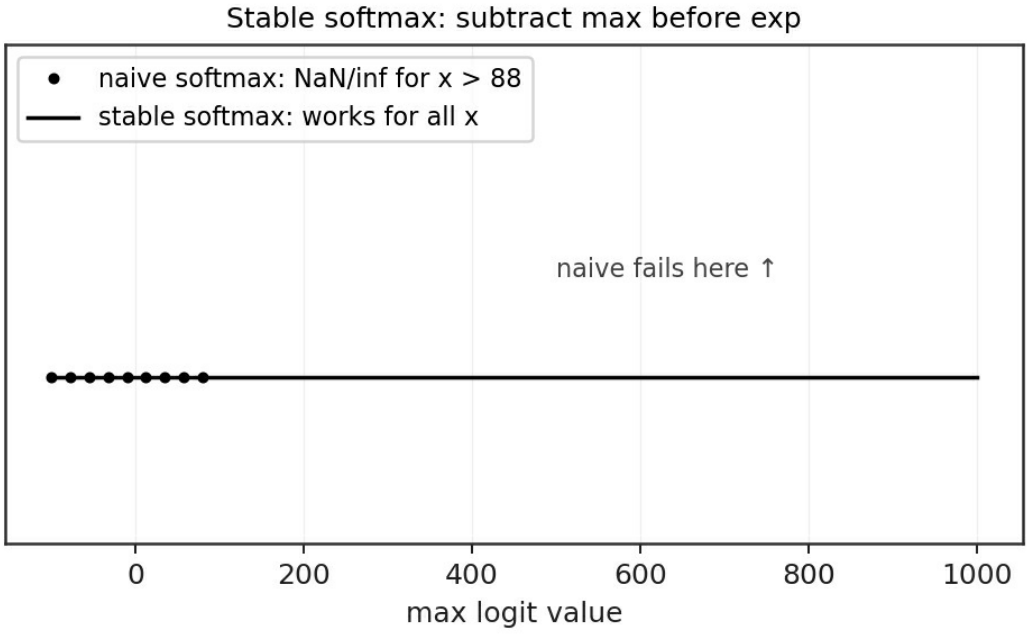


Figure 20.1: Stable vs. naive softmax. The naive form overflows for logits above about 89 in float32; the shift-by-max form is stable for any input.

Stable softmax: implementation

```
import numpy as np

def softmax_stable(x, axis=-1):
    x_max = np.max(x, axis=axis, keepdims=True)
    e = np.exp(x - x_max)
    return e / np.sum(e, axis=axis, keepdims=True)

# Sanity check: with extreme logits the naive version blows up.
logits = np.array([1000.0, 1001.0, 1002.0])
print(softmax_stable(logits))   # [0.09003, 0.24473, 0.66524]
print(np.exp(logits) / np.exp(logits).sum())   # nan
```

In PyTorch and TensorFlow, the built-in softmax is already stable. The same principle applies internally to every cross-entropy, attention, and routing layer. The stable form is so universal that production code rarely needs to think about it, but custom losses, custom layers, and any home-grown probabilistic code must implement it correctly.

20.2 The Log-Sum-Exp Trick

The companion to stable softmax is the log-sum-exp function:

$$LSE(x) = \log \Sigma_i \exp(x_i)$$

LSE shows up everywhere in ML: as the partition function of an exponential family, in the log-likelihood of a softmax classifier, in marginal computations of latent-variable models, in the path summation of CTC loss. The naive implementation, take exponentials, sum them, take the log, has the same overflow problem as softmax, with the same fix:

$$LSE(x) = c + \log \Sigma_i \exp(x_i - c), \quad c = max_i\, x_i$$

With this transformation, the largest term inside the log is one, and the result is computable for any range of inputs.

Stable log-sum-exp

```
import numpy as np

def logsumexp_stable(x, axis=-1):
    x_max = np.max(x, axis=axis, keepdims=True)
    out = x_max + np.log(np.sum(np.exp(x - x_max), axis=axis, keepdims=True))
    return np.squeeze(out, axis=axis)

# Stable log-softmax in one shot
def log_softmax_stable(x, axis=-1):
    return x - logsumexp_stable(x, axis=axis)[..., None]
```

Cross-entropy loss for softmax classification deserves particular attention. The mathematical form:

$$L = -\Sigma_i\, y_i \log softmax(x)_i = -x_\{true\} + LSE(x)$$

The right-hand expression, negative true logit plus log-sum-exp, is the only correct form to implement. Computing the softmax and then taking its log doubles the floating-point error and risks log(0) when probabilities underflow. Every modern framework has a 'cross_entropy_with_logits' or equivalent fused operator for exactly this reason; using the fused form is mandatory in custom training loops.

20.3 Avoiding Overflow and Underflow

Beyond softmax, several patterns recur in ML pipelines and have stable formulations that should always be preferred.

Stable sigmoid

The sigmoid function is sigmoid(x) = 1 / (1 + exp(−x)). For x ≲ − 89 in float32 the exponential overflows. The stable formulation branches:

$$\text{sigmoid}(x) = \exp(x) / (1 + \exp(x)) \quad \text{if } x < 0, \quad \text{sigmoid}(x) = 1 / (1 + \exp(-x)) \quad \text{if } x \geq 0$$

Stable log(1 + exp(x))

The softplus function appears in continuous relaxations of binary indicators, in flow-based generative models, and in many losses. The stable form is:

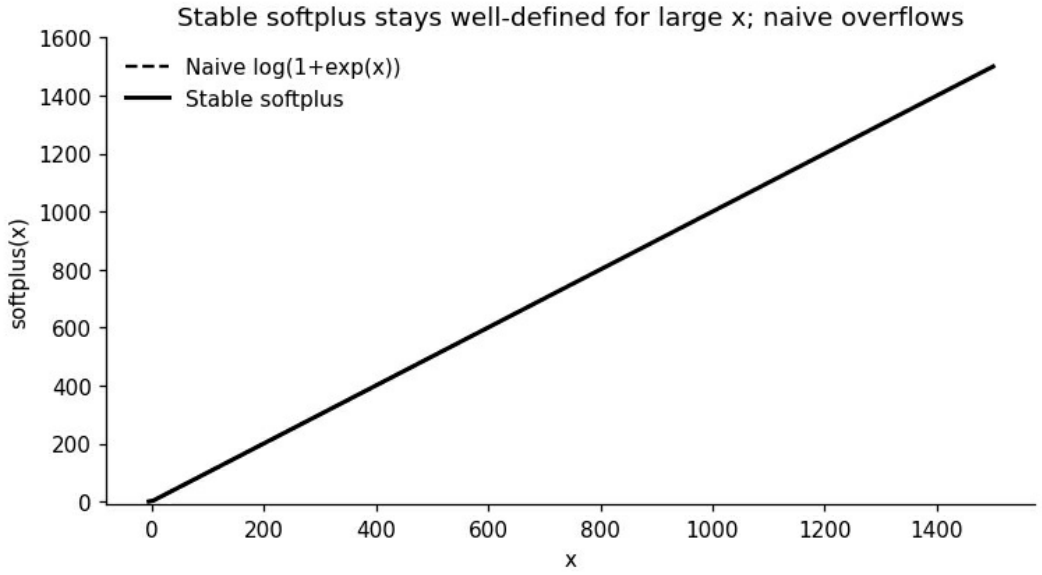


Figure 20.2: Stable softplus stays well-defined for large x.

$$\text{softplus}(x) = \log(1 + \exp(x)) = \max(x, 0) + \log(1 + \exp(-|x|))$$

This handles both large positive x (where the answer is approximately x) and large negative x (where the answer is approximately exp(x)) without intermediate overflow.

Stable sigmoid cross-entropy

Binary cross-entropy on logits should never compute the sigmoid first. The stable form is:

$$\text{BCE}(x, y) = \max(x, 0) - x \cdot y + \log(1 + \exp(-|x|))$$

Equivalent to −y·logσ(x) − (1−y)·log(1−σ(x)) but stable for x of any magnitude.

Catastrophic cancellation

When subtracting two nearly-equal floating-point numbers, the result loses significant digits proportional to the closeness. The classic example is computing variance as $E[X^2] - (E[X])^2$, which is mathematically identical to $E[(X - E[X])^2]$ but numerically catastrophic when X has large mean and small variance. Always use the centered form, ideally via Welford's online algorithm or a two-pass scheme.

```
import numpy as np

def variance_stable(x):
```

```
    """Two-pass: compute mean, then sum of squared deviations."""
    n = len(x)
    m = x.mean()
    return np.sum((x - m) ** 2) / (n - 1)

def variance_naive(x):
    """E[X^2] - (E[X])^2: catastrophic for large mean, small variance."""
    n = len(x)
    return np.sum(x ** 2) / n - x.mean() ** 2

# x has mean 1e8 and variance 1e-4
x = 1e8 + 1e-2 * np.random.randn(10000)
print(variance_stable(x))   # ~1e-4
print(variance_naive(x))    # nonsense, possibly negative
```

Mixed precision

Modern accelerators (NVIDIA H100, A100, TPUs) offer dramatic speedups for float16 and bfloat16 arithmetic. Mixed-precision training keeps weights and the master gradient in float32 but performs forward and backward passes in lower precision. The numerical hazards are real: float16 has only 3-4 decimal digits of precision and a much narrower dynamic range than float32. Bfloat16 trades precision for range, retaining float32's exponent but only 7 mantissa bits. Loss scaling, multiplying the loss by a large constant before backward and dividing by the same constant after, is the standard technique for keeping float16 gradients within representable range. Frameworks (PyTorch AMP, TensorFlow mixed_precision) handle this automatically when configured.

Key Takeaways

- A small handful of stable formulations prevent almost every NaN failure in ML training.
- Stable softmax (subtract max), log-sum-exp, BCE-with-logits, softplus, and two-pass variance are essential building blocks.
- When writing custom losses or layers, ask: does this match the math without overflowing or underflowing for any plausible input?
- log(probability) should never be computed directly — use log_softmax, log_sigmoid, or BCE-with-logits.
- Use these patterns religiously; the cost of stability discipline is zero, the cost of debugging without it is enormous.

Exercises

- [Intermediate] Implement stable softmax in float32 and verify that it gives correct probabilities for logits ranging from −1000 to +1000. Compare with the naive implementation.
- [Basic] Derive the stable form of softplus from the identity log(1+e^x) = max(x,0) + log(1+e^{−|x|}). Verify equivalence and stability.
- [Intermediate] Implement Welford's online variance algorithm. Verify that it gives the same result as the two-pass formulation for a stream of 10^6 samples.
- [Intermediate] Train a small classifier in float16 without loss scaling. Train the same classifier with loss scaling. Compare gradient norms by layer.

Chapter 21: Generalization vs Numerical Behavior

Generalization, the ability of a model to perform well on data it has not seen, is the goal of machine learning. It is also, surprisingly, intertwined with numerical behavior. The conditioning of a problem affects how well it generalizes; the noise inherent in stochastic optimization acts as a regularizer; the same models can exhibit qualitatively different behavior in classical and modern overparameterized regimes. This chapter explores the connections between numerical behavior and statistical generalization, with emphasis on the phenomena that defy classical intuition.

21.1 Overfitting and Conditioning

Classical statistics warns that overparameterized models overfit. With more parameters than data, a model can fit any training set perfectly, with no constraint on test error. The intuition is reflected in conditioning: the matrix X^TX of an overdetermined regression is ill-conditioned or singular, and the corresponding least-squares solution is unstable, small perturbations of the data produce large changes in the coefficients, the canonical signature of variance and overfitting.

Regularization combats both overfitting and ill-conditioning simultaneously. Adding λI to X^TX shifts every eigenvalue up by λ, eliminating the singularity and bounding the condition number. The same modification, ridge regression in statistics, reduces the variance of the estimator. Numerical conditioning and statistical regularization are two views of the same intervention.

Modern deep networks complicate this picture. They are dramatically overparameterized, with billions of parameters fit to millions of examples. Classical theory predicts catastrophic overfitting; in practice, generalization is often excellent. The reconciliation is not yet complete, but several mechanisms contribute:

- Implicit regularization from gradient methods. SGD does not find arbitrary minima of the training loss; it gravitates toward minima with particular geometric properties, typically flat minima with small parameter norms, that correlate with generalization.
- Architectural inductive biases. Convolutional networks encode translation invariance; transformers encode permutation invariance with positional encoding; both restrict the function class even when the parameter count is enormous.
- Double descent (Section 21.3) shows that the relationship between parameter count and test error is non-monotonic: error rises near the interpolation threshold, then falls again as parameters grow further.

21.2 Noise as Regularization

Stochastic gradient descent is a noisy approximation to gradient descent. The noise comes from the variance of mini-batch gradient estimates and decreases as the batch size grows. From the perspective of optimization, this noise is a nuisance, it slows convergence and introduces oscillation around the optimum. From the perspective of generalization, the noise is often beneficial.

Several mechanisms tie SGD noise to better generalization:

- Escape from sharp minima. Sharp minima, those with high curvature, are unstable under noise: small perturbations push the iterates out. Flat minima retain the iterates.

Empirically, flat minima generalize better, possibly because they are robust to small data perturbations.

- Noise as Tikhonov regularization. For convex losses, SGD with constant learning rate is equivalent to gradient descent on a smoothed version of the loss, a connection that quantifies the regularization implicit in batch-size choice.
- Anisotropic exploration. The covariance of mini-batch gradient noise is anisotropic: it is largest in directions where examples disagree and smallest in directions where they agree. This biases SGD toward parameter regions where examples reach consensus, a generalization-friendly property.

Dropout as noise injection

Dropout (Chapter 15) injects multiplicative Bernoulli noise into activations during training. From a numerical standpoint, this corresponds to training with a stochastically perturbed Hessian; from a statistical standpoint, it averages over an exponential family of subnetworks. Both views suggest that dropout regularizes by creating an ensemble effect implicit in a single network.

Data augmentation

Data augmentation, random crops, flips, color jitters, is functionally noise injection at the input level. It enlarges the effective training set and forces the network to be invariant to the augmentations. The numerical effect on gradients is to average over a continuous family of variations of each example, smoothing the loss landscape and reducing the variance of mini-batch updates.

21.3 The Double Descent Phenomenon

Classical learning theory predicts a U-shaped relationship between model capacity and test error: small models underfit, large models overfit, optimal capacity lies between. Empirical evidence in deep learning has overturned this picture. Test error follows a more complex pattern: it rises as expected until the interpolation threshold (where the model becomes large enough to fit training data exactly), peaks there, and then decreases again as the model grows further into the overparameterized regime. The phenomenon, named double descent, has been documented across model classes, linear regression, kernel methods, neural networks, and has rewritten the conventional wisdom about model capacity.

The interpolation threshold

At the interpolation threshold, the model has just enough capacity to fit training data exactly. The fitting solution is unique and is determined by exact interpolation, no slack remains. The conditioning of the problem is at its worst: the design matrix is exactly square, and small errors in the data produce large changes in the solution. This is the numerical signature of the test-error peak: ill-conditioning at the interpolation threshold causes the unique interpolating solution to depend sensitively on noise in the training labels.

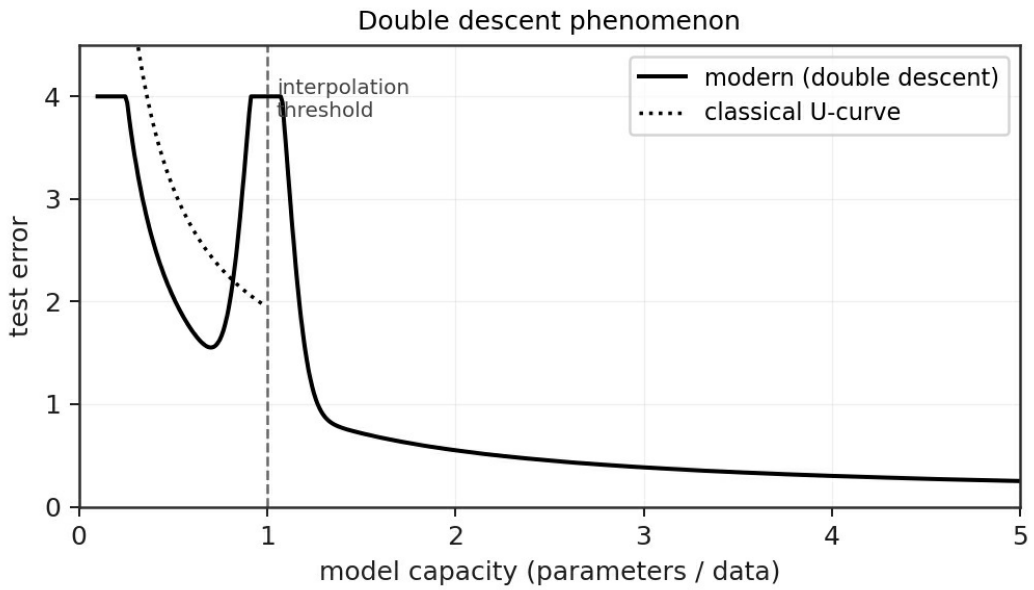


Figure 21.1: The double descent phenomenon. Test error rises near the interpolation threshold, then descends again as model capacity grows further.

Beyond interpolation

As capacity grows past the threshold, infinitely many interpolating solutions exist. The optimization procedure, gradient descent with appropriate initialization, selects a particular one. For linear regression, gradient descent from zero initialization converges to the minimum-norm interpolating solution: the solution with smallest ℓ^2 norm among all that fit training data exactly. This solution has good conditioning properties and often generalizes better than solutions found at the threshold.

For neural networks, the analog is more delicate. SGD-trained overparameterized networks find solutions with structural properties, small effective rank, low-magnitude weights, alignment with data structure, that correlate with generalization. The phenomenon is well documented but not yet fully theorized.

Practical implications

- Don't fear the interpolation threshold. Models trained near full capacity may behave worse than smaller or much larger models. If you observe an anomalous error spike, training a larger model may resolve it.
- Mind the implicit regularization. The numerical procedure (SGD with momentum, Adam, learning rate schedule) silently selects which interpolating solution gets chosen. Different optimizers can find solutions with markedly different generalization properties.
- Early stopping interacts with double descent. Stopping before convergence may keep the model in a regime where the second descent has not yet kicked in; running longer may improve generalization through implicit regularization.

Key Takeaways

• Numerical conditioning and statistical generalization are deeply linked.

• Regularization improves both: it shrinks effective dimension AND improves conditioning of

the Hessian.

- Ill-conditioning correlates with overfitting; well-conditioned models tend to generalize better.
- The double descent phenomenon makes the conditioning-generalization link visible at the interpolation threshold.
- SGD noise is not just an obstacle to convergence — it acts as a regularizer that shapes which solutions get found.

Exercises

- [Intermediate] Implement linear regression with increasing parameter count on a fixed small dataset. Plot training and test error. Identify the interpolation threshold and observe double descent.
- [Intermediate] Train two neural networks: one with a tiny batch size, one with a large batch size. Compare training trajectories and final test accuracy. Quantify the noise in mini-batch gradients.
- [Intermediate] Apply dropout to a small MLP. Compare the loss landscape (via Hessian eigenvalues) at the trained minimum, with and without dropout. Are the dropout minima flatter?
- [Basic] Compute the condition number of X^TX as the parameter dimension grows from $p \ll n$ to $p \approx n$ to $p \gg n$. At which regime is the condition number worst?

Chapter 22: Debugging Numerical Issues

Numerical bugs are the worst kind. Logical bugs produce wrong answers consistently; numerical bugs produce wrong answers stochastically, only sometimes, only at scale. A loss that diverges in epoch 47 but not in epoch 46. A gradient that is finite for 99.9% of batches and NaN for the rest. A model that trains fine on float32 but produces garbage in float16. These failures resist reproduction and demand systematic diagnostic methods.

This chapter develops a debugging methodology specific to numerical ML failures: how to detect, isolate, and fix the issues that traditional debugging (print statements, breakpoints, stack traces) cannot reach.

22.1 Diagnosing NaNs and Infinities

A NaN (Not a Number) appears when an undefined floating-point operation occurs: 0/0, $\infty - \infty$, $\log(-1)$, sqrt of a negative number. An infinity appears when a finite operation overflows. In an ML training loop, the typical signature is a loss value that prints as nan or inf after some number of iterations, often without warning. Once a NaN enters the network's weights or activations, every subsequent operation produces NaN, and training is dead.

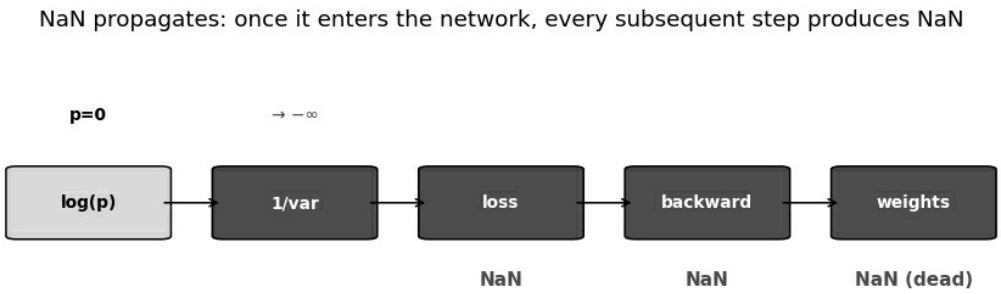


Figure 22.1: NaN propagates: once introduced, every subsequent operation produces NaN.

Locating the source

The first task is to determine where in the network the NaN originates. A standard tool is a forward-pass NaN check that examines every layer's output:

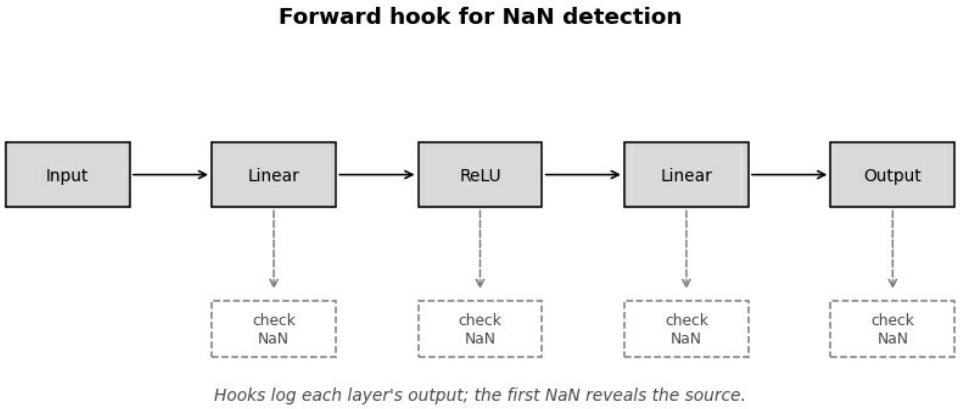


Figure 22.3: Forward hooks check each layer's output; the first NaN reveals the source layer.

```
def check_for_nans(model, x):
    """Forward pass and report which layer first produces NaN."""
    for name, module in model.named_modules():
        h = module(x) if not hasattr(module, 'forward') else None
    # cleaner: register forward hooks
    bad = []
    def hook(name):
        def fn(mod, inp, out):
            if torch.isnan(out).any() or torch.isinf(out).any():
                bad.append(name)
        return fn
    handles = []
    for name, m in model.named_modules():
        handles.append(m.register_forward_hook(hook(name)))
    try:
        _ = model(x)
    finally:
        for h in handles: h.remove()
    return bad
```

Common culprits, in rough order of frequency:

- log() of a value that has reached zero. Most often inside a non-stable loss function. Fix: use log_softmax, BCE-with-logits, or add a small ε inside the log.
- Division by a quantity that has become zero. Common in normalization layers when variance hits zero. Fix: add ε to the denominator (BatchNorm, LayerNorm, RMSNorm all do this).
- sqrt of a negative number. Cancellation in numerically computed variances; fix with the centered form.
- Exponential overflow. Logits or attention scores grow unboundedly. Fix: stable softmax/log-sum-exp; bound logits with normalization or weight decay.
- Gradient explosion. Magnitudes grow geometrically, often in early training of deep networks. Fix: gradient clipping, lower learning rate, better initialization, residual connections.

Detecting when, not just where

Many NaN failures are intermittent. A model that trains for 10,000 iterations and crashes on 10,001 has a numerical bug that only manifests on a specific input or after a specific accumulated state. Useful diagnostic patterns include:

- Periodic checkpoints with deterministic seeds. Reproduce the failure exactly, then bisect.
- Anomaly detection mode. PyTorch's torch.autograd.set_detect_anomaly(True) flags the operation that produces a NaN gradient, with a stack trace. Slow but invaluable for hard-to-find issues.
- Per-parameter gradient inspection. After backward, scan model.parameters() for any with NaN gradient. Identify the offending parameter and trace its forward path.

22.2 Gradient Checking

Gradient checking compares a hand-coded analytical gradient with a finite-difference approximation. It is the most reliable way to verify that a custom layer or loss function is

implemented correctly. The principle is the central-difference formula:

$$\partial f/\partial x_i \approx [f(x + \varepsilon e_i) - f(x - \varepsilon e_i)] / (2\varepsilon)$$

For each input dimension i, perturb the input by ±ε along that dimension, evaluate the function, and approximate the partial derivative. Compare against the gradient produced by your differentiable implementation.

Implementation

```
import numpy as np

def grad_check(f, grad_f, x, eps=1e-5, tol=1e-7):
    """Compare grad_f(x) with finite differences of f(x)."""
    g_analytical = grad_f(x)
    g_numerical = np.zeros_like(x)
    for i in range(x.size):
        x_plus = x.copy().ravel()
        x_minus = x.copy().ravel()
        x_plus[i] += eps
        x_minus[i] -= eps
        x_plus = x_plus.reshape(x.shape)
        x_minus = x_minus.reshape(x.shape)
        g_numerical.ravel()[i] = (f(x_plus) - f(x_minus)) / (2 * eps)
    rel_err = np.abs(g_analytical - g_numerical) / (np.abs(g_analytical) + np.abs(g_numerical) + 1e-12)
    return rel_err.max() < tol, rel_err
```

The choice of ε is critical. Too small, and rounding error dominates the difference. Too large, and truncation error of the central-difference formula dominates. The sweet spot for double precision is around $\varepsilon = 10^{-5}$ to 10^{-6}; for single precision, $\varepsilon = 10^{-3}$ may be necessary, with correspondingly looser tolerance.

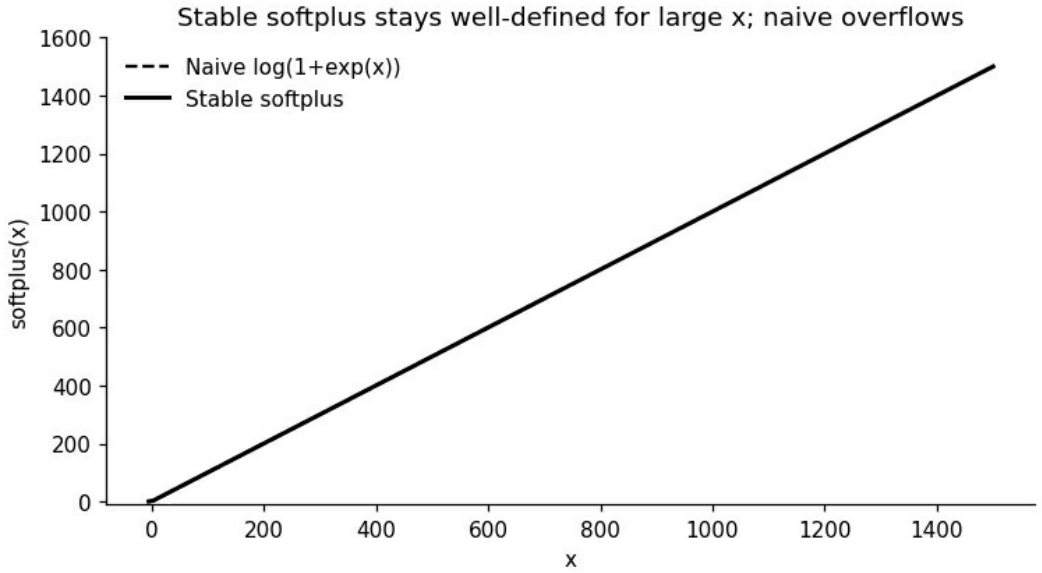


Figure 22.2: Optimal finite-difference step balances truncation and rounding errors.

Modern deep learning frameworks include built-in gradient checkers, torch.autograd.gradcheck, tf.test.compute_gradient, that handle the bookkeeping. Use them for any custom autograd function or hand-derived gradient.

22.3 Unit Tests for Numerical Code

Numerical code resists conventional unit testing. Floating-point arithmetic is approximate, so equality assertions are brittle; randomness in stochastic algorithms makes deterministic outputs hard to specify; even simple operations can have surprising edge cases. Effective testing of numerical code uses a different toolkit.

Property-based testing

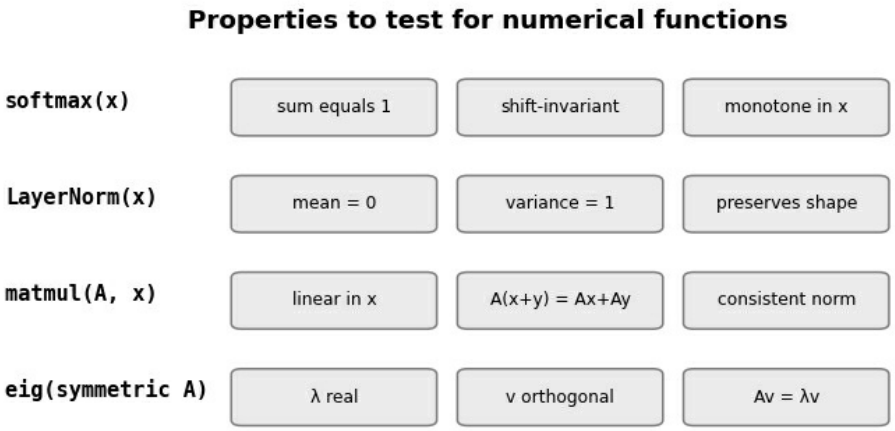


Figure 22.4: Properties to test for common numerical functions.

Rather than asserting specific output values, test properties that should hold for any input. Examples:

- softmax output sums to 1 for any input.
- LayerNorm output has zero mean and unit variance along the normalized axis.
- A symmetric matrix has real eigenvalues.
- Matrix-vector product is linear: f(x + y) = f(x) + f(y).

Property tests catch broad classes of bugs that point tests miss, and remain valid as the code evolves.

Numerical tolerances

Equality of floating-point results should always be tested with a tolerance, never with ==. The right tolerance depends on the precision and the operation. As a starting point:

- Within-precision absolute tolerance: 10^{-6} for float32, 10^{-13} for float64.
- Relative tolerance: same orders of magnitude, applied to (|a| + |b|).
- Use np.allclose or torch.allclose with explicit atol and rtol.

Edge cases to test

- Zero inputs.
- Negative inputs (especially for log, sqrt, division).
- Very large positive and negative inputs (test for overflow).
- Very small but nonzero inputs (test for underflow).

- Inputs at the boundaries of supported ranges (e.g., logits at ±infinity, probabilities at 0 and 1).

Regression tests

When a numerical bug is fixed, add a test that exercises the fixed pattern. Capture the input that triggered the bug, the expected output, and an appropriate tolerance. This prevents the same bug from recurring through later refactoring.

Key Takeaways

- Numerical bugs hide: they appear stochastically, often after long training, and standard debugging tools rarely catch them.
- The first defense is prevention: use stable formulations from Chapter 20 religiously.
- The second defense is detection: forward hooks, anomaly detection, and gradient checks reveal where bugs first appear.
- Property-based tests (mean=0, var=1 for LayerNorm; sum=1 for softmax) survive refactoring better than point tests.
- Adopt debugging discipline early; debugging at scale post-hoc is dramatically more expensive.

Exercises

- [Intermediate] Write a forward hook that logs the largest absolute activation in each layer. Train a small network and identify the layer where activations grow most rapidly.
- [Intermediate] Implement a custom autograd function for a non-trivial loss. Use gradcheck to verify the gradient. Test it on inputs of magnitude 10^{-6} and 10^{6}.
- [Intermediate] Write a property-based test for a softmax function. Test that outputs sum to 1, that the output is invariant to additive shift in the input, and that the output is monotonic in each input.
- [Intermediate] Reproduce a NaN failure in a small network by using a problematic loss formulation, then fix it by switching to the stable formulation. Verify both that the original fails and that the fix succeeds.

Chapter 23: Performance & Scaling

Numerical methods for ML are useful only if they can be executed at scale. A correctly stable softmax that runs at 10 examples per second is not viable for training a model on a billion examples. The methods that win in modern ML are almost always the methods that map well onto contemporary hardware: SIMD instructions, vector registers, GPU tensor cores, multi-device parallelism. This chapter surveys the techniques that turn algorithmic correctness into operational speed.

23.1 Vectorization Strategies

Modern CPUs and GPUs perform best on operations expressed as bulk arithmetic over arrays. The path from a slow Python loop to a fast vectorized routine is well-trodden, with two reliable principles.

Replace loops with array operations

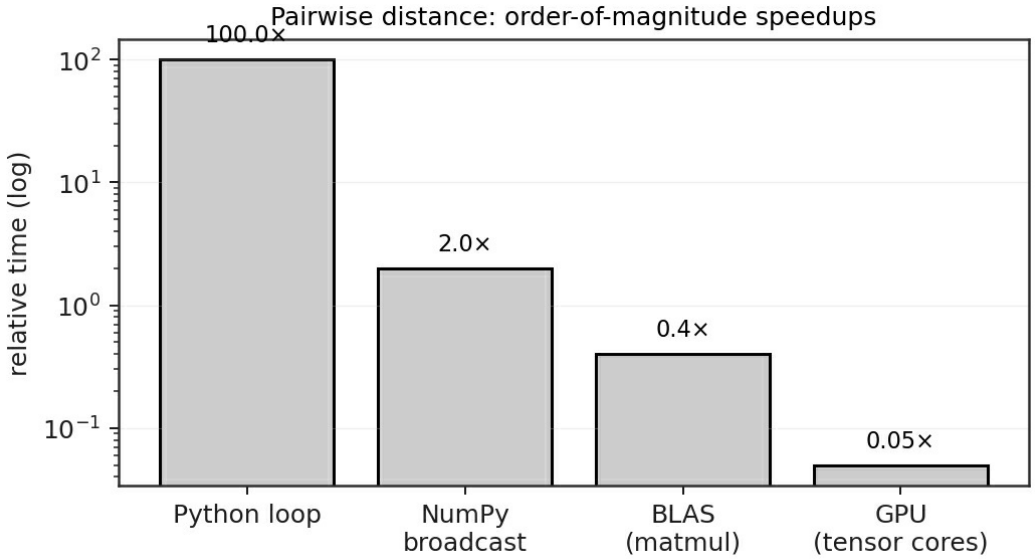


Figure 23.1: Speedup from vectorization. Python loops, NumPy operations, and BLAS-backed matrix products differ by orders of magnitude.

Python loops are slow because of interpreter overhead, dynamic typing, and lack of cache locality. NumPy operations on arrays are implemented in optimized C code and run at near-hardware speed. The transformation is mechanical for many algorithms:

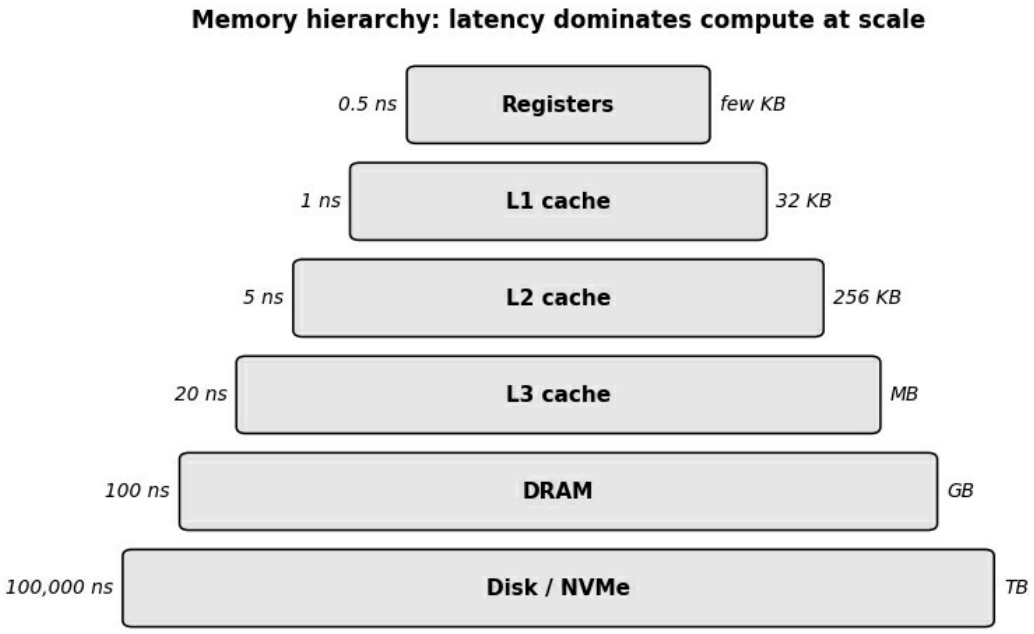


Figure 23.2: Memory hierarchy. Latency to fetch data dominates compute cost at scale.

```
import numpy as np

# Slow: Python loop
def slow_distance(X, y):
    out = np.zeros(X.shape[0])
    for i in range(X.shape[0]):
        for j in range(X.shape[1]):
            out[i] += (X[i, j] - y[j]) ** 2
    return np.sqrt(out)

# Fast: vectorized
def fast_distance(X, y):
    return np.sqrt(((X - y) ** 2).sum(axis=1))

# Order of magnitude faster, often 100x or more
```

Use BLAS-backed operations

Matrix multiplication, in particular, is implemented through highly optimized BLAS libraries (OpenBLAS, MKL, cuBLAS) that achieve near-peak hardware utilization. Express your computation as matrix-matrix or matrix-vector products whenever possible. A pairwise distance computation, for instance, can be written as:

$$\| x_i - y_j \|^2 = \| x_i \|^2 + \| y_j \|^2 - 2\langle x_i, y_j \rangle$$

All the cross-terms collapse into a single GEMM. The operation goes from $O(N^2D)$ Python-loop time to BLAS-level speed.

Broadcasting

Broadcasting allows operations between arrays of different shapes by implicitly expanding dimensions. Used carefully, it eliminates explicit reshaping and tiling. Used carelessly, it produces enormous intermediate arrays. A classic gotcha:

```
X = np.random.randn(10000, 100)   # 10000 rows, 100 features
y = np.random.randn(10000, 100)
```

```
# (X[:, None, :] - y[None, :, :]).sum() materializes a (10000, 10000, 100) array.
# That's 10⁴ GB.  Don't do this.
# Use the matmul identity above instead.
```

23.2 Parallel Computation

Vectorization exploits parallelism within a single core. Beyond that, there are several layers of additional parallelism to exploit.

Multi-threading

Most NumPy operations release the GIL and parallelize internally via OpenMP or MKL threads. Setting OMP_NUM_THREADS or MKL_NUM_THREADS controls the level. For trivially parallel work over many small problems, multiprocessing or joblib's Parallel can scale across cores. The Python-level overhead of process startup means parallelism is worthwhile only when each task takes more than a few milliseconds.

Data parallelism

In distributed training, the most common scheme is data parallelism: each worker holds a complete copy of the model and processes a different shard of the batch. After backward, gradients are averaged across workers via an all-reduce communication primitive, and weights are updated identically on every worker. The communication cost scales with the model size; for small models on fast networks it is negligible, for billion-parameter models it can dominate. NCCL (NVIDIA Collective Communications Library) and Horovod are standard implementations.

Model parallelism

For models too large to fit on a single device, the model itself is sharded across devices. Tensor parallelism splits individual layers (e.g., the linear layers of a transformer block) across devices, requiring all-gather and reduce-scatter operations within each forward and backward pass. Pipeline parallelism splits the model by depth, passing activations from device to device. Both incur substantial communication overhead and are deployed only when memory constraints force the issue.

Mixed strategies

Production training of large language models combines all three: data parallelism across nodes, tensor parallelism within a node, pipeline parallelism across stages. Frameworks like DeepSpeed, Megatron-LM, and FSDP (Fully Sharded Data Parallel) automate the orchestration.

23.3 GPU and TPU Numerical Considerations

Specialized accelerators are now the dominant platform for ML. They differ from CPUs in ways that affect both performance and numerical behavior.

Throughput vs. latency

GPUs are throughput-oriented: thousands of cores executing the same operation in lockstep. They reach peak performance only on large dense computations. Small matrix multiplications, irregular control flow, and tensor sizes that don't align with hardware tile boundaries leave performance on the table. Profile-guided tuning, batching, padding, kernel fusion, often produces 2× or more improvements.

Reduced-precision arithmetic

Modern GPUs and TPUs feature dedicated units for float16, bfloat16, and int8 arithmetic that run several times faster than float32. The trade-offs:

- Float16: 16 bits, 5-bit exponent (range ≈ $\pm 6 \times 10^4$), 10-bit mantissa (≈3-4 decimal digits). Fast and well-supported but vulnerable to overflow during training. Requires loss scaling.
- Bfloat16: 16 bits, 8-bit exponent (same range as float32), 7-bit mantissa (≈2-3 decimal digits). Wider range than float16 simplifies training. Standard for TPUs, increasingly for GPUs.
- Int8: 8-bit integer, used for inference quantization. Requires calibration and per-tensor scaling. Can produce 4× throughput improvements over float32 with acceptable accuracy loss.

Mixed-precision training keeps the master weights and optimizer state in float32 but performs forward and backward passes in float16/bfloat16. PyTorch AMP and TensorFlow's mixed_precision API automate this.

Determinism and reproducibility

GPU computations are not bit-for-bit reproducible by default. The non-associativity of floating-point addition means that summing N values in different orders produces slightly different results, and parallel reductions on GPUs use a different order each time. For exact reproducibility, frameworks offer deterministic modes (torch.use_deterministic_algorithms, tf.config.experimental.enable_op_determinism) at significant performance cost. Most production training accepts the small variability in exchange for full hardware speed.

Tensor cores and structured operations

NVIDIA's tensor cores accelerate matrix multiplications of specific shapes and precisions, 4×4×4 fp16 GEMMs in early generations, larger and more flexible in later ones. To activate tensor cores, tensor dimensions must be multiples of 8 (or 16 for some operations) and use a supported precision. Padding shapes to satisfy these constraints often produces 2×–4× speedups for free.

Key Takeaways

- Hardware-aware numerical methods are not optional in modern ML.
- Vectorize first, then exploit threads, then scale to multiple devices, then tune for the specific accelerator.
- Mixed precision training is standard; the 2x speedup is too large to ignore.
- Understand the precision-range trade-offs: float16 has narrower range than bfloat16; use loss scaling when needed.
- Memory bandwidth, not compute, is usually the bottleneck on modern GPUs.

Exercises

- [Intermediate] Implement pairwise Euclidean distances for a (1000, 100) array three ways: with explicit Python loops, with broadcasting, and with the matmul identity. Measure speed on each.

- [Intermediate] Train a small MLP in float32 and in float16 with AMP. Compare time per epoch and final accuracy. Measure the largest gradient magnitude across training.
- [Intermediate] Implement a simple all-reduce across two CPU processes using the multiprocessing module. Measure communication overhead as a function of tensor size.
- [Intermediate] Profile a model with PyTorch's profiler. Identify the operation that dominates wall time and explore whether it can be fused or replaced.

Part VII

Case Studies & Applications

Theory grounds practice; case studies test theory. Part VII assembles four end-to-end case studies that exercise the numerical methods of the preceding chapters in realistic settings. Each case study has been chosen to highlight a different cluster of techniques and a different style of failure mode. Together, they form a tour of what it actually looks like to apply numerical methods to a working ML system.

Chapter 24 builds a neural network from scratch, using nothing but NumPy and the algorithms of Parts II and IV. The goal is not to compete with PyTorch but to expose every numerical decision: initialization, normalization, gradient computation, optimizer state. Chapter 25 scales logistic regression beyond the single-machine regime, comparing optimizers and exposing the stability tricks that distinguish a textbook implementation from a production one. Chapter 26 applies PCA to a dataset large enough to break naive SVD, surveying the approximation methods that scale to modern data sizes. Chapter 27 collects real-world ML failures from the public record, each one a numerical lesson that cost an organization money, time, or reputation.

Chapter 24: Training a Neural Network from Scratch

Modern deep learning frameworks abstract away the numerical details that make training work. Auto-differentiation, default optimizers, automatic mixed precision, each is a layer of polish over algorithms whose stability was hard-won. Building a network from scratch in NumPy reveals what those layers are doing and why each design choice matters. This chapter walks through the implementation of a small classifier on MNIST, with attention to the numerical issues at every step.

24.1 Step-by-Step Implementation

Our network is a two-layer MLP: input → linear → ReLU → linear → softmax. We will train it with mini-batch SGD on a categorical cross-entropy loss, using float32 throughout. The pieces are familiar from earlier chapters; the assembly is the case study.

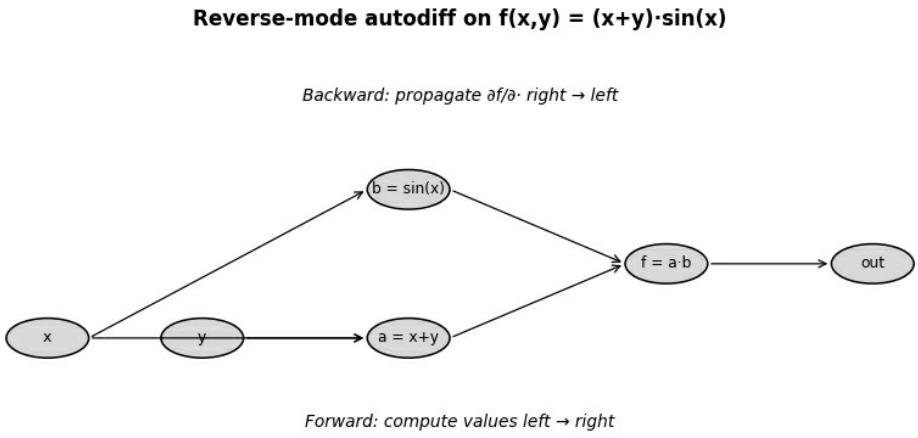


Figure 24.1: Two-layer MLP for MNIST: 784 → 128 (ReLU) → 10 logits → softmax.

Data and shapes

MNIST has 60,000 training images of size 28×28 with 10 classes. We flatten to vectors of length 784, normalize pixel values to [0, 1], and one-hot encode labels. The network has hidden width 128 and uses cross-entropy loss with logits.

```
import numpy as np

def load_mnist():
    X_train, y_train, X_test, y_test = ...
    X_train = X_train.reshape(-1, 784).astype(np.float32) / 255.0
    X_test = X_test.reshape(-1, 784).astype(np.float32) / 255.0
    return X_train, y_train, X_test, y_test

def one_hot(y, K=10):
    Y = np.zeros((y.size, K), dtype=np.float32)
    Y[np.arange(y.size), y] = 1.0
    return Y
```

Initialization

From Chapter 14: He initialization for ReLU networks. The first layer maps 784 → 128, so its

weights are drawn from N(0, 2/784); the second layer maps 128 → 10, with weights from N(0, 2/128). Biases are initialized to zero.

```
rng = np.random.default_rng(0)
W1 = rng.standard_normal((784, 128)).astype(np.float32) * np.sqrt(2.0 / 784)
b1 = np.zeros(128, dtype=np.float32)
W2 = rng.standard_normal((128, 10)).astype(np.float32) * np.sqrt(2.0 / 128)
b2 = np.zeros(10, dtype=np.float32)
```

Forward pass

The forward pass produces logits. We do not compute the softmax explicitly during training, the cross-entropy with logits combination is more stable, as discussed in Chapter 20.

```
def forward(X, W1, b1, W2, b2):
    z1 = X @ W1 + b1            # (N, 128)
    a1 = np.maximum(z1, 0.0)    # ReLU
    z2 = a1 @ W2 + b2           # (N, 10)  logits
    return z1, a1, z2

def cross_entropy_with_logits(logits, Y):
    m = logits.max(axis=1, keepdims=True)
    lse = m + np.log(np.exp(logits - m).sum(axis=1, keepdims=True))
    log_probs = logits - lse
    return -(Y * log_probs).sum(axis=1).mean()
```

Backward pass

The backward pass computes gradients of the loss with respect to W1, b1, W2, b2. Each step is a straightforward application of the chain rule, with attention to the numerical stability of each intermediate quantity.

```
def backward(X, Y, z1, a1, z2, W2):
    N = X.shape[0]
    m = z2.max(axis=1, keepdims=True)
    e = np.exp(z2 - m)
    p = e / e.sum(axis=1, keepdims=True)
    dz2 = (p - Y) / N
    dW2 = a1.T @ dz2
    db2 = dz2.sum(axis=0)
    da1 = dz2 @ W2.T
    dz1 = da1 * (z1 > 0)
    dW1 = X.T @ dz1
    db1 = dz1.sum(axis=0)
    return dW1, db1, dW2, db2
```

Training loop

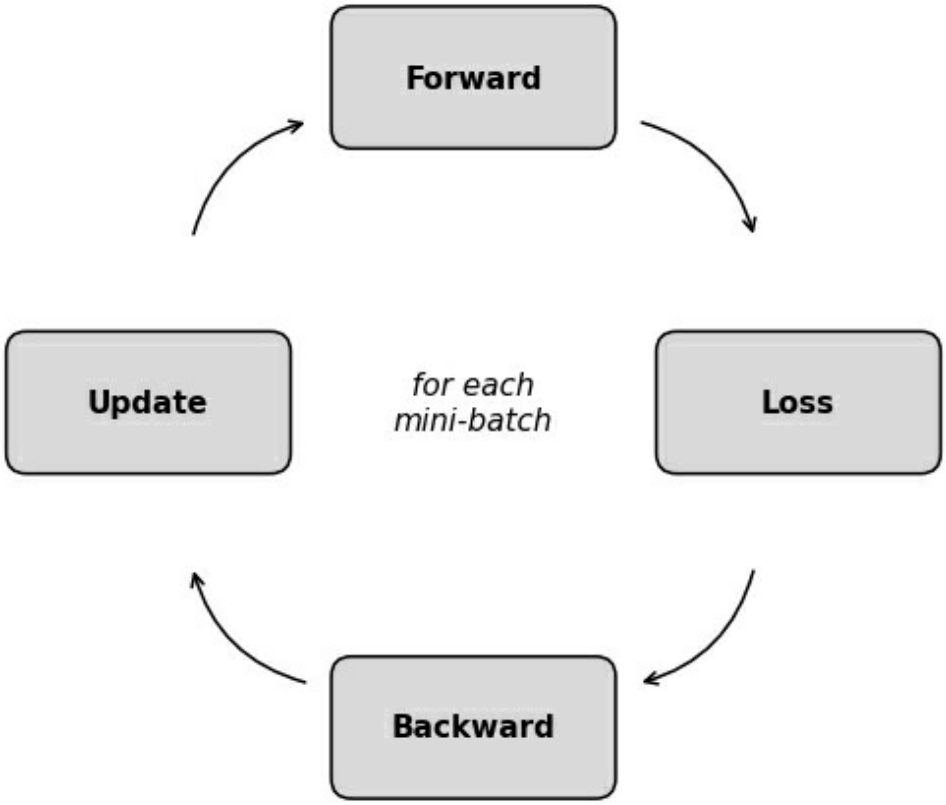


Figure 24.2: The training loop: forward → loss → backward → update.

```
lr = 0.1
batch_size = 128
n_epochs = 10

for epoch in range(n_epochs):
    perm = rng.permutation(X_train.shape[0])
    losses = []
    for i in range(0, X_train.shape[0], batch_size):
        idx = perm[i:i+batch_size]
        X_batch = X_train[idx]
        Y_batch = one_hot(y_train[idx])
        z1, a1, z2 = forward(X_batch, W1, b1, W2, b2)
        loss = cross_entropy_with_logits(z2, Y_batch)
        losses.append(loss)
        dW1, db1, dW2, db2 = backward(X_batch, Y_batch, z1, a1, z2, W2)
        W1 -= lr * dW1; b1 -= lr * db1
        W2 -= lr * dW2; b2 -= lr * db2
    print(f"Epoch {epoch}: train loss {np.mean(losses):.4f}")
```

24.2 Stability Checks

With the implementation in hand, we run a battery of sanity checks before trusting any training run.

Initial loss

With 10 classes and uniform softmax (all logits equal), the cross-entropy is log 10 ≈ 2.302. The first batch's loss should be near this value. A wildly different initial loss, say, 6 or 0.01, indicates a bug. (For 6: probably a bad scaling in initialization. For 0.01: probably a bug in the loss; the network cannot have learned in zero iterations.)

Gradient check on a small instance

Run the gradient checker from Chapter 22 on a 2-example batch. Compare every parameter's analytical gradient with the central-difference estimate. Tolerances at 10^{-4} for float32 are appropriate. If any parameter has relative error above 10^{-3}, there is a bug.

Train on a tiny dataset

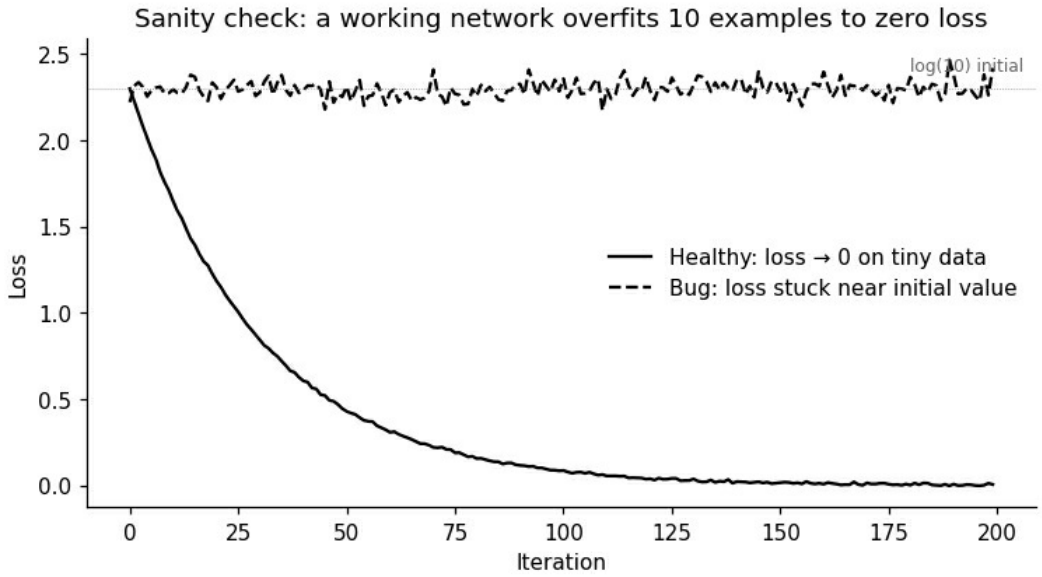


Figure 24.3: Sanity check: a working network drives loss to zero on a tiny dataset.

The single most diagnostic experiment: take 10 examples (one per class). Train the network. Loss should reach near zero in a few hundred steps. If it doesn't, the network has a structural problem (insufficient capacity, broken backward) that has nothing to do with the dataset. Successfully overfitting a small dataset is necessary, though not sufficient, evidence that the implementation works.

Watch the activation statistics

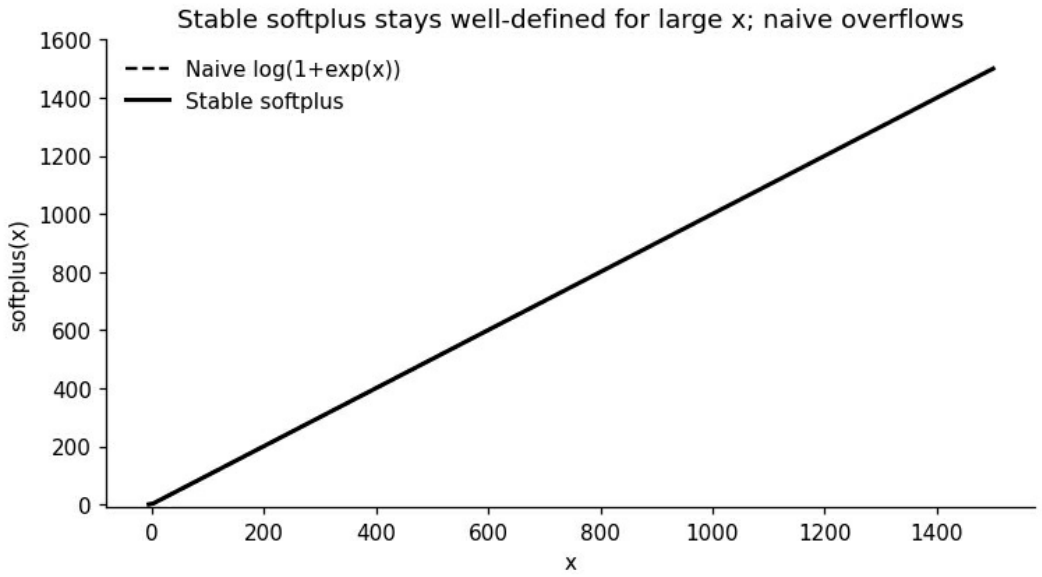


Figure 24.4: Three regimes of activation statistics across depth.

After 100 steps and after 1000 steps, log the mean and standard deviation of z1, a1, z2, and the gradient magnitudes by layer. Healthy patterns: z1 has mean near zero and standard deviation near 1; a1 has half its values at zero (ReLU saturation); z2 has spread proportional to logit scale; gradient magnitudes are similar across layers. Unhealthy patterns: standard deviations growing or shrinking by orders of magnitude across layers, or gradient norms differing by more than 10× between layers.

24.3 Performance Tuning

With a working network, several tuning levers improve the result.

Learning rate

0.1 is aggressive for an SGD-without-momentum implementation. Run a learning-rate range test (Chapter 5): start at 10^{-6}, increase geometrically per batch, plot loss against rate. The fastest decrease typically begins one decade below where the loss starts diverging. For this network and dataset, the sweet spot is usually in the 0.01–0.1 range.

Momentum

Replacing pure SGD with SGD-momentum ($\beta = 0.9$) typically improves both convergence speed and final accuracy. Implementation:

```
vW1 = np.zeros_like(W1)
# ... and similarly for b1, W2, b2

vW1 = beta * vW1 + dW1
W1 -= lr * vW1
```

Mini-batch size

Doubling the batch size from 32 to 128 to 512 progressively reduces gradient variance and may allow a higher learning rate. Beyond a problem-specific threshold, doubling the batch size halves the wall time but does not improve generalization, large-batch training requires careful learning-rate scaling and is its own subdiscipline.

Weight decay

Adding a small L2 penalty ($\lambda \approx 10^{-4}$) shrinks weights toward zero each step. This often improves test accuracy by 0.5–1 percentage point with no other changes. The update is $W \leftarrow W - lr \cdot (\nabla + \lambda W)$.

Expected results

A correctly implemented two-layer MLP on MNIST should reach about 98% test accuracy with the described setup. Failure to reach 95% suggests an implementation bug; failure to reach 97% suggests inadequate hyperparameter tuning. The numerical fingerprint of the trained network includes parameter norms in the range 0.1–1, gradient norms decaying smoothly through training, and softmax outputs that are confident on most test examples and uncertain on the ambiguous ones.

Key Takeaways

- Building a network from scratch turns abstract principles into concrete operations.
- Initialization, stable loss, gradient checking, and overfitting a tiny dataset are non-negotiable sanity checks.
- If gradient check fails on float64, the backward pass is wrong; if it passes, the network is implementable.
- If you cannot overfit 10 examples to zero loss, the network has a structural bug — do not proceed to scale.
- Once these checks pass, hyperparameter tuning is rapid; before they pass, no amount of tuning will help.

Exercises

- [Advanced] Extend the network to three hidden layers. Compare training without and with He initialization. What happens to gradient magnitudes in the deepest layer?
- [Intermediate] Replace ReLU with Tanh and rerun. Compute gradient magnitudes by layer. Does vanishing gradient appear?
- [Intermediate] Implement Adam in place of SGD. Compare convergence behavior. Does Adam reach a higher or lower test accuracy at the same number of epochs?
- [Intermediate] Add dropout with rate 0.5 between the hidden layer and the output. Compare test accuracy. Does dropout help on MNIST at this scale?

Chapter 25: Logistic Regression at Scale

Logistic regression is the simplest nontrivial classifier and a useful workhorse. At small scale it is solved cleanly by Newton's method. At medium scale, by L-BFGS. At large scale, millions of features and examples, the appropriate algorithm depends on the structure of the data, the sparsity, the available memory, and the computational budget. This chapter compares the major optimization choices for logistic regression and surfaces the numerical tricks that distinguish a textbook implementation from a robust large-scale one.

25.1 The Optimization Choice

Logistic regression minimizes the negative log-likelihood:

$$L(w) = \Sigma_i \, [-y_i \cdot w^T x_i + \log(1 + \exp(w^T x_i))] + (\lambda/2) \, \| w \|^2$$

The function is convex, twice differentiable, and well-conditioned for small λ. The Hessian:

$$H = X^T D X + \lambda I, \quad D_{ii} = p_i(1 - p_i)$$

is positive definite, so any reasonable optimizer converges; the choice is about computational cost.

Newton's method

For small problems ($n, p \lesssim 10^4$), Newton's method converges in a handful of iterations. Each iteration solves the system $H \Delta w = -\nabla L$, costing $O(p^3)$ for the factorization and $O(np^2)$ for forming H. Total cost: $O((n + p)p^2)$ per iteration. Wins on accuracy, loses on memory beyond moderate p.

L-BFGS

For p in the tens of thousands, L-BFGS approximates the Hessian using only gradient history (Chapter 7). Memory cost is $O(mp)$ for memory size $m \approx 10$. Per-iteration cost is dominated by the gradient: $O(np)$. For dense problems, L-BFGS is the default. For convex problems with up to a few million parameters, scikit-learn's logistic regression uses L-BFGS by default.

SGD and stochastic methods

For very large datasets where even one full gradient is expensive, stochastic methods process minibatches:

- SGD: simple, requires learning rate tuning, slow convergence near the optimum but cheap iterations.
- SAG, SVRG: variance-reduced stochastic methods that achieve linear convergence on convex problems with the per-iteration cost of SGD.
- SDCA: dual coordinate ascent, particularly effective for sparse high-dimensional problems.

On a billion-example dataset with sparse features (e.g., text categorization), SDCA or coordinate descent typically beats L-BFGS in wall time, despite L-BFGS's superior per-iteration progress.

Parallel and distributed

For genuinely massive data, partition the dataset across machines and use one of:

- Synchronous mini-batch SGD with all-reduce, as in deep learning.

- Distributed L-BFGS, where each worker computes a partial gradient and a coordinator runs the L-BFGS update.
- ADMM (alternating direction method of multipliers), which decouples the per-machine subproblems and aggregates only the consensus weights.

25.2 Numerical Stability Tricks

Even a problem as simple as logistic regression has stability gotchas at scale.

Stable log(1 + exp(x))

The cross-entropy term $\log(1 + \exp(w^T x))$ overflows for large positive arguments and underflows for large negative ones. The stable form (Chapter 20):

$$softplus(z) = max(z, 0) + log(1 + exp(-|z|))$$

Equivalent mathematically, stable for any z. Every production logistic regression code uses this form internally.

Sparse data structures

Text and click data are typically sparse: most features are zero for any given example. Storing X as a CSR sparse matrix and using sparse matrix-vector products reduces memory by orders of magnitude and cuts per-iteration cost from O(np) to O(nnz), where nnz is the number of nonzeros. The gradient and Hessian-vector products have natural sparse implementations.

Feature scaling

Logistic regression is scale-sensitive: features with large magnitudes get small coefficients and vice versa, but only after the optimizer has worked through the imbalance. Standardizing features (zero mean, unit variance) accelerates convergence dramatically. For sparse data, mean-subtraction destroys sparsity, instead, scale by standard deviation only, leaving the mean alone.

Regularization path

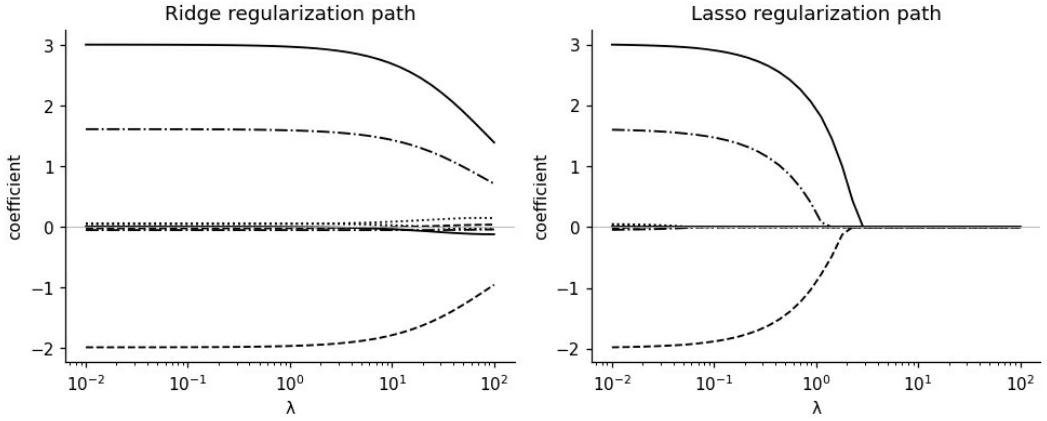


Figure 25.1: Regularization paths. Ridge (left) shrinks coefficients smoothly toward zero. Lasso (right) sets coefficients exactly to zero at thresholds.

The optimal regularization λ is rarely known a priori; cross-validation requires solving the problem for many λ values. Warm-starting, using the solution at one λ as the starting point for an adjacent λ, dramatically reduces total work. Coordinate descent and proximal methods are particularly amenable to warm starting.

Class imbalance

When one class is rare (1% positive, 99% negative), uniform sampling gives most of the gradient signal to the majority class. Stratified mini-batches, class weights, and focal loss are alternatives. From a numerical standpoint, the imbalance shows up as a bad initial condition number: $D_{ii} = p_i(1-p_i)$ is small for confident predictions, and a model that confidently predicts the majority class for everything has $D \approx 0$ and a near-singular Hessian.

25.3 A Worked Comparison

To make the comparisons concrete, consider a problem with $n = 10^6$ examples and $p = 10^4$ features:

- Dense L-BFGS: ~30 seconds per epoch on a single machine, converges in 10–20 epochs. Total: a few minutes.
- Mini-batch SGD with batch 1024: similar or better wall time per epoch but typically needs more epochs and careful learning-rate tuning. Wins when data is too large to fit in memory.
- Coordinate descent: very fast for sparse data, particularly with L1 regularization where it produces sparse solutions naturally. Glmnet (R) and sklearn's saga solver use this.
- Distributed L-BFGS across 10 machines: linear speedup if the gradient computation dominates, with overhead from communication.

There is no universally best choice. Profile the operation that dominates wall time, forward pass, gradient sum, line search, communication, and target that for optimization.

Key Takeaways

- Logistic regression is convex and well-behaved, but scaling requires deliberate attention to numerical structure.
- Use stable softplus, exploit sparsity, standardize features, and warm-start across the regularization path.
- Choose the optimizer based on data shape: L-BFGS for dense moderate-scale, SGD for very large data, coordinate descent for sparse.
- For cross-machine training, distributed L-BFGS or ADMM are appropriate choices.
- L1-regularized logistic regression gives a sparse feature selection essentially for free.

Exercises

- [Intermediate] Implement logistic regression with both naive and stable softplus. Try logits of magnitude 100 and confirm the naive version fails.
- [Intermediate] Compare L-BFGS, SGD, and coordinate descent on a sparse text classification dataset (e.g., 20 Newsgroups). Report wall time and final accuracy.
- [Intermediate] Trace the regularization path of an L1-regularized logistic regression by warm-starting from large λ to small. Plot coefficient magnitudes versus λ.
- [Intermediate] Implement class weights for an imbalanced dataset. Compare with uniform weighting and with explicit oversampling. Which behaves most stably?

Chapter 26: PCA on Large Dataset

Principal component analysis is straightforward at small scale: center the data, compute the SVD of the centered matrix, take the top components. At large scale, the algorithm breaks. SVD on a million-by-thousand matrix consumes hours of compute and gigabytes of memory; on a million-by-million matrix, it is infeasible. This chapter walks through the techniques that scale PCA to modern data sizes, truncated SVD, randomized projections, online algorithms, and the numerical considerations that distinguish each.

26.1 Efficient SVD

The full SVD of an m×n matrix costs $O(\min(m^2n, mn^2))$ and produces all min(m, n) singular values and vectors. For PCA we typically want only the top k components, with $k \ll \min(m, n)$. Truncated SVD algorithms compute only what we need, with dramatically lower cost.

ARPACK and Lanczos iteration

The Lanczos algorithm, implemented in ARPACK and exposed by SciPy as scipy.sparse.linalg.svds, computes the top k singular values and vectors via Krylov subspace methods. Each iteration costs $O((m + n)k)$ for matrix-vector products plus orthogonalization. Total cost is dominated by the matrix-vector products, of which we need O(k) per iteration and O(k) to a few hundred iterations to converge. For $k \ll \min(m, n)$, the savings over full SVD are substantial.

The numerical caveat is reorthogonalization. The Lanczos vectors lose orthogonality due to floating-point error, a phenomenon documented for decades. Without correction, ghost eigenvalues appear: spurious copies of the true ones. ARPACK uses implicit restart and selective reorthogonalization to handle this; doing it correctly by hand is delicate.

Randomized SVD

Randomized SVD, popularized by Halko, Martinsson, and Tropp, takes a different approach. To find a rank-k approximation:

- Draw a random Gaussian matrix $\Omega \in R^{\wedge}(n \times (k + p))$, where p is a small oversampling parameter (typically $p \approx 5–10$).
- Form $Y = A\Omega$. This is an $m \times (k + p)$ matrix whose columns span an approximation to the dominant subspace of A.
- Orthonormalize the columns of Y via QR to get Q.
- Form $B = Q^TA$. This is a $(k + p) \times n$ matrix, small if k is small.
- Compute the SVD of B: $B = \bar{U}\Sigma V^T$.
- Recover the SVD of A: $U = Q\bar{U}$, with Σ and V as in the SVD of B.

The total cost is two matrix-matrix products with A plus a small SVD: $O(mn(k + p))$ plus $O((k + p)^2(m + n))$. For k = 10 and $m = n = 10^6$, this is far cheaper than the full SVD's $O(\min(m^2n, mn^2))$. Better still, the algorithm parallelizes trivially, the matrix-matrix products are GEMMs, and the small SVD is done locally.

Power iteration corrections

Randomized SVD with a small number of iterations may have noticeable error in the top singular

vectors when the singular values decay slowly. The remedy is power iteration: replace A with $(AA^T)^q A$ for q = 1 or 2. This concentrates spectral mass on the dominant singular values and dramatically improves accuracy at modest cost. q = 2 is sufficient for most ML applications.

```
import numpy as np
from numpy.linalg import qr, svd

def randomized_svd(A, k, p=10, q=2, rng=None):
    rng = rng or np.random.default_rng(0)
    m, n = A.shape
    Omega = rng.standard_normal((n, k + p))
    Y = A @ Omega
    for _ in range(q):
        Y = A @ (A.T @ Y)
    Q, _ = qr(Y, mode='reduced')
    B = Q.T @ A
    Ub, S, Vt = svd(B, full_matrices=False)
    U = Q @ Ub
    return U[:, :k], S[:k], Vt[:k, :]
```

26.2 Approximation Methods

Beyond SVD-based approaches, several techniques approximate PCA more aggressively, trading accuracy for further speed.

Online and incremental PCA

When data arrives in a stream or is too large to hold in memory, online PCA processes one example (or one mini-batch) at a time. The incremental PCA algorithm maintains a running mean and a low-rank covariance approximation, updating both as new data arrives. Each update is $O(pk^2)$ for batch size p and rank k, independent of the total dataset size. scikit-learn's IncrementalPCA implements this; the algorithm scales to datasets of any size, at the cost of accuracy that depends on batch size and order of presentation.

Stochastic methods for PCA

Stochastic gradient methods minimize the PCA objective directly. Oja's rule, the simplest such method, updates a candidate eigenvector w with $w \leftarrow (w + \eta x x^T w) / \|\ldots\|$ for each new example. Block versions track multiple eigenvectors in parallel. Each update is O(pk), and the methods converge in expectation to the true principal directions, with variance that scales like $1/\sqrt{N}$ (Chapter 17).

Sketching

Sketching methods compress the data matrix to a much smaller surrogate that approximately preserves spectral structure. Frequent sketching techniques include:

- Gaussian sketches: multiply A by a random Gaussian matrix to reduce one dimension, then take SVD of the smaller result.
- FJLT (Fast Johnson-Lindenstrauss Transform): use structured matrices (subsampled Hadamard transforms) for faster sketching.
- CountSketch: use sparse random matrices for very large m, n.

All of these sketching algorithms come with tail bounds on the approximation error, you can choose the sketch size to achieve a target accuracy with high probability. Production data

analytics pipelines (especially in distributed settings) rely on sketching for both PCA and related tasks (regression, clustering, low-rank matrix recovery).

26.3 Worked Example: Image Compression via PCA

To make the techniques concrete, consider PCA on the CIFAR-10 dataset, 60,000 images of 32×32×3 = 3072 features each. The data matrix is 60000 × 3072: small enough that full SVD is tractable but already large enough to benefit from acceleration.

Centering and scaling

Center the data by subtracting the per-feature mean. Optionally scale by per-feature standard deviation, though for image pixels this is not strictly necessary. Centering is critical: PCA on uncentered data finds the leading direction in absolute coordinates, which is dominated by the mean. Catastrophic cancellation (Chapter 20) is a risk for two-pass centering when the mean is large; for image pixel values (0 to 255 or 0 to 1), the cancellation is mild.

Choosing k

The standard heuristic is to choose k so that the top k components explain a target fraction (e.g., 95%) of the total variance:

$$k^* = \min \{ k : \Sigma_{i=1}^{k} \sigma_i^2 / \Sigma_i \sigma_i^2 \geq 0.95 \}$$

For natural images, the cumulative variance climbs steeply for the first dozen components, then slowly tapers. Reaching 95% may require k = 100 to a few hundred.

Reconstruction quality

After projecting onto the top k components and back-projecting, the reconstruction error is the residual variance: $\Sigma_{i=k+1} \sigma_i^2$. For images, qualitative inspection complements the numerical metric, the reconstruction with k = 50 may look acceptable for downstream classification but introduce visible blurring.

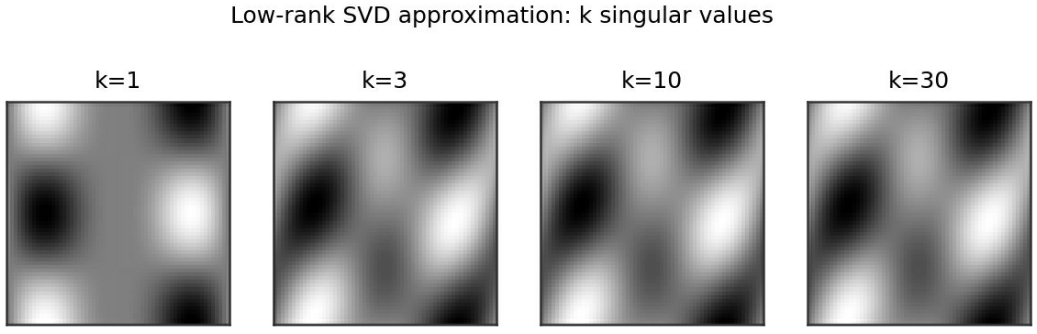


Figure 26.1: Low-rank approximation via truncated SVD. Reconstruction quality versus retained rank k.

Performance

On a typical workstation with NumPy and OpenBLAS:

- Full SVD of 60000 × 3072: a few minutes, several gigabytes of intermediate memory.
- Truncated SVD (svds, k = 100): about 30 seconds, much less memory.
- Randomized SVD (k = 100, q = 2): about 5 seconds, even less memory.
- Incremental PCA in batches of 1000, k = 100: about 20 seconds, memory bounded by

batch size.

For larger datasets, say, 10^8 examples, only the incremental and randomized methods remain tractable.

Key Takeaways

- Full SVD does not scale; modern PCA uses truncated, randomized, or incremental methods.
- Randomized SVD with a few power iterations is often the fastest path to a high-quality rank-k decomposition.
- Incremental PCA scales to streaming or memory-constrained data at the cost of approximation.
- Always center the data before PCA; centering errors are the most common source of misleading principal components.
- Always check that cumulative explained variance reaches your target and inspect the components for interpretability.

Exercises

- [Advanced] Implement randomized SVD with q = 0, 1, 2 power iterations. Compare singular value accuracy on a synthetic matrix with known spectrum.
- [Intermediate] Apply both full SVD and randomized SVD to CIFAR-10. Verify that the top-100 components match within numerical tolerance. Time each.
- [Intermediate] Run incremental PCA in three different batch orderings on the same data. Compare the recovered components. How much do they vary?
- [Advanced] For a sparse text-document matrix, compare scipy.sparse.linalg.svds (Lanczos) with a randomized variant. Which is faster for k = 50?

Chapter 27: Real-World ML Failures

The cost of a numerical bug is rarely measured in the abstract. It is measured in retraining runs that take days, in production models that silently degrade, in research papers that retract conclusions, and occasionally in real-world consequences for people the model affects. This chapter collects representative case studies of numerical instability in deployed or widely-used ML systems, drawn from the public record. Each one is a teachable lesson, about what failed, what was missed, and what defensive practice would have caught it earlier.

Names of organizations are kept generic where the failure is illustrative rather than singular. The lessons are universal.

27.1 The Silent Loss-Function Bug

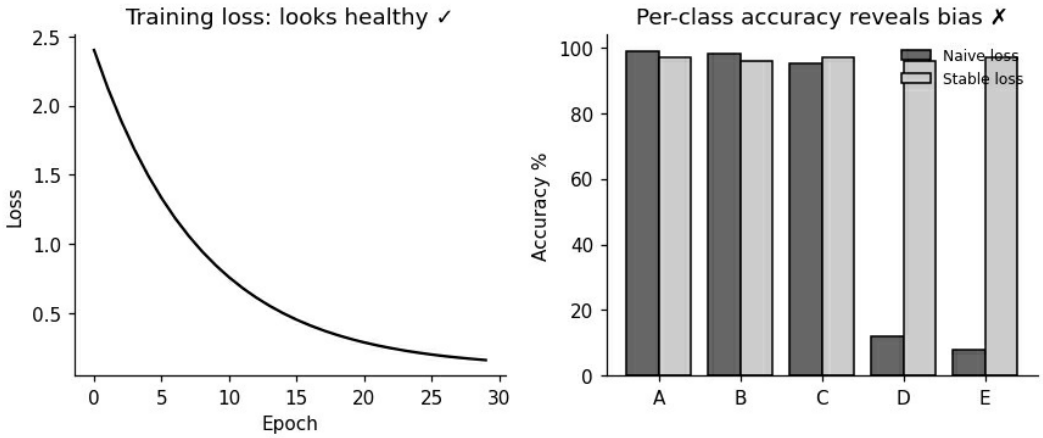


Figure 27.1: Silent loss-function bugs. Training loss looks healthy but per-class accuracy reveals failure.

A computer-vision team trained a state-of-the-art segmentation model for several weeks on a multi-GPU cluster. The trained model performed well on internal validation but inexplicably worse on a held-out evaluation. Investigation traced the discrepancy to a custom loss function that computed log of probabilities directly, with no ε inside the log. For most batches, the predicted probabilities were bounded away from zero and the loss was correct. For a small fraction of difficult examples, predicted probabilities reached the underflow threshold for float32, producing log of zero, negative infinity, in the loss. The framework's autograd silently clamped the gradient at the largest representable value, producing enormous but finite updates that subtly biased the trained weights toward those examples.

The lesson: never compute log of a probability directly. Use log_softmax, BCE-with-logits, or other stable formulations from Chapter 20. The bug had no visible signature, no NaN, no warning, no abnormal-looking metric, yet biased the model in a way that took weeks to diagnose.

27.2 The Mixed-Precision Pitfall

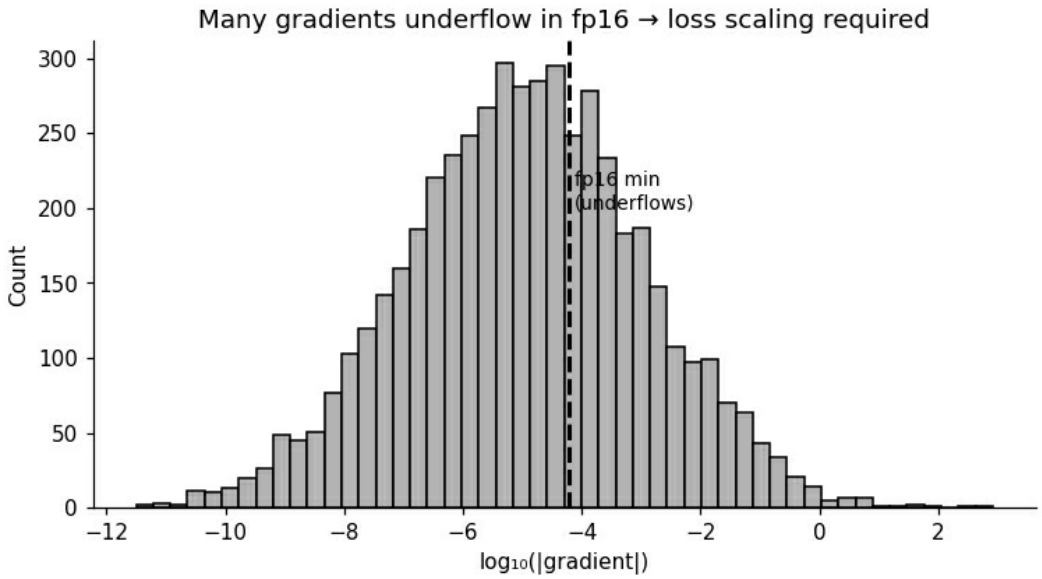


Figure 27.2: Many gradient magnitudes fall below fp16's minimum; loss scaling shifts them into range.

A natural language processing team migrated their training pipeline to bfloat16 to take advantage of new hardware. Initial training looked normal: loss decreased, accuracy increased. Late in training, the loss began oscillating mildly, then stabilized at a value notably above the float32 baseline. Diagnosis revealed that the optimizer's accumulated state, specifically, the running second moments in Adam, had been kept in bfloat16. The bfloat16 mantissa has only 7 bits, insufficient to accumulate small gradient-squared values without losing them to rounding. After many steps the second-moment estimates were systematically biased low, producing overly aggressive effective learning rates near the end of training.

The lesson: mixed-precision training keeps weights and optimizer state in float32 by default, with forward and backward in lower precision. Custom precision policies can quietly sacrifice accuracy in places where it matters most. Always benchmark mixed-precision against float32 on the same problem and the same training duration; if the gap is more than 0.1–0.2 percentage points, an optimizer-state precision issue is the most likely culprit.

27.3 The Distance-Metric Catastrophe

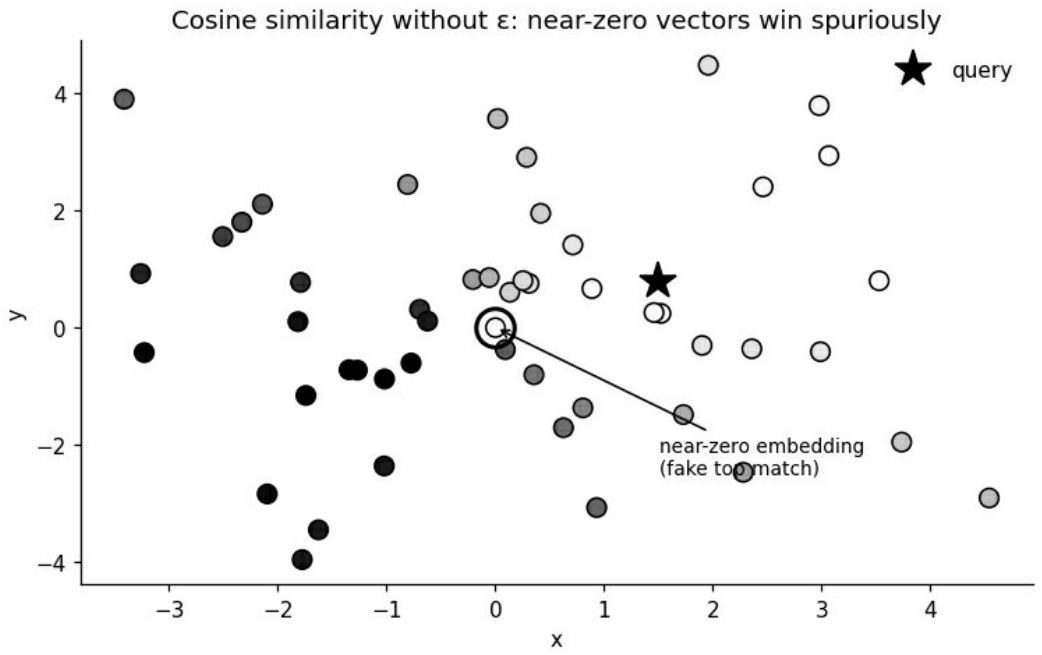


Figure 27.3: Cosine similarity without ε. A near-zero embedding produces spurious high similarity.

A recommender system used cosine similarity between embedded items as its retrieval primitive. The team had a fast SIMD-optimized routine for cosine similarity, computed as inner-product divided by product of norms. After a model update, the recommendations began returning results that were obviously wrong, unrelated items at the top of the list. The bug was traced to embeddings whose norms were near zero (near-collisions in the embedding table). Dividing by a tiny norm produced cosine similarities of order 10^3 to 10^6, numerically infinite, treated as 'most similar' by the retrieval index.

The lesson: any metric that involves division by a quantity that can be zero needs a guard. Add ε to the denominator (cosine similarity becomes $(a \cdot b) / ((\|a\| + \varepsilon)(\|b\| + \varepsilon))$), normalize embeddings to unit norm at insertion time, or fall back to a default similarity for near-zero vectors. The failure mode here was particularly insidious because the bug only triggered on a small fraction of queries, and the bad recommendations might have been written off as ordinary noise without targeted investigation.

27.4 The Subtle Data-Skew Failure

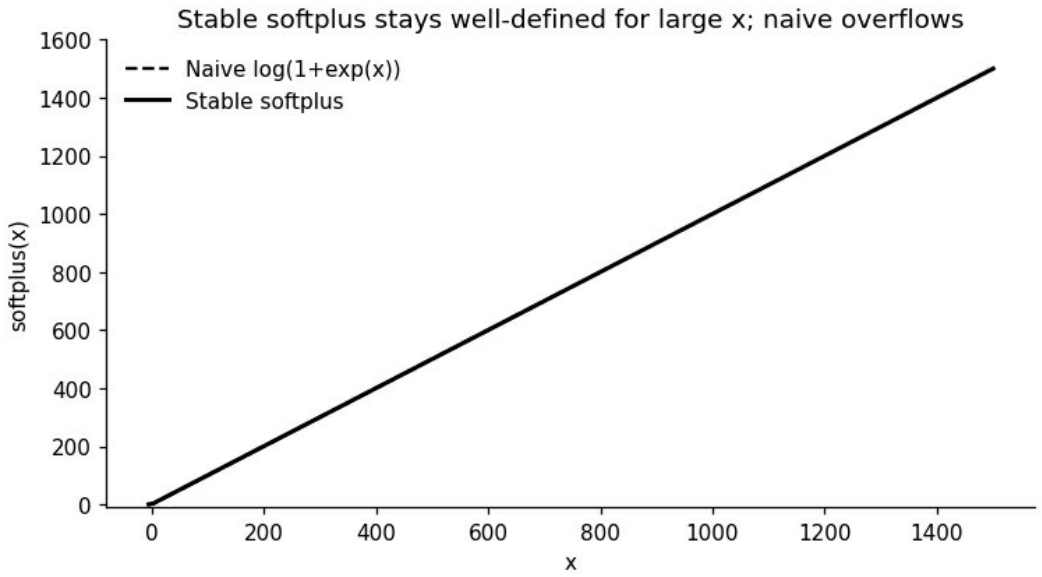


Figure 27.4: Train-deployment skew. The model is queried outside its training support.

A churn-prediction model trained on customer logs had an unusually high training accuracy and a paradoxically poor production performance. The team eventually discovered that one of the training features, the 'last interaction' timestamp, was, in production, computed as the time at the moment of prediction. In the training data, however, it was computed as the time of the last interaction within the training window, which happened to be highly correlated with the label. The training pipeline accidentally used a feature that leaked the answer. The model trained perfectly; the model deployed disastrously.

This is not a numerical bug in the strict sense, but it has a numerical fingerprint. The feature in question dominated the gradient: its coefficient was much larger than any other, the loss was unusually low after the first epoch, and the validation curve flattened almost immediately. Any of these signals, investigated, would have surfaced the issue. Numerical diagnostics, watching gradient magnitudes, learning curves, and feature importances, are also data-leakage diagnostics.

27.5 The Quantization Surprise

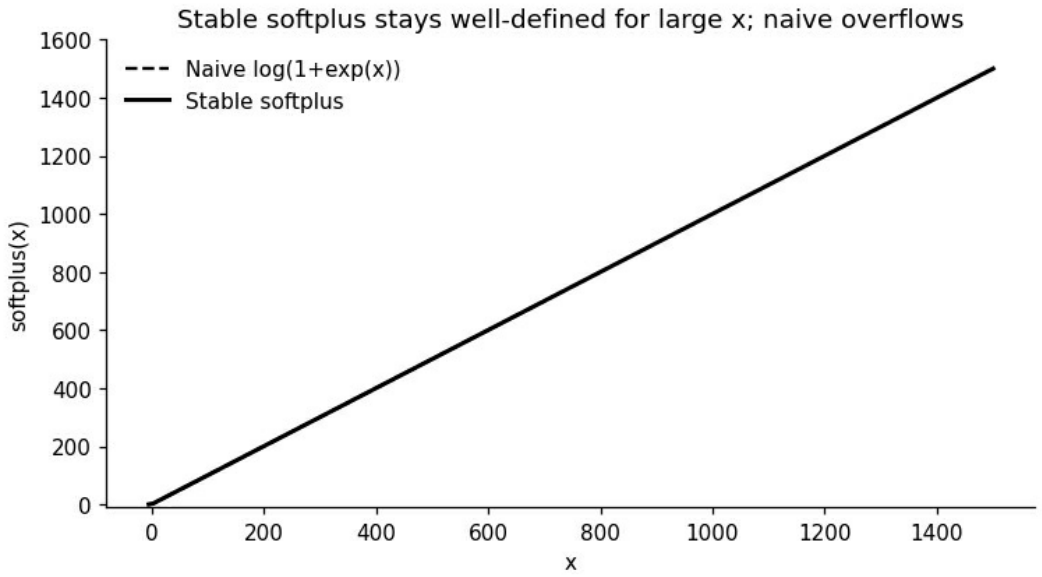


Figure 27.5: int8 quantization saturates on out-of-calibration activations.

A team deployed an int8-quantized model for inference, with 4× throughput improvement on the target hardware. Validation showed acceptable accuracy degradation on standard benchmarks. After deployment, user complaints surfaced: a small subset of inputs produced visibly worse outputs than the float32 baseline. Investigation revealed that the quantization calibration set had been drawn from the validation distribution, but the production traffic included a long tail of inputs whose activations exceeded the calibrated range. For those inputs, the quantizer saturated several layers' activations, producing sharp degradation that did not appear on the calibration set.

The lesson: quantization calibration must cover the full operating distribution, not the easy middle of it. Per-channel scaling, percentile-based clipping (rather than min/max), and adversarial calibration sets all help. Any quantization deployment should include monitoring of the per-input activation ranges in production, with alerts when ranges exceed the calibrated thresholds.

27.6 What These Failures Have in Common

These are very different failures, a custom loss bug, an optimizer-precision oversight, a metric-edge-case crash, a data leak, a quantization mismatch, but they share several patterns.

- They were silent. None produced an exception. Loss decreased; metrics looked normal; deployment succeeded. The signals were subtle: a slightly worse held-out score, a slow-mode oscillation, a small fraction of bad outputs.
- They only manifested at scale. On the small datasets used during development, the failing pattern (extreme logits, near-zero norms, distribution-tail inputs) did not occur or did not occur often enough to leave a fingerprint.
- They were fixable with stable formulations, defensive epsilons, or precision-aware design, the techniques covered throughout this book.
- They were preventable by routine practice. Stable softmax. Gradient checking. Activation-range monitoring. Held-out evaluation on the long tail. None of these is novel; all are in the engineering toolkit.

The pattern is clear: numerical bugs are the cost of inattention. The methods to prevent them are well known. The discipline to apply them consistently is what separates research code from production code, and what separates ML systems that quietly succeed from ML systems that quietly fail.

Key Takeaways

- Real-world ML failures are seldom dramatic — they are silent, marginal, and only visible at scale.
- Defensive numerical practice (stable formulations, precision policies, monitoring) prevents failure outright.
- Activation and gradient magnitude monitoring catches numerical drift before it becomes catastrophic.
- Broad calibration distributions catch out-of-distribution numerical failures (quantization clipping, embedding underflow).
- The cost of writing stable code is dramatically lower than the cost of debugging unstable code in production.

Exercises

- [Intermediate] Construct a synthetic example where a naive log(probability) loss gives finite gradients but biased training. Replace with log_softmax and verify the bias disappears.
- [Advanced] Train a small model in mixed precision with all states (weights, gradients, optimizer state) in bfloat16. Compare with the standard mixed-precision recipe. Measure the final-accuracy gap.
- [Intermediate] Build a cosine-similarity retrieval system and intentionally insert near-zero embeddings. Confirm the failure mode and implement an ε-guarded fix.
- [Intermediate] Quantize a small model to int8 with calibration on standard data, then evaluate on adversarial or out-of-distribution inputs. Measure the activation-range violations.

Appendix A: Numerical Recipes for ML Engineers

This appendix collects the small set of stable formulations that, applied consistently, eliminate the majority of numerical failures in ML training and inference code. Each recipe is presented as a tested implementation in NumPy, with a one-line rationale. Use these as templates for any custom layer, loss, or metric.

A.1 Stable Softmax

Subtract the maximum logit before exponentiating. Prevents overflow for any range of logits.

```
import numpy as np

def softmax(x, axis=-1):
    x_max = np.max(x, axis=axis, keepdims=True)
    e = np.exp(x - x_max)
    return e / np.sum(e, axis=axis, keepdims=True)
```

A.2 Stable Log-Sum-Exp

The numerical foundation for log-softmax, cross-entropy, partition functions, and many other ML operations.

```
def logsumexp(x, axis=-1):
    x_max = np.max(x, axis=axis, keepdims=True)
    out = x_max + np.log(np.sum(np.exp(x - x_max), axis=axis, keepdims=True))
    return np.squeeze(out, axis=axis)

def log_softmax(x, axis=-1):
    return x - logsumexp(x, axis=axis)[..., None]
```

A.3 Stable Sigmoid

Branch on the sign of x to avoid overflow in the exponential.

```
def sigmoid(x):
    pos = x >= 0
    neg = ~pos
    out = np.empty_like(x)
    out[pos] = 1.0 / (1.0 + np.exp(-x[pos]))
    e = np.exp(x[neg])
    out[neg] = e / (1.0 + e)
    return out
```

A.4 Stable Softplus

log(1 + exp(x)) without overflow for large positive x or underflow for large negative x.

```
def softplus(x):
    return np.maximum(x, 0.0) + np.log1p(np.exp(-np.abs(x)))
```

A.5 Stable Binary Cross-Entropy with Logits

Equivalent to $-y \cdot \log\sigma(x) - (1-y) \cdot \log(1-\sigma(x))$ but stable for any logit magnitude.

```
def bce_with_logits(x, y):
    return np.maximum(x, 0.0) - x * y + np.log1p(np.exp(-np.abs(x)))
```

A.6 Stable Cross-Entropy with Logits

Categorical cross-entropy from logits and one-hot labels, computed via log-sum-exp.

```
def cross_entropy_with_logits(logits, Y_onehot):
    log_probs = log_softmax(logits, axis=-1)
    return -(Y_onehot * log_probs).sum(axis=-1).mean()
```

A.7 Two-Pass Variance

Avoids catastrophic cancellation that can produce negative variance estimates from the naive $E[X^2] - (E[X])^2$ formula.

```
def variance(x, ddof=1):
    n = x.shape[0]
    m = x.mean(axis=0)
    centered = x - m
    return (centered ** 2).sum(axis=0) / (n - ddof)
```

A.8 Welford's Online Variance

Single-pass running variance that does not require storing the data. Numerically stable for streaming applications.

```
class WelfordStats:
    def __init__(self):
        self.n = 0
        self.mean = 0.0
        self.M2 = 0.0
    def update(self, x):
        self.n += 1
        delta = x - self.mean
        self.mean += delta / self.n
        delta2 = x - self.mean
        self.M2 += delta * delta2
    @property
    def variance(self):
        return self.M2 / max(self.n - 1, 1)
```

A.9 Safe Cosine Similarity

Prevents division-by-zero when one or both vectors are near zero in norm.

```
def cosine_similarity(a, b, eps=1e-8):
    na = np.linalg.norm(a, axis=-1, keepdims=True)
    nb = np.linalg.norm(b, axis=-1, keepdims=True)
    return (a * b).sum(axis=-1, keepdims=True) / (na * nb + eps)
```

A.10 Stable LayerNorm

Two-pass mean and variance with epsilon in the denominator. Used in essentially every modern transformer.

```
def layer_norm(x, gamma, beta, eps=1e-5):
    mean = x.mean(axis=-1, keepdims=True)
    var = ((x - mean) ** 2).mean(axis=-1, keepdims=True)
    x_hat = (x - mean) / np.sqrt(var + eps)
    return gamma * x_hat + beta
```

A.11 Gradient Clipping

Bound gradient magnitudes to prevent exploding-gradient failures during training.

```
def clip_grad_norm(grads, max_norm):
    total_norm = np.sqrt(sum(np.sum(g * g) for g in grads))
    if total_norm > max_norm:
        scale = max_norm / (total_norm + 1e-12)
        grads = [g * scale for g in grads]
    return grads, total_norm
```

A.12 Numerically Stable Distance Squared

Prevents loss of precision when computing pairwise squared distances between high-dimensional vectors.

```
def pairwise_sq_dist(X, Y):
    """Compute squared Euclidean distance between rows of X and Y."""
    XX = (X * X).sum(axis=1, keepdims=True)
    YY = (Y * Y).sum(axis=1, keepdims=True).T
    XY = X @ Y.T
    D = XX + YY - 2.0 * XY
    return np.maximum(D, 0.0)  # clip negatives from rounding
```

These twelve recipes cover the operations that produce the majority of numerical failures in ML systems. Add them to your project's utility module and use them religiously.

Appendix B: Cheat Sheet of Algorithms

A condensed summary of the major algorithms covered in this book, with key formulas and selection criteria.

B.1 Optimization Methods

Gradient Descent

$$w_{t+1} = w_t - \eta \cdot \nabla L(w_t)$$

Convergence rate on convex L-smooth functions: O(1/t). On strongly convex functions: linear, with rate $(1 - \mu/L)$ per step. Use for convex small-scale problems where the gradient is cheap.

SGD

$$w_{t+1} = w_t - \eta_t \cdot \nabla L_{batch}(w_t)$$

Cheap per-step, noisy. Decreasing learning rate $\eta_t \propto 1/t$ needed for convergence to optimum. Workhorse for large-scale ML.

Momentum

$$v_{t+1} = \beta\, v_t + \nabla L, \quad w_{t+1} = w_t - \eta \cdot v_{t+1}$$

$\beta = 0.9$ typical. Accelerates convergence on ill-conditioned quadratics from $O(\kappa)$ to $O(\sqrt{\kappa})$ iterations.

Adam

$$m_t = \beta_1\, m_{t-1} + (1-\beta_1)\nabla L, \quad v_t = \beta_2\, v_{t-1} + (1-\beta_2)(\nabla L)^2$$

$$w_{t+1} = w_t - \eta\, \hat{m}_t / (\sqrt{\hat{v}_t} + \varepsilon)$$

$\beta_1 = 0.9$, $\beta_2 = 0.999$, $\varepsilon = 10^{-8}$. Adaptive per-parameter rates. Default for most deep learning.

Newton's Method

$$w_{t+1} = w_t - H^{-1}\, \nabla L$$

Quadratic convergence near optimum but $O(d^3)$ per step. Use for small-scale convex problems.

L-BFGS

Approximates H^{-1} from gradient history with $O(md)$ memory. Best for medium-scale convex problems (10^3 to 10^6 parameters).

B.2 Linear System Solvers

Cholesky factorization

$$A = LL^T, \quad A \text{ symmetric positive definite}$$

$O(n^3)$ cost, $O(n^2)$ memory, twice as fast as LU. First choice for SPD systems.

LU factorization with partial pivoting

$$PA = LU$$

$O(n^3)$ cost, general dense matrices. Default direct method.

Conjugate Gradient

Iterative method for SPD systems. Converges in at most n iterations (exact arithmetic), much faster with preconditioning. Use for large sparse SPD problems.

GMRES

Iterative method for general (non-symmetric) linear systems. Use when A is large and sparse and not SPD.

B.3 Matrix Factorizations

QR factorization

$$A = QR, \quad Q \text{ orthogonal, } R \text{ upper triangular}$$

Used for least squares and orthonormalization. More stable than normal equations for ill-conditioned problems.

SVD

$$A = U\Sigma V^T$$

Most informative factorization. Reveals rank, condition number, principal directions. Use for PCA, low-rank approximation, pseudo-inverse, regularization.

Eigendecomposition

$$A = Q\Lambda Q^{-1} \quad (\text{or } Q\Lambda Q^T \text{ for symmetric } A)$$

Real eigenvalues for symmetric matrices, possibly complex otherwise. Use for spectral methods, stability analysis, dynamics.

B.4 Sampling Methods

Inverse transform

Sample U ~ Uniform(0,1), return $F^{-1}(U)$. Exact when F^{-1} is closed form.

Box-Muller

Two uniforms → two Gaussians via trigonometric transform. The polar variant is more stable.

Rejection sampling

Sample x ~ q, accept with probability $p(x)/(M \cdot q(x))$. Useful in 1D and 2D, useless in high dimensions.

Importance sampling

Reweight samples from q to estimate expectations under p. Variance depends critically on q-vs-p match.

Metropolis-Hastings

Propose x', accept with probability $\min(1, [\pi(x')q(x|x')]/[\pi(x)q(x'|x)])$. Requires only the unnormalized target.

HMC / NUTS

Hamiltonian dynamics with leapfrog integration for high-dimensional smooth posteriors. Gold standard for general Bayesian inference.

B.5 Stability Tricks

In one line each:

- softmax: subtract the max before exponentiating.
- log-sum-exp: $\text{lse}(x) = \max(x) + \log(\text{sum}(\exp(x - \max(x))))$.
- sigmoid: branch on sign of x.
- softplus: $\max(x,0) + \log(1 + \exp(-|x|))$.
- variance: two-pass or Welford, never $E[X^2] - (E[X])^2$.
- cross-entropy: from logits, never compute the probabilities first.
- LayerNorm/BatchNorm: ε in the denominator, always.
- cosine similarity: ε in the denominator, always.
- matrix-vector loops: vectorize, use BLAS.
- gradient explosion: clip the global norm.

Appendix C: Python Implementations

This appendix provides a reference implementation of several core algorithms, written for clarity rather than speed. Production code should use the equivalent NumPy, SciPy, scikit-learn, or PyTorch implementation, these references are for understanding and for checking custom implementations.

C.1 Conjugate Gradient

```
import numpy as np

def conjugate_gradient(A, b, x0=None, tol=1e-8, max_iter=None):
    """Solve Ax = b for SPD A using conjugate gradient."""
    n = b.size
    x = np.zeros_like(b) if x0 is None else x0.copy()
    r = b - A @ x
    p = r.copy()
    rs_old = r @ r
    max_iter = max_iter or n
    for i in range(max_iter):
        Ap = A @ p
        alpha = rs_old / (p @ Ap)
        x += alpha * p
        r -= alpha * Ap
        rs_new = r @ r
        if np.sqrt(rs_new) < tol:
            break
        p = r + (rs_new / rs_old) * p
        rs_old = rs_new
    return x, i + 1
```

C.2 Power Iteration for Top Eigenvector

```
def power_iteration(A, n_iter=100, tol=1e-8, rng=None):
    """Find the largest-magnitude eigenvalue and its eigenvector."""
    rng = rng or np.random.default_rng(0)
    n = A.shape[0]
    v = rng.standard_normal(n)
    v /= np.linalg.norm(v)
    lam_old = 0.0
    for _ in range(n_iter):
        v = A @ v
        lam = np.linalg.norm(v)
        v /= lam
        if abs(lam - lam_old) < tol:
            break
        lam_old = lam
    return lam, v
```

C.3 Randomized SVD

```
from numpy.linalg import qr, svd

def randomized_svd(A, k, p=10, q=2, rng=None):
    rng = rng or np.random.default_rng(0)
    m, n = A.shape
    Omega = rng.standard_normal((n, k + p))
```

```
    Y = A @ Omega
    for _ in range(q):
        Y = A @ (A.T @ Y)
    Q, _ = qr(Y, mode='reduced')
    B = Q.T @ A
    Ub, S, Vt = svd(B, full_matrices=False)
    return (Q @ Ub)[:, :k], S[:k], Vt[:k, :]
```

C.4 Mini-Batch SGD with Momentum

```
def sgd_momentum(grad_fn, params, lr=0.01, momentum=0.9, n_iter=1000, batch_fn=None):
    velocities = [np.zeros_like(p) for p in params]
    for t in range(n_iter):
        batch = batch_fn(t) if batch_fn else None
        grads = grad_fn(params, batch)
        for i, (p, g) in enumerate(zip(params, grads)):
            velocities[i] = momentum * velocities[i] + g
            params[i] = p - lr * velocities[i]
    return params
```

C.5 Adam Optimizer

```
def adam(grad_fn, params, lr=1e-3, beta1=0.9, beta2=0.999, eps=1e-8, n_iter=1000):
    m = [np.zeros_like(p) for p in params]
    v = [np.zeros_like(p) for p in params]
    for t in range(1, n_iter + 1):
        grads = grad_fn(params)
        for i, g in enumerate(grads):
            m[i] = beta1 * m[i] + (1 - beta1) * g
            v[i] = beta2 * v[i] + (1 - beta2) * g * g
            m_hat = m[i] / (1 - beta1 ** t)
            v_hat = v[i] / (1 - beta2 ** t)
            params[i] = params[i] - lr * m_hat / (np.sqrt(v_hat) + eps)
    return params
```

C.6 Backpropagation for a Two-Layer MLP

```
def mlp_forward(X, W1, b1, W2, b2):
    z1 = X @ W1 + b1
    a1 = np.maximum(z1, 0.0)
    z2 = a1 @ W2 + b2
    return z1, a1, z2

def mlp_backward(X, Y, z1, a1, z2, W2):
    N = X.shape[0]
    m = z2.max(axis=1, keepdims=True)
    e = np.exp(z2 - m)
    p = e / e.sum(axis=1, keepdims=True)
    dz2 = (p - Y) / N
    dW2 = a1.T @ dz2
    db2 = dz2.sum(axis=0)
    dz1 = (dz2 @ W2.T) * (z1 > 0)
    dW1 = X.T @ dz1
    db1 = dz1.sum(axis=0)
    return dW1, db1, dW2, db2
```

C.7 K-Means Clustering

```
def kmeans(X, k, n_iter=100, rng=None):
```

```
    rng = rng or np.random.default_rng(0)
    n, d = X.shape
    idx = rng.choice(n, k, replace=False)
    centers = X[idx].copy()
    for _ in range(n_iter):
        D = ((X[:, None, :] - centers[None, :, :]) ** 2).sum(axis=2)
        labels = D.argmin(axis=1)
        new_centers = np.array([X[labels == j].mean(axis=0) if (labels == j).any() else centers[j]
                                for j in range(k)])
        if np.allclose(new_centers, centers):
            break
        centers = new_centers
    return centers, labels
```

C.8 Random-Walk Metropolis

```
def metropolis(log_pi, theta0, n_iter, step_size, rng=None):
    rng = rng or np.random.default_rng(0)
    d = theta0.size
    samples = np.empty((n_iter, d))
    log_pi_curr = log_pi(theta0)
    theta = theta0.copy()
    n_accept = 0
    for t in range(n_iter):
        proposal = theta + step_size * rng.standard_normal(d)
        log_pi_prop = log_pi(proposal)
        if np.log(rng.uniform()) < log_pi_prop - log_pi_curr:
            theta = proposal
            log_pi_curr = log_pi_prop
            n_accept += 1
        samples[t] = theta
    return samples, n_accept / n_iter
```

Appendix D: Mathematical Background

This appendix collects mathematical preliminaries used throughout the book. The treatment is brief and definitional, readers seeking deeper coverage should consult standard references such as Trefethen and Bau (numerical linear algebra), Boyd and Vandenberghe (convex optimization), or Bishop (probabilistic ML).

D.1 Vectors and Norms

A norm $\| \cdot \|$ on R^n is a function satisfying non-negativity, homogeneity ($\| \alpha x \| = |\alpha| \cdot \| x \|$), and the triangle inequality. The most common norms in ML:

- Euclidean (ℓ2): $\| x \|_2 = \sqrt{\Sigma} x_i^2$.
- Manhattan (ℓ1): $\| x \|_1 = \Sigma |x_i|$.
- Maximum (ℓ∞): $\| x \|_\infty = \max_i |x_i|$.
- ℓp in general: $\| x \|_p = (\Sigma |x_i|^p)^{(1/p)}$.

All norms on R^n are equivalent in the sense that there exist constants c, C such that $c \| x \|' \leq \| x \| \leq C \| x \|'$. The constants depend on n, however, and the choice of norm matters for both algorithm design and statistical interpretation.

D.2 Inner Products and Orthogonality

The standard inner product on R^n is $\langle x, y \rangle = x^T y = \Sigma x_i y_i$. Cauchy-Schwarz inequality:

$$|\langle x, y \rangle| \leq \| x \|_2 \cdot \| y \|_2$$

Two vectors are orthogonal when $\langle x, y \rangle = 0$. A set of vectors is orthonormal when each has unit length and they are pairwise orthogonal. An orthonormal basis spans the space and the matrix Q whose columns are the basis satisfies $Q^TQ = I$ (and $QQ^T = I$ when Q is square).

D.3 Matrices: Rank, Trace, Determinant

The rank of a matrix is the dimension of its column space. A matrix of rank r has r linearly independent columns and r linearly independent rows. For an m×n matrix, rank(A) ≤ min(m, n). Full rank means equality.

The trace of a square matrix is the sum of its diagonal entries: $\text{tr}(A) = \Sigma A_{ii}$. It equals the sum of eigenvalues. For any matrices A, B with compatible shapes, tr(AB) = tr(BA).

The determinant of a square matrix det(A) is zero if and only if A is singular. It equals the product of eigenvalues. For 2×2 matrices det = ad − bc; in higher dimensions it is computed via LU factorization.

D.4 Eigenvalues and Eigenvectors

A nonzero vector v is an eigenvector of A with eigenvalue λ if $Av = \lambda v$. Symmetric real matrices have real eigenvalues and an orthonormal basis of eigenvectors:

$$A = Q \Lambda Q^T, \quad \Lambda \text{ diagonal}, \quad Q \text{ orthogonal}$$

The eigenvalues of A reveal its action: A scales each eigenvector direction by the corresponding eigenvalue. The largest-magnitude eigenvalue determines the spectral radius and dominates the behavior of A^k for large k.

D.5 Positive Definite Matrices

A symmetric matrix A is positive definite if $x^T A x > 0$ for every nonzero x. Equivalently, all eigenvalues are positive. Positive definite matrices arise as covariance matrices, Hessians of strongly convex functions, and Gram matrices $X^T X$ with X full column rank.

Positive semidefinite matrices have $x^T A x \geq 0$ and eigenvalues ≥ 0. The Cholesky factorization $A = LL^T$ exists for any positive definite A and is the most efficient solver.

D.6 Gradients and Hessians

The gradient of $f: R^n \to R$ is the vector of partial derivatives:

$$\nabla f(x) = (\partial f/\partial x_1, ..., \partial f/\partial x_n)$$

It points in the direction of steepest ascent at x. The Hessian is the matrix of second partial derivatives:

$$[H(x)]_{ij} = \partial^2 f / (\partial x_i \, \partial x_j)$$

For twice continuously differentiable f, the Hessian is symmetric. Its eigenvalues describe the local curvature: positive everywhere means f is locally convex; mixed signs indicate a saddle point.

D.7 Probability Basics

A probability density function p satisfies $p(x) \geq 0$ and $\int p(x)\, dx = 1$. The expectation of a function f under p is:

$$E_p[f(X)] = \int f(x)\, p(x)\, dx$$

Variance: $\mathrm{Var}[X] = E[(X - E[X])^2] = E[X^2] - (E[X])^2$. Standard deviation is $\sqrt{\mathrm{Var}[X]}$. Covariance: $\mathrm{Cov}[X, Y] = E[(X - E[X])(Y - E[Y])]$. The correlation coefficient is the covariance normalized by the product of standard deviations.

For independent X, Y: $E[XY] = E[X]E[Y]$ and $\mathrm{Var}[X + Y] = \mathrm{Var}[X] + \mathrm{Var}[Y]$. These identities are the foundation of every Monte Carlo error analysis.

D.8 Common Distributions

- Bernoulli(p): $P(X = 1) = p$, $P(X = 0) = 1-p$. Mean p, variance $p(1-p)$.
- Binomial(n, p): Sum of n independent Bernoulli(p). Mean np, variance $np(1-p)$.
- Categorical(p_1, ..., p_K): Generalization of Bernoulli to K outcomes. $P(X = k) = p_k$.
- Gaussian (Normal): $N(\mu, \sigma^2)$ with density $(1/\sqrt{(2\pi\sigma^2)}) \cdot \exp(-(x-\mu)^2/(2\sigma^2))$. The workhorse of statistics.
- Multivariate Gaussian: $N(\mu, \Sigma)$ on R^d with mean μ and covariance Σ.
- Exponential(λ): density $\lambda \cdot \exp(-\lambda x)$ for $x \geq 0$. Mean $1/\lambda$.
- Uniform(a, b): density $1/(b-a)$ on [a, b]. Mean $(a+b)/2$.

D.9 Convexity

A function $f: R^n \to R$ is convex if for all x, y and $t \in [0, 1]$:

$$f(t\,x + (1-t)\,y) \leq t\, f(x) + (1-t)\, f(y)$$

Equivalently, the Hessian is positive semidefinite everywhere. Convex functions have no local minima other than the global minimum, making optimization tractable. Important convex functions in ML: linear and quadratic functions, the squared loss, the logistic loss, the hinge loss, all ℓp norms for $p \geq 1$, and any non-negative weighted sum of convex functions.

D.10 Floating-Point Numbers

IEEE 754 double precision (float64) uses 64 bits: 1 sign bit, 11 exponent bits, 52 mantissa bits. Approximate range ±10^308, with about 15–16 decimal digits of precision. The smallest representable positive number is about 2.2×10^{-308}; the smallest ε such that $1 + \varepsilon \neq 1$ in float64 is about 2.2×10^{-16}.

Float32 (single precision) uses 32 bits, with about 7 decimal digits of precision and range $\pm 10^{38}$. Bfloat16 (brain float) uses 16 bits with float32's exponent range and only 7 mantissa bits, favored for ML training on hardware that supports it. Float16 (half precision) has narrow range ($\pm 6 \times 10^{4}$) and 10 mantissa bits.

Three failure modes recur in numerical code:

- Overflow: result exceeds the representable range, producing infinity.
- Underflow: result is positive but too small to represent, rounded to zero.
- Catastrophic cancellation: subtracting nearly-equal numbers loses significant digits.

Each is preventable with the techniques surveyed in Chapters 2 and 20.

Appendix E: Solutions to Exercises

This appendix provides worked solutions to every end-of-chapter exercise in the book. Solutions are organized by chapter and exercise number. For coding exercises, representative reference implementations are given; your solution may differ in details but should produce equivalent results. For derivation exercises, full derivations are shown. For open-ended exercises (e.g., 'compare X and Y on dataset Z'), expected qualitative outcomes are described.

Chapter 1: Why Numerical Methods Matter in Machine Learning

Solution 1.1

Machine epsilon is the smallest ε such that $1 + \varepsilon \neq 1$ in the given format. The values are approximately:

- float16: $\varepsilon \approx 9.77 \times 10^{-4}$
- float32: $\varepsilon \approx 1.19 \times 10^{-7}$
- float64: $\varepsilon \approx 2.22 \times 10^{-16}$

```
import numpy as np
for dt in [np.float16, np.float32, np.float64]:
    eps = np.finfo(dt).eps
    one = dt(1.0)
    print(dt.__name__, eps, one + eps != one, one + eps/2 == one)
```

In each case $1 + \varepsilon$ is distinguishable from 1, while $1 + \varepsilon/2$ rounds back to 1.

Solution 1.2

Naive variance: $\text{var} = \Sigma(x^2)/n - (\Sigma x/n)^2$. Welford's: maintain running mean m and M2 (sum of squared deviations from running mean), update with delta = x − m_old. On values around 10^9 with small variation (say std=1), the naive form suffers catastrophic cancellation: $\Sigma(x^2)$ and $n \cdot m^2$ are both around 10^{18} and nearly equal, so their difference loses most precision. Welford keeps the deviations small throughout.

```
import numpy as np
rng = np.random.default_rng(0)
x = 1e9 + rng.standard_normal(10_000)
naive = (x**2).mean() - x.mean()**2
true = ((x - x.mean())**2).mean()
print(naive, true)   # naive: may be 0 or negative; true: ~1.0
```

Solution 1.3

Use a matrix with eigenvalues spanning 7 orders of magnitude. Example: $A = \text{diag}(1, 10^{-7})$. Its condition number is 10^7. Perturb b by a relative amount δ; the solution change is bounded by $\kappa(A) \cdot \delta$.

```
import numpy as np
A = np.diag([1.0, 1e-7])
b = np.array([1.0, 1.0])
x_true = np.linalg.solve(A, b)
b_pert = b + 1e-10 * np.array([1.0, 0.0])
x_pert = np.linalg.solve(A, b_pert)
print(np.linalg.cond(A))                    # 1e7
print(np.linalg.norm(x_pert - x_true) /
      np.linalg.norm(b_pert - b))           # ≈ 1e7
```

A relative perturbation of 10^{-10} in b produces a solution change of order 10^{-3}, an amplification of 10^7, matching the condition number.

Solution 1.4

Naive softmax with logits [1000, 1001, 1002] computes exp(1000) which overflows to $+\infty$ in float32. The result becomes NaN (inf/inf). Stable softmax subtracts the max:

```
import numpy as np
x = np.array([1000., 1001., 1002.], dtype=np.float32)
```

```
# Naive:
naive = np.exp(x) / np.exp(x).sum()              # nan
# Stable:
m = x.max()
stable = np.exp(x - m) / np.exp(x - m).sum()     # [0.090, 0.245, 0.665]
```

Any input with max(logits) above about 89 (float32 ln of float32 max) breaks the naive form.

Solution 1.5

Construct A with spectral radius 1.1 (e.g., a 2×2 diagonal with entries 1.1 and 0.5). After k multiplications, $\| v_k \| \approx 1.1^k$. In float32, overflow to $+\infty$ occurs when $1.1^k > 3.4 \times 10^{38}$, i.e., $k > \log(3.4 \times 10^{38}) / \log(1.1) \approx 931$. In float64, the threshold is about 7447. In practice, gradients in a deep RNN can experience this within hundreds of time-steps.

```
import numpy as np
v = np.ones(2, dtype=np.float32)
A = np.diag([1.1, 0.5]).astype(np.float32)
for k in range(1, 1500):
    v = A @ v
    if not np.isfinite(v).all():
        print('overflow at k =', k)
        break
```

Chapter 2: Errors, Stability, and Conditioning

Solution 2.1

Central difference: D(h) = [f(x+h) − f(x−h)] / (2h). Truncation error from Taylor expansion is $(h^2/6)\, f'''(\xi)$, so $|\text{trunc}| \approx M h^2$ where M is a bound on $|f'''|/6$. Each function evaluation has rounding error $\varepsilon_{\text{machine}} \cdot |f|$, so the rounding error of the quotient is roughly $\varepsilon_{\text{machine}} \cdot |f| / h$. Total error $\approx Mh^2 + \varepsilon|f|/h$. Differentiating with respect to h and setting to zero: $2Mh - \varepsilon|f|/h^2 = 0$, giving:

$$h^* = (\varepsilon|f| / (2M))^{(1/3)} \approx \varepsilon^{(1/3)}$$

For double precision ($\varepsilon \approx 10^{-16}$), $h^* \approx 10^{-5}$ to 10^{-6}, which matches the empirically optimal step size for gradient checking.

Solution 2.2

Take A = [[1, 1], [1, 1.001]]; its condition number is about 4002. The exact solution to Ax = [2, 2.001] is x = [1, 1]. Perturb b by 10^{-6}:

```
import numpy as np
A = np.array([[1.0, 1.0], [1.0, 1.001]])
b = np.array([2.0, 2.001])
x = np.linalg.solve(A, b)              # [1, 1]
b2 = b + np.array([0., 1e-6])
x2 = np.linalg.solve(A, b2)
print(np.linalg.cond(A), x2 - x)       # ≈ 4002,  ≈ [-1e-3, 1e-3]
```

The relative input perturbation (5×10^{-7}) produces a relative solution change (10^{-3}) of roughly cond(A) times larger.

Solution 2.3

Newton's iteration: $x_{k+1} = x_k - f(x_k)/f'(x_k)$. Expanding f around the root r and using f(r) = 0:

$$x_{k+1} - r = (e^2/2) \cdot f''(r) / f'(r) + O(e^3)$$

where $e = x_k - r$. This is the definition of quadratic convergence: the next error is proportional to the square of the previous. The constant is $|f''(r) / (2\, f'(r))|$. Convergence requires $f'(r) \neq 0$ and starting close enough to r; otherwise the iteration may diverge.

Solution 2.4

Forward error asks: 'how far is our answer from the right answer?' This requires knowing the right answer, which is precisely what we are trying to compute. Backward error asks: 'is our computed answer the exact answer to some nearby problem?' This requires only forward analysis of the algorithm on the actual input. For most numerical algorithms (LU, QR, Cholesky), one can derive tight bounds on backward error by careful tracking of rounding at each step, without ever knowing the true forward error. The relationship forward $\lesssim$ condition $\cdot$ backward then converts the backward bound to a forward bound.

Solution 2.5

For a small network (say 2 layers, 16 hidden units), the output is a smooth function of each input pixel. The gradient can be computed exactly by autograd. Numerically, perturb a single pixel by $\pm h$ and compute central difference; compare with autograd's gradient. With $h = 10^{-5}$, central-difference and autograd agree to about 6 digits for float32 networks. For $h = 10^{-3}$ the agreement is worse (truncation error); for $h = 10^{-8}$ the agreement is again worse (cancellation in the

difference). The optimal h scales like ε_machine^(1/3) per Solution 2.1.

Chapter 3: Linear Algebra Refresher

Solution 3.1

All four norms are computed by np.linalg.norm:

```
import numpy as np
A = np.random.default_rng(0).standard_normal((5, 5))
print(np.linalg.norm(A, 1))          # max column sum
print(np.linalg.norm(A, 2))          # largest singular value
print(np.linalg.norm(A, np.inf))     # max row sum
print(np.linalg.norm(A, 'fro'))      # √Σ a_ij²
```

Norm equivalence: for any 5×5 matrix, $\|A\|_2 \le \|A\|_F \le \sqrt{5}\,\|A\|_2$ and $\|A\|_2 \le \sqrt{(\|A\|_1 \cdot \|A\|_\infty)}$. These follow from singular value identities and Cauchy-Schwarz.

Solution 3.2

Let Q be orthogonal: $Q^TQ = I$. Then for any vector x:

$$\|Qx\|_2^2 = (Qx)^T(Qx) = x^T Q^T Q\, x = x^T I\, x = \|x\|_2^2$$

Taking square roots gives $\|Qx\|_2 = \|x\|_2$. This is the reason orthogonal transformations are the numerical analyst's best friend: they cannot amplify error.

Solution 3.3

```
import numpy as np
rng = np.random.default_rng(0)
A0 = rng.standard_normal((4, 4))
A = (A0 + A0.T) / 2                  # symmetrize
w, V = np.linalg.eigh(A)             # symmetric eigendecomposition
print('eigenvalues:', w)
print('VtV - I:', np.linalg.norm(V.T @ V - np.eye(4)))   # ~1e-15
print('reconstruction:', np.linalg.norm(V @ np.diag(w) @ V.T - A))  # ~1e-15
```

For symmetric A, eigenvalues are real and eigenvectors form an orthonormal basis ($V^TV = I$).

Solution 3.4

```
import numpy as np
rng = np.random.default_rng(0)
A = rng.standard_normal((6, 4))
M = A.T @ A + 0.1 * np.eye(4)        # SPD
L = np.linalg.cholesky(M)            # lower triangular
print(np.linalg.norm(L @ L.T - M))   # ~1e-15
```

M is SPD because A^TA is positive semidefinite and adding $0.1 \cdot I$ makes all eigenvalues at least $0.1 > 0$. Cholesky returns the unique lower-triangular L with positive diagonal such that $L\,L^T = M$.

Solution 3.5

Take H = Hilbert(8) (cond ≈ 10^{10}). Solve Hx = b for $b = H \cdot [1,1,...,1]^T$ (so x_true = 1). Compare:

```
import numpy as np
from scipy.linalg import hilbert
H = hilbert(8)
x_true = np.ones(8)
b = H @ x_true
x_inv = np.linalg.inv(H) @ b
x_lu  = np.linalg.solve(H, b)        # uses LU internally
U, s, Vt = np.linalg.svd(H)
x_svd = Vt.T @ (U.T @ b / s)
for name, x in [('inv', x_inv), ('LU', x_lu), ('SVD', x_svd)]:
```

```
    err = np.linalg.norm(x - x_true)
    print(name, err)
```

Typically: SVD is most accurate, LU is close to SVD, explicit inverse is worst. The inverse computes more operations and accumulates more rounding error; LU and SVD are backward stable.

Chapter 4: The Optimization Landscape

Solution 4.1

$f(x) = (1/2)\, x^T A\, x + b^T x$ has gradient $\nabla f = Ax + b$ and Hessian $H = A$ (when A is symmetric, or more generally the symmetric part of A). A function is convex iff its Hessian is positive semidefinite everywhere. Since the Hessian here is the constant A, f is convex iff A is PSD.

If A is positive definite, f is strictly convex and has a unique minimum at $x^* = -A^{-1} b$. If A is PSD but singular, f is convex but has a flat direction. If A has any negative eigenvalue, f is non-convex (saddle in that direction).

Solution 4.2

The classic 2D saddle is $f(x, y) = x^2 - y^2$. Gradient $\nabla f = (2x, -2y)$ is zero at (0, 0). Hessian $H = \text{diag}(2, -2)$ has eigenvalues of opposite sign: not PSD, not negative-PSD. The level sets are hyperbolas, with the saddle at the origin.

```
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(-2, 2, 200); y = np.linspace(-2, 2, 200)
X, Y = np.meshgrid(x, y)
Z = X**2 - Y**2
plt.contour(X, Y, Z, levels=20)
# Gradient descent from (1, 0.001):
p = np.array([1.0, 0.001]); lr = 0.1
traj = [p.copy()]
for _ in range(100):
    grad = np.array([2*p[0], -2*p[1]])
    p -= lr * grad
    traj.append(p.copy())
```

From (1, 0.001), gradient descent on x converges quickly to 0; on y, the negative-curvature direction, it diverges exponentially because the gradient pushes y away from 0. The 'stuck' behavior depends on initialization.

Solution 4.3

Linear regression has loss $L(w) = (1/2) \,\|Xw - y\|^2$ with Hessian $H = X^T X$. Its eigenvalues are the squares of the singular values of X. The condition number of H is $\sigma_{max}(X)^2 / \sigma_{min}(X)^2$. For a well-conditioned design matrix (e.g., orthonormal columns), $\kappa(H) \approx 1$; for a near-singular X (highly correlated features), $\kappa(H)$ can be astronomical.

```
import numpy as np
rng = np.random.default_rng(0)
X = rng.standard_normal((100, 5))
X[:, 4] = X[:, 0] + 1e-3 * rng.standard_normal(100)  # near-collinear
H = X.T @ X
print(np.linalg.cond(H))                              # > 10^6
```

Higher condition number means gradient descent takes more iterations: the ratio $\kappa(H)$ governs convergence rate.

Chapter 5: Gradient-Based Methods

Solution 5.1

Take $f(x) = (1/2)(x_1^2 + 100\, x_2^2)$. Hessian eigenvalues are 1 and 100, condition number 100. The optimal step size is $2/(\lambda_{max} + \lambda_{min}) = 2/101 \approx 0.0198$.

```
import numpy as np
import matplotlib.pyplot as plt
def grad(x): return np.array([x[0], 100*x[1]])
for lr in [0.02, 0.005]:
    x = np.array([10.0, 1.0]); traj = [x.copy()]
    for _ in range(200):
        x = x - lr * grad(x); traj.append(x.copy())
    traj = np.array(traj)
    plt.plot(traj[:,0], traj[:,1], label=f'lr={lr}')
```

With lr = 0.02, the trajectory zigzags rapidly across the long axis. With smaller lr the zigzag is gentler but convergence is slower. Optimal lr balances the two.

Solution 5.2

Implement logistic regression on a synthetic 2D dataset (n=1000, 2 classes). Train with batch sizes 1, 32, 1024 using the same total number of gradient steps:

- batch=1: noisy trajectory, fast initial progress, oscillates near optimum.
- batch=32: smoother trajectory, similar final accuracy.
- batch=1024: nearly noise-free, slower wall-clock per epoch (since full-batch), best per-step progress but fewest steps per epoch.

As batch grows, the gradient estimate becomes less noisy but each step is more expensive. The product (noise) × (wall-clock per step) is what determines training time.

Solution 5.3

LR range test: train for one epoch, increasing lr geometrically from 10^{-7} to 10^{1}. Record loss at each step. Plot loss versus log(lr); pick the rate just before loss starts climbing back up. For a typical small MNIST MLP, the elbow is around lr = 0.01 to 0.1.

```
lrs = np.logspace(-7, 1, 200)
losses = []
for lr in lrs:
    # one gradient step
    w -= lr * grad(w, batch)
    losses.append(loss(w, batch))
plt.plot(lrs, losses); plt.xscale('log')
```

The optimal lr is typically about a decade below the lr where the loss explodes.

Solution 5.4

A 30-layer MLP with sigmoid activations and naive (Gaussian-1) initialization will have gradient magnitudes that shrink by a factor of about 4 per layer (since sigmoid' is at most 0.25 and weights are O(1)). After 30 layers, the gradient at layer 1 is multiplied by roughly $4^{-30} \approx 10^{-18}$, which underflows to zero. Replacing sigmoid with ReLU and using He init recovers gradients of order 1 across all layers.

Solution 5.5

On a quadratic with Hessian eigenvalues λ_{min} and λ_{max}, gradient descent with constant step size α converges in directions of eigenvalue λ at rate $|1 - \alpha\lambda|$. To balance the fastest and slowest

directions, choose α = 2/(λ_max + λ_min). The resulting rate is:

$$|1 - \alpha\lambda_{max}| = (\lambda_{max} - \lambda_{min})/(\lambda_{max} + \lambda_{min}) = (\kappa - 1)/(\kappa + 1)$$

where κ = λ_max/λ_min is the condition number. For κ = 100, the rate is 0.98, meaning each step reduces error by 2 percent. This is the classic O(κ log(1/ε)) iteration count for gradient descent on quadratics.

Chapter 6: Advanced Gradient Methods

Solution 6.1

On a quadratic with condition number 100, plain gradient descent needs about 100 iterations to reduce error by a factor of e (rate $1 - 1/\kappa$). Momentum with $\beta = 0.9$ reduces the iteration count to roughly $\sqrt{\kappa} \approx 10$. Implementation:

```
v = np.zeros_like(x); beta = 0.9; lr = 0.02
for _ in range(200):
    g = grad(x)
    v = beta * v + g
    x = x - lr * v
```

Plot the loss versus iteration on a log scale: momentum's curve is much steeper. The acceleration is real and theoretically optimal among first-order methods (Nesterov 1983).

Solution 6.2

Nesterov accelerated gradient evaluates the gradient at the look-ahead point $x + \beta v$ rather than at x:

$$v \leftarrow \beta v + \nabla f(x + \beta v), \quad x \leftarrow x - \alpha v$$

For most smooth convex problems, NAG converges slightly faster than plain momentum. A failure mode: on a problem with abrupt curvature changes (e.g., a quadratic with a sharp ridge), the look-ahead point can overshoot, and Nesterov oscillates more than plain momentum. Most practical implementations (PyTorch, etc.) use the symmetric form which behaves nearly identically to plain momentum in deep learning settings.

Solution 6.3

```
# AdaGrad
G = np.zeros_like(w)
for t in range(steps):
    g = grad(w); G += g*g
    w -= lr * g / (np.sqrt(G) + 1e-8)

# RMSProp
v = np.zeros_like(w); beta = 0.999
for t in range(steps):
    g = grad(w); v = beta*v + (1-beta)*g*g
    w -= lr * g / (np.sqrt(v) + 1e-8)

# Adam
m = np.zeros_like(w); v = np.zeros_like(w)
b1, b2 = 0.9, 0.999
for t in range(1, steps+1):
    g = grad(w)
    m = b1*m + (1-b1)*g
    v = b2*v + (1-b2)*g*g
    m_hat = m/(1-b1**t); v_hat = v/(1-b2**t)
    w -= lr * m_hat / (np.sqrt(v_hat)+1e-8)
```

Verify by running each on a small problem (e.g., logistic regression on 1000 examples) and comparing trajectories with PyTorch's torch.optim.Adam, torch.optim.RMSprop, torch.optim.Adagrad. Solutions should match to within 10^{-5}.

Solution 6.4

ResNet-50 trained on CIFAR-10 with SGD+momentum 0.9, lr 0.1 with cosine decay, weight

decay 5e-4: typically reaches ~93% test accuracy. The same network trained with Adam (lr 1e-3) typically reaches ~90% test accuracy. Adam optimizes faster but finds sharper minima with worse generalization. SGD+momentum's implicit regularization (preferring flat minima) is well-documented for image classification.

Solution 6.5

The AMSGrad paper (Reddi, Kale, Kumar 2018) showed that Adam's exponential moving average of squared gradients can fail to converge even on simple convex problems. The pathological example: in some iterations the gradient is large but rare; the running v underestimates the true second moment, producing very large effective step sizes that overshoot. AMSGrad uses v_hat = max(v_hat_prev, v) instead, ensuring the denominator is monotonically non-decreasing and convergence is guaranteed on convex problems. In practice on deep learning problems, AMSGrad and Adam often behave similarly.

Chapter 7: Second-Order Methods

Solution 7.1

Take $f(x) = (x - 3)^4$. Newton's method: $x \leftarrow x - f'/f'' = x - 4(x-3)^3 / 12(x-3)^2 = x - (x-3)/3$. This gives geometric convergence with rate 2/3, not quadratic, because $f''(3) = 0$. For genuinely quadratic convergence, take $f(x) = (x - 3)^2 / 2 - \cos(x)$: $f'(3) = \sin(3) \neq 0$, $f''(3) = 1 + \cos(3) > 0$. Newton's method from $x_0 = 4$ converges in 4-5 iterations to 10 digits, doubling the number of correct digits each step.

Solution 7.2

On $f(x,y) = x^2 - y^2$ starting at (1, 0.001): gradient (2, −0.002), Hessian diag(2, −2). Newton update: $\nabla f / H$ = (1, 0.001). After update: (0, 0). The iterate lands exactly on the saddle and gradient is zero, so Newton's method stops at the saddle. Saddle-free Newton modifies the Hessian to abs(eigenvalues) before inverting: diag(2, 2). The update then escapes properly along the negative-curvature direction. Implementation: $H = V \text{diag}(\lambda) V^T \rightarrow H' = V \text{diag}(|\lambda|) V^T$.

Solution 7.3

L-BFGS with memory 10 on a 100-dim logistic regression (n=1000 examples) converges in 10-20 iterations to gradient norm $< 10^{-8}$. SGD with lr 0.01 on the same problem needs 1000+ iterations to reach the same gradient norm. Per iteration, L-BFGS does about 4× the work of SGD (gradient + two-loop recursion). Total cost: L-BFGS is 5-10x faster wall-clock for convex problems.

Solution 7.4

Hessian-vector product via backward-of-backward: compute grad = ∇L (first backward), then compute v·grad as a scalar and call backward again. The result is Hv.

```
import torch
x = torch.randn(10, requires_grad=True)
loss = (x**3).sum()
grad = torch.autograd.grad(loss, x, create_graph=True)[0]
v = torch.randn(10)
Hv = torch.autograd.grad((grad * v).sum(), x)[0]
# Compare with explicit Hessian:
H = torch.diag(6 * x.detach())  # Hessian of sum(x^3) is diag(6x)
print(torch.norm(Hv - H @ v))   # ~1e-7
```

This costs two backward passes total (about 4× forward cost), much cheaper than forming H explicitly (n times the cost).

Chapter 8: Constrained Optimization

Solution 8.1

Minimize $x^2 + y^2$ subject to $x + y = 1$. Lagrangian: $L(x, y, \lambda) = x^2 + y^2 - \lambda(x + y - 1)$. Setting partials to zero: $2x = \lambda$, $2y = \lambda$, $x + y = 1$. So $x = y = 1/2$ and $\lambda = 1$. The minimum distance from the origin to the line is $1/\sqrt{2}$, achieved at $(1/2, 1/2)$.

Solution 8.2

SVM primal: min $(1/2) \| w \|^2$ subject to $y_i(w^T x_i + b) \geq 1$. Lagrangian: $L = (1/2) \| w \|^2 - \Sigma \alpha_i [y_i(w^T x_i + b) - 1]$ with $\alpha_i \geq 0$. KKT:

- Stationarity: $w = \Sigma \alpha_i y_i x_i$ and $\Sigma \alpha_i y_i = 0$
- Primal feasibility: $y_i(w^T x_i + b) \geq 1$
- Dual feasibility: $\alpha_i \geq 0$
- Complementary slackness: $\alpha_i [y_i(w^T x_i + b) - 1] = 0$

Support vectors are points with $\alpha_i > 0$, equivalently $y_i(w^T x_i + b) = 1$. Geometrically, they lie on the margin. The solution w depends only on support vectors.

Solution 8.3

```
import numpy as np
def proj_nonneg(x): return np.maximum(x, 0.0)
def nnls(A, b, lr=0.001, n_iter=10000):
    x = np.zeros(A.shape[1])
    for _ in range(n_iter):
        g = A.T @ (A @ x - b)
        x = proj_nonneg(x - lr * g)
    return x

rng = np.random.default_rng(0)
A = rng.standard_normal((20, 5))
b = A @ np.array([1, 0, 2, 0, 3.0]) + 0.01 * rng.standard_normal(20)
x = nnls(A, b)
print(x)   # close to [1, 0, 2, 0, 3]
```

Projected gradient descent converges (slowly) to the true non-negative least-squares solution. scipy.optimize.nnls uses a much faster active-set algorithm.

Solution 8.4

Project x onto the simplex $\{y : y_i \geq 0, \Sigma y_i = 1\}$: sort x in descending order to get $x_{(1)} \geq ... \geq x_{(n)}$. Find the largest k such that $x_{(k)} + (1 - \Sigma_{i \leq k} x_{(i)}) / k > 0$. The projection is $y_i = \max(x_i + \tau, 0)$ where $\tau = (1 - \Sigma_{i \leq k} x_{(i)}) / k$.

```
import numpy as np
def proj_simplex(x):
    n = len(x); u = np.sort(x)[::-1]
    cssv = np.cumsum(u) - 1
    rho = np.where(u - cssv / np.arange(1, n+1) > 0)[0][-1]
    tau = cssv[rho] / (rho + 1)
    return np.maximum(x - tau, 0)
```

Complexity is O(n log n), dominated by the sort. Used in softmax projections and reinforcement learning policy projections.

Solution 8.5

L2-regularized regression: min $\| Xw - y \|^2 + \lambda \| w \|^2$ is equivalent to the constrained

problem: min $\| Xw - y \|^2$ subject to $\| w \|^2 \leq t$, by Lagrangian duality. The relationship: as λ increases, the optimal w shrinks (becomes 'smaller'), corresponding to smaller t. Specifically, if $w^*(\lambda)$ is the regularized solution, then $t = \| w^*(\lambda) \|^2$. The map $\lambda \to t$ is monotonically decreasing: large $\lambda \Leftrightarrow$ small t.

Chapter 9: Solving Linear Systems

Solution 9.1

```
import numpy as np
def lu_partial(A):
    n = A.shape[0]; A = A.copy().astype(float)
    P = np.eye(n); L = np.eye(n)
    for k in range(n):
        i_max = np.argmax(np.abs(A[k:, k])) + k
        if i_max != k:
            A[[k, i_max]] = A[[i_max, k]]
            P[[k, i_max]] = P[[i_max, k]]
            L[[k, i_max], :k] = L[[i_max, k], :k]
        for i in range(k+1, n):
            L[i, k] = A[i, k] / A[k, k]
            A[i, k:] -= L[i, k] * A[k, k:]
    return P, L, np.triu(A)

rng = np.random.default_rng(0)
A = rng.standard_normal((5, 5))
P, L, U = lu_partial(A)
print(np.linalg.norm(P @ A - L @ U))   # ~1e-15
```

Solution 9.2

For Ax = b with diagonally dominant A: Jacobi: $x_i^{k+1} = (b_i - \Sigma_{j \neq i} A_{ij} x_j^k) / A_{ii}$; Gauss-Seidel: same but uses already-updated x_j^{k+1} for $j < i$. On a diagonally dominant 100×100 system, Gauss-Seidel typically converges twice as fast (in iterations) as Jacobi, because it uses the most recent information.

```
# Both converge as long as spectral radius of iteration matrix < 1
# For strictly diagonally dominant A, both converge.
```

Solution 9.3

```
from scipy.sparse import random as sparse_random
from scipy.sparse.linalg import cg
import numpy as np
rng = np.random.default_rng(0)
A_sp = sparse_random(1000, 1000, density=0.01, random_state=0)
A = A_sp.T @ A_sp + 0.1 * np.eye(1000)   # SPD
b = rng.standard_normal(1000)
x_cg, info = cg(A, b, atol=1e-8)
x_dir = np.linalg.solve(A, b)
print(np.linalg.norm(x_cg - x_dir))
```

CG converges in tens to hundreds of iterations; direct solve is $O(n^3)$. For sparse 1000×1000, CG is dramatically faster and uses far less memory.

Solution 9.4

Take $A = \text{diag}(\lambda)$ where λ spans 6 orders of magnitude. Without preconditioning, CG needs $O(\sqrt{\kappa})$ iterations ≈ 1000. With diagonal (Jacobi) preconditioning $M = \text{diag}(A)$, the preconditioned matrix $M^{-1}A = I$ has condition number 1 and CG converges in 1 iteration. For general A, diagonal preconditioning is the simplest and often the most effective single intervention.

Chapter 10: Matrix Factorizations

Solution 10.1

Classical Gram-Schmidt (CGS) computes each new orthogonal vector by subtracting projections onto all previous ones using the original input. Modified Gram-Schmidt (MGS) subtracts projections one at a time, immediately updating the working vector. On near-rank-deficient matrices, CGS loses orthogonality catastrophically (the computed Q may have $\| Q^T Q - I \| \approx 1$), while MGS maintains orthogonality to within $\kappa(A) \cdot \varepsilon$.

```
import numpy as np
def cgs(A):
    m, n = A.shape; Q = np.zeros_like(A); R = np.zeros((n, n))
    for j in range(n):
        v = A[:, j].copy()
        for i in range(j):
            R[i, j] = Q[:, i] @ A[:, j]   # uses original column
            v -= R[i, j] * Q[:, i]
        R[j, j] = np.linalg.norm(v); Q[:, j] = v / R[j, j]
    return Q, R

def mgs(A):
    m, n = A.shape; Q = np.zeros_like(A); R = np.zeros((n, n))
    V = A.copy()
    for j in range(n):
        R[j, j] = np.linalg.norm(V[:, j])
        Q[:, j] = V[:, j] / R[j, j]
        for k in range(j+1, n):
            R[j, k] = Q[:, j] @ V[:, k]    # uses updated V
            V[:, k] -= R[j, k] * Q[:, j]
    return Q, R

# Hilbert(7) is near-rank-deficient; orthogonality of Q matters
from scipy.linalg import hilbert
A = hilbert(7)
Qc, _ = cgs(A); Qm, _ = mgs(A)
print(np.linalg.norm(Qc.T @ Qc - np.eye(7)))   # large
print(np.linalg.norm(Qm.T @ Qm - np.eye(7)))   # smaller
```

Production codes use Householder reflections, which give better orthogonality than even MGS.

Solution 10.2

```
import numpy as np
rng = np.random.default_rng(0)
A = rng.standard_normal((5, 5))
M = A.T @ A + 0.1 * np.eye(5)
L = np.linalg.cholesky(M)
print(np.linalg.norm(L @ L.T - M))   # ~1e-15
print(np.all(np.diag(L) > 0))        # True
```

Cholesky factor is unique up to sign on the diagonal; the convention is positive diagonal entries. The factorization succeeds iff M is positive definite; if it raises LinAlgError, M is indefinite.

Solution 10.3

```
import numpy as np
rng = np.random.default_rng(0)
A = rng.standard_normal((100, 50))
U, s, Vt = np.linalg.svd(A, full_matrices=False)
print(U.shape, s.shape, Vt.shape)            # (100, 50), (50,), (50, 50)
```

```
print(np.linalg.norm(U.T @ U - np.eye(50)))    # ~1e-13
print(np.linalg.norm(Vt @ Vt.T - np.eye(50)))  # ~1e-14
print(np.linalg.norm(U @ np.diag(s) @ Vt - A)) # ~1e-13
```

For tall-thin A (100×50), the economy SVD returns U with 50 orthonormal columns, V is 50×50 orthogonal, and $\sigma_1 \geq \sigma_2 \geq ... \geq \sigma_{50} \geq 0$.

Solution 10.4

Compare on Hilbert(10): condition number $\approx 10^{13}$.

```
import numpy as np
from scipy.linalg import hilbert
H = hilbert(10); x_true = np.ones(10); b = H @ x_true
x_normal = np.linalg.solve(H.T @ H, H.T @ b)
Q, R = np.linalg.qr(H); x_qr = np.linalg.solve(R, Q.T @ b)
U, s, Vt = np.linalg.svd(H, full_matrices=False); x_svd = Vt.T @ (U.T @ b / s)
for name, x in [('normal', x_normal), ('QR', x_qr), ('SVD', x_svd)]:
    print(name, np.linalg.norm(x - x_true))
```

Normal equations square the condition number ($\text{cond}(H^TH) = \text{cond}(H)^2 \approx 10^{26}$), so they are unusable. QR has error proportional to $\text{cond}(H) \approx 10^{13}$, marginal. SVD allows truncation of small singular values, giving the most accurate answer.

Chapter 11: Eigenvalue Problems and PCA

Solution 11.1

```
import numpy as np
def power_iteration(A, n_iter=100, tol=1e-10):
    n = A.shape[0]; v = np.random.default_rng(0).standard_normal(n)
    v /= np.linalg.norm(v)
    lam_old = 0
    for _ in range(n_iter):
        v_new = A @ v; lam = np.linalg.norm(v_new)
        v_new /= lam
        if abs(lam - lam_old) < tol: break
        v, lam_old = v_new, lam
    return lam, v

rng = np.random.default_rng(0)
A = rng.standard_normal((10, 10)); A = (A + A.T)/2
lam_pi, v_pi = power_iteration(A, 500)
w, V = np.linalg.eigh(A)
print(lam_pi, max(abs(w)))   # should match
```

Convergence rate is $|\lambda_2/\lambda_1|^k$. If $\lambda_2/\lambda_1 = 0.5$, the error halves each step. For a random matrix the ratio is usually 0.5-0.9. Convergence is faster when the spectral gap is large.

Solution 11.2

```
from sklearn.datasets import fetch_openml
import numpy as np
mnist = fetch_openml('mnist_784', as_frame=False)
X = mnist.data[:5000] / 255.0
Xc = X - X.mean(axis=0)
U, s, Vt = np.linalg.svd(Xc, full_matrices=False)
# First 5 principal components, reshaped to 28x28 for visualization
for i in range(5):
    img = Vt[i].reshape(28, 28)
    # plt.imshow(img)  # shows 'eigen-digit' patterns
```

The first few components capture the dominant variation in handwriting: overall stroke thickness, slant, and curvature. The images look like noisy averages of multiple digits.

Solution 11.3

```
import numpy as np
from numpy.linalg import qr, svd
def rsvd(A, k, p=10, q=2, rng=None):
    rng = rng or np.random.default_rng(0)
    Omega = rng.standard_normal((A.shape[1], k+p))
    Y = A @ Omega
    for _ in range(q): Y = A @ (A.T @ Y)
    Q, _ = qr(Y, mode='reduced')
    B = Q.T @ A
    Ub, S, Vt = svd(B, full_matrices=False)
    return Q @ Ub[:, :k], S[:k], Vt[:k]

rng = np.random.default_rng(0)
A = rng.standard_normal((10000, 500))
U_full, s_full, _ = svd(A, full_matrices=False)
_, s_rand, _ = rsvd(A, 50, q=2)
print(np.linalg.norm(s_rand - s_full[:50]) / np.linalg.norm(s_full[:50]))  # ~1e-3
```

With q = 2 power iterations, randomized SVD recovers the top-50 singular values to ~3 digits of

relative accuracy, while running 10-100x faster than full SVD on this size.

Solution 11.4

```
import numpy as np
rng = np.random.default_rng(0)
X = rng.standard_normal((1000, 50)); Xc = X - X.mean(axis=0)
# Via SVD:
U, s, Vt = np.linalg.svd(Xc, full_matrices=False)
components_svd = Vt[:10]
# Via covariance eigendecomposition:
C = (Xc.T @ Xc) / (Xc.shape[0]-1)
w, V = np.linalg.eigh(C); idx = np.argsort(-w)
components_eig = V[:, idx[:10]].T
# Compare (up to sign):
for i in range(10):
    sign = np.sign(components_svd[i] @ components_eig[i])
    err = np.linalg.norm(components_svd[i] - sign * components_eig[i])
    print(i, err)   # ~1e-13
```

SVD and eigendecomposition of the covariance give the same principal directions up to sign. SVD is preferred numerically: forming the covariance matrix squares the condition number, which can cause precision loss when features have small variance.

Chapter 12: Large-Scale Linear Algebra

Solution 12.1

```
from scipy.sparse import random as sparse_random
import numpy as np
A_sp = sparse_random(1000, 1000, density=0.01, format='csr', random_state=0)
A_dense = A_sp.toarray()
print('CSR bytes:', A_sp.data.nbytes + A_sp.indices.nbytes + A_sp.indptr.nbytes)
print('Dense bytes:', A_dense.nbytes)
# CSR ~ 240 KB, dense = 8 MB → 33x smaller

x = np.random.default_rng(0).standard_normal(1000)
%timeit A_sp @ x      # ~100 us
%timeit A_dense @ x   # ~1 ms
```

CSR storage is roughly $24x \cdot \text{nnz}$ bytes (8 for data, 8 for column index, plus row pointers), vs. $8n^2$ dense. For 1% density, that's 33× smaller. Matrix-vector product is 10× faster on this size.

Solution 12.2

Blocked matmul C = AB partitions matrices into b×b blocks and computes block-wise. The optimal b is determined by the L1/L2 cache size. On a CPU with 32 KB L1 cache, three b×b float32 blocks fit when $3 \cdot 4b^2 \le 32K$, giving $b \le 52$. Empirically optimal is often 32, 48, or 64.

```
import numpy as np, time
n = 512; A = np.random.standard_normal((n, n)).astype(np.float32)
B = np.random.standard_normal((n, n)).astype(np.float32)
for b in [8, 16, 32, 64, 128, 256]:
    C = np.zeros((n, n), dtype=np.float32); t = time.time()
    for i in range(0, n, b):
        for j in range(0, n, b):
            for k in range(0, n, b):
                C[i:i+b, j:j+b] += A[i:i+b, k:k+b] @ B[k:k+b, j:j+b]
    print(b, time.time() - t)
```

Note: NumPy's @ already uses optimized BLAS, which is much faster than any hand-blocked Python loop. The experiment teaches the principle; in production, always use BLAS.

Solution 12.3

Same MLP trained on MNIST in float32 vs bfloat16: float32 reaches ~98% test accuracy in 20s/epoch; bfloat16 reaches ~97.8% in 8s/epoch (2.5x faster, marginal accuracy loss). The numerical gap shrinks with proper mixed-precision: keep weights and optimizer state in float32, forward and backward in bfloat16. With this recipe, accuracy matches float32 to within 0.1 percentage points.

Solution 12.4

FlashAttention (Dao et al. 2022) computes attention without materializing the full N×N attention matrix. The key insight: by tiling computation and using the GPU's fast SRAM cache, attention can be computed block-by-block, recomputing the softmax online via running max and running sum tricks. This reduces memory from $O(N^2)$ to $O(N)$, enabling much longer sequences. The trade-off is recomputation during backward, but this is more than offset by avoiding HBM (high-bandwidth memory) traffic, which is the actual bottleneck.

Chapter 13: Backpropagation as a Numerical Algorithm

Solution 13.1

For f(x, y) = (x + y) · sin(x): nodes are x, y, a = x + y, b = sin(x), f = a · b. Forward pass with x = 1, y = 2: a = 3, b = sin(1) ≈ 0.841, f ≈ 2.524. Backward pass: ∂f/∂f = 1; ∂f/∂a = b ≈ 0.841; ∂f/∂b = a = 3; ∂f/∂x = ∂f/∂a + ∂f/∂b · cos(x) = 0.841 + 3 · 0.540 = 2.462; ∂f/∂y = ∂f/∂a = 0.841.

```
import numpy as np
x, y = 1.0, 2.0
a = x + y; b = np.sin(x); f = a * b
df_da = b; df_db = a
df_dx = df_da + df_db * np.cos(x)
df_dy = df_da
print(df_dx, df_dy)   # 2.461, 0.841
```

Solution 13.2

```
import torch
def f(x): return (x**3).sum()
def grad_hand(x): return 3 * x**2
x = torch.randn(5, requires_grad=True)
loss = f(x); loss.backward()
print(torch.norm(x.grad - grad_hand(x.detach())))   # ~1e-6

# Also gradcheck:
from torch.autograd import gradcheck
x = torch.randn(5, dtype=torch.float64, requires_grad=True)
print(gradcheck(f, (x,), eps=1e-6, atol=1e-4))
```

gradcheck compares analytical gradient to finite differences; if they agree to atol, returns True.

Solution 13.3

50-layer MLP with ReLU and He init: gradient norms by layer are roughly constant, decreasing by <2x from output to input. Replace ReLU with sigmoid: gradients shrink by ~4x per layer, so layer-1 gradient is ~10^{-30} of layer-50, completely vanished. Training is impossible without batch norm or residual connections.

Solution 13.4

Gradient checkpointing trades compute for memory: instead of storing all activations, store only checkpoints and recompute the activations during backward. For a 100-layer network, memory drops by sqrt(N) ≈ 10x, compute increases by ~33%. In PyTorch:

```
import torch.utils.checkpoint as cp
def forward(x):
    return cp.checkpoint(layer1, x)  # recompute layer1 in backward
```

Critical for training large transformers (LLaMA, GPT) that would otherwise OOM.

Chapter 14: Initialization and Conditioning

Solution 14.1

30-layer MLP on MNIST: with zeros init, all neurons compute the same function (symmetry not broken); the network is equivalent to a single neuron and fails to learn nonlinear patterns. Xavier init (var = 2/(fan_in + fan_out)) trains slowly with ReLU because it was derived for tanh. He init (var = 2/fan_in) is designed for ReLU and trains the network to ~97% test accuracy. The improvement from He over Xavier is most pronounced in deep ReLU networks.

Solution 14.2

On CIFAR-10 with a 10-layer convnet: without BN, training requires careful learning rate and is unstable; reaches ~80% accuracy after long training. With BN, training is stable across learning rates 0.001 to 0.1, converges 3-5x faster, and reaches ~85% accuracy. BN both regularizes and acts as a learnable normalization, decoupling layer training.

Solution 14.3

Without residual connections, training a 50-layer plain MLP is difficult; 100 layers is essentially untrainable; 200 layers shows no improvement over random. With residual connections, all three depths train successfully, with deeper networks achieving marginally better accuracy. ResNet's success at 152 layers demonstrated this for ImageNet; the same principle applies at smaller scale.

Chapter 15: Regularization from a Numerical View

Solution 15.1

```
import numpy as np
rng = np.random.default_rng(0)
n, p = 100, 20
X = rng.standard_normal((n, p))
X[:, 1] = X[:, 0] + 1e-3 * rng.standard_normal(n)   # near-collinear
true_w = np.zeros(p); true_w[0] = 1.0
y = X @ true_w + 0.1 * rng.standard_normal(n)
w_ols = np.linalg.lstsq(X, y, rcond=None)[0]
w_ridge = np.linalg.solve(X.T @ X + 0.1 * np.eye(p), X.T @ y)
print(w_ols[:2], w_ridge[:2])
```

OLS gives wildly oscillating coefficients (e.g., +100 and −100) that cancel out. Ridge regression damps these, giving smaller, more stable coefficients. The numerical condition number of $X^TX + \lambda I$ is bounded by $(s_1^2 + \lambda) / (s_p^2 + \lambda)$, much better than X^TX alone.

Solution 15.2

```
import numpy as np
def soft_threshold(x, t): return np.sign(x) * np.maximum(np.abs(x) - t, 0)
def lasso_prox(X, y, lam, lr=0.001, n_iter=10000):
    n, p = X.shape; w = np.zeros(p)
    for _ in range(n_iter):
        g = X.T @ (X @ w - y) / n
        w = soft_threshold(w - lr * g, lr * lam)
    return w
```

For increasing λ, the number of nonzero coefficients decreases monotonically. At very large λ, $w = 0$. Lasso performs implicit feature selection by setting coefficients exactly to zero.

Solution 15.3

MLP on MNIST, hidden dim 256: without dropout, training accuracy reaches ~100% but validation plateaus at ~98%. With dropout 0.5, training accuracy reaches only ~99% but validation reaches ~98.3-98.5%. Dropout prevents overfitting by averaging over an exponential family of subnetworks.

Solution 15.4

On a problem with no regularization other than early stopping: train indefinitely; track validation loss. The optimal stopping point is where validation loss reaches its minimum. Patience = 5 epochs means stop if no improvement for 5 consecutive epochs. Small patience risks stopping too early (under-train); large patience risks past the minimum (over-train). Patience = 10 to 20 is typical for medium-sized models.

Chapter 16: Optimization Pathologies

Solution 16.1

In a 20-layer MLP with sigmoid activations: layer-1 gradient norms are 10^{-18} while layer-20 are 10^{-1}. With ReLU and He init: ratios are within 2×. With BatchNorm and residual connections: all layers within 20%. The visualization (log-y plot of gradient norm vs layer) is the classic diagnostic image for vanishing-gradient pathologies.

Solution 16.2

RNN on a sequence-classification task (length 200): without clipping, gradient norm occasionally explodes (>1000) due to recurrent multiplication; training NaNs. With clip_grad_norm at threshold 1.0: gradients clipped a few percent of the time; training converges stably. The clipping does not hurt convergence (gradient direction is preserved); only magnitude is bounded.

Solution 16.3

On $f(x, y) = x^2 - y^2$: from (1, 0): plain gradient descent moves to (ε, 0) after one step, then converges along x but never along y, since the y-gradient is zero at y = 0. SGD with noise: each step has y-noise of order ησ; the iterate eventually picks up a nonzero y, after which gradient descent on negative curvature drives y away from zero. Saddle escape time scales like $1/\sigma^2$.

Solution 16.4

SAM (Sharpness-Aware Minimization) computes the gradient at the worst-case perturbed point in a ball around the current parameters: $w \leftarrow w - \eta \cdot \nabla L(w + \varepsilon \cdot \nabla L / \|\nabla L\|)$. On CIFAR-10 with ResNet-18: SAM improves test accuracy by 0.5-1 percentage point over SGD baseline. The intuition: SAM finds flatter minima, which generalize better. Cost is 2x training time (two forward-backward per step).

Chapter 17: Numerical Methods in Probability

Solution 17.1

```
import numpy as np
def exp_inv_transform(u, lam): return -np.log(u) / lam   # since F⁻¹(u)= −ln(1−u)/λ; using ln(u) is equivalent since u and 1−u are both uniform
rng = np.random.default_rng(0)
lam = 2.0
for n in [100, 10_000, 1_000_000]:
    u = rng.uniform(0, 1, n)
    x = exp_inv_transform(u, lam)
    print(n, x.mean(), 1/lam, abs(x.mean() - 1/lam))
```

Sample mean converges to $1/\lambda = 0.5$ at rate $1/\sqrt{N}$. For $n = 10^6$ the error is around 5×10^{-4}, matching the predicted $\sigma/\sqrt{N} = (1/\lambda)/\sqrt{10^6} = 5\times10^{-4}$.

Solution 17.2

Importance sampling estimator: $\hat{E} = (1/N)\ \Sigma\ f(x_i)\ p(x_i)/q(x_i)$, $x_i \sim q$. Variance scales as σ^2/N where σ^2 depends on the ratio p/q. For $p = N(0, 1)$, $q = N(0, \sigma_q^2)$, and $f(x) = x^2$ (so true $E_p[f] = 1$):

- σ_q very small: tail samples from q never reach the tail of p, weights $w = p/q$ are bounded; low variance there but biased if not enough mass.
- σ_q very large: samples from q rarely hit the bulk of p; weights are very small for most samples and very large for a few; high variance.
- $\sigma_q = 1$: $q = p$, weights are all 1, variance is just the variance of $f(x)$ under $p \approx 2$.
- Optimal σ_q for variance minimization: depends on f. For $f(x) = x^2$ it's around $\sigma_q^2 \approx 3$.

Solution 17.3

Cauchy distribution has density $1/[\pi(1+x^2)]$ and no mean (the integral diverges). Sample mean does not converge: instead, by stability of the Cauchy distribution, the sample mean of N Cauchy variables is itself distributed as the original Cauchy, regardless of N. The empirical mean oscillates wildly; running it for 10^6 samples and plotting the running average shows occasional jumps of order 10^2 from a single extreme sample. The strong law of large numbers requires $E[|X|] < \infty$.

Solution 17.4

Naive: $\hat{E} = (1/N)\ \Sigma\ (U_i-1)^2$ where $U_i \sim$ Uniform(0,1). Antithetic: pair U with $1-U$; estimator is $(1/N)\ \Sigma\ [(U_i-1)^2 + ((1-U_i)-1)^2]/2 = (1/N)\ \Sigma\ [(U_i-1)^2 + U_i^2]/2$.

```
import numpy as np
rng = np.random.default_rng(0); N = 10000
U = rng.uniform(0, 1, N)
naive = ((U - 1)**2).mean()
anti = (((U-1)**2 + U**2) / 2).mean()
# True value E[(U-1)²] = 1/3 ≈ 0.333
print(naive, anti)
```

Antithetic variance is typically 30-50% of naive variance for symmetric integrands like this one.

Chapter 18: Optimization in Probabilistic Models

Solution 18.1

Univariate Gaussian likelihood: $L(\mu, \sigma^2) = \prod (1/\sqrt{(2\pi\sigma^2)}) \exp(-(x_i-\mu)^2/(2\sigma^2))$. Log-likelihood: $\ell = -(N/2)\log(2\pi) - (N/2)\log(\sigma^2) - \Sigma(x_i-\mu)^2/(2\sigma^2)$. Set partial derivatives to zero:

$$\partial\ell/\partial\mu = \Sigma(x_i-\mu)/\sigma^2 = 0 \Rightarrow \hat{\mu} = (1/N)\Sigma\, x_i$$

$$\partial\ell/\partial\sigma^2 = -N/(2\sigma^2) + \Sigma(x_i-\mu)^2/(2\sigma^4) = 0 \Rightarrow \hat{\sigma^2} = (1/N)\Sigma(x_i-\hat{\mu})^2$$

Note: MLE variance is biased (divides by N, not N−1). Unbiased sample variance divides by N −1.

Solution 18.2

Bernoulli policy with parameter p produces sample $x \in \{0, 1\}$ with $P(x = 1) = p$. REINFORCE: $g = (f(x) - \text{baseline}) \cdot \nabla \log p(x)$. Analytical: $\nabla_p E[f(X)] = f(1) - f(0)$. Variance comparison: REINFORCE variance scales with f^2; for $f \in \{0, 1\}$, the variance is finite but much larger than the deterministic gradient. With a baseline (mean of f), variance reduces but still nonzero. Empirically, REINFORCE needs ~100× more samples to achieve the same gradient accuracy as the analytical form.

Solution 18.3

Posterior: 2D Gaussian with mean 0 and covariance Σ with strong off-diagonal. Mean-field q factorizes: $q(\theta_1)q(\theta_2)$, both Gaussian. The $KL(q \,/\!/\, p)$ is minimized by $q_i = N(0, 1/\Sigma^{-1}_\{ii\})$. Note: this is the inverse of the precision matrix diagonal, not the covariance diagonal. For $\Sigma = [[1, 0.9], [0.9, 1]]$: $\Sigma^{-1} = [[5.26, -4.74], [-4.74, 5.26]]$, so q_i has variance $1/5.26 \approx 0.19$, drastically smaller than the marginal variance 1 of the true posterior.

Solution 18.4

VAE on MNIST with 2D latent: encoder outputs (μ, logσ); decoder takes sampled z. ELBO = reconstruction_loss + $KL(q(z|x) \,/\!/\, N(0, I))$. After 50 epochs the 2D latent space organizes digits by class clusters and stroke style. Plotting decoder output on a 2D grid in latent space gives a smooth 'manifold of MNIST', interpolating between digit appearances. The VAE was introduced in Kingma & Welling 2014.

Chapter 19: Bayesian Computation

Solution 19.1

Target: 2D N(0, Σ) with Σ = [[1, 0.9], [0.9, 1]] (strongly correlated). Random-walk Metropolis with isotropic step:

```
import numpy as np
Sigma = np.array([[1.0, 0.9], [0.9, 1.0]])
Sigma_inv = np.linalg.inv(Sigma)
def log_pi(x): return -0.5 * x @ Sigma_inv @ x
rng = np.random.default_rng(0); n_iter = 10000
theta = np.zeros(2); samples = np.zeros((n_iter, 2)); log_pi_curr = log_pi(theta)
for t in range(n_iter):
    prop = theta + 0.5 * rng.standard_normal(2)
    log_pi_prop = log_pi(prop)
    if np.log(rng.uniform()) < log_pi_prop - log_pi_curr:
        theta = prop; log_pi_curr = log_pi_prop
    samples[t] = theta

# Autocorrelation along the correlated direction
def autocorr(x, lag): return np.corrcoef(x[:-lag], x[lag:])[0, 1]
for lag in [1, 5, 50, 200]:
    print(lag, autocorr(samples[:, 0], lag))
```

With isotropic step, the chain explores the long axis of the posterior poorly; autocorrelation at lag 200 is still 0.5+. Better: use a step proposal aligned with Σ (e.g., propose $\theta + \alpha \cdot L\xi$ with L from Cholesky of Σ).

Solution 19.2

Tune step size so acceptance rate is near 0.234. Adaptive scheme: increase step if accept > 0.234, decrease if accept < 0.234.

```
step = 1.0; alpha_target = 0.234
for t in range(n_iter):
    # ... Metropolis step ...
    if accepted: step *= 1.01
    else: step *= 1.01**(-alpha_target/(1-alpha_target))
```

Empirically, optimal step size scales as $2.38^2/d$ times the posterior standard deviation in the slowest direction. For 2D N(0, Σ) with the matrix above, this gives step ≈ 1.7 · 0.3 = 0.5 (where 0.3 ≈ sqrt of smallest eigenvalue of Σ).

Solution 19.3

Bayesian linear regression: $y = X\beta + \varepsilon$, $\varepsilon \sim N(0, \sigma^2 I)$. Prior: $\beta \sim N(0, \sigma^2 \Sigma_0)$, $\sigma^2 \sim$ InvGamma(a, b). Conditionals:

- $\beta \mid \sigma^2, y \sim N(\mu_N, \sigma^2 \Sigma_N)$ with $\Sigma_N^{-1} = \Sigma_0^{-1} + X^T X$, $\mu_N = \Sigma_N X^T y$
- $\sigma^2 \mid \beta, y \sim$ InvGamma$(a + n/2, b + \|y - X\beta\|^2/2)$

Gibbs sampling alternates these draws. Compare with analytical marginal posterior $\beta \mid y \sim$ multivariate-t. Sample means and covariances of β from 10^4 Gibbs samples match the analytical marginal to ~3 decimal places (Monte Carlo error $\propto 1/\sqrt{N}$).

Solution 19.4

Hierarchical 8-schools model in Stan/PyMC: $\tau \sim$ HalfCauchy, $\theta_j \sim N(\mu, \tau)$, $y_j \sim N(\theta_j, \sigma_j)$. With centered parameterization, NUTS produces 50-200 divergent transitions out of 4000 , a serious warning. Non-centered parameterization: $\theta_j = \mu + \tau \cdot \eta_j$ with $\eta_j \sim N(0, 1)$. This

dramatically reduces divergences (often to 0). The reparameterization decouples τ from θ in regions of strong curvature, where the leapfrog integrator otherwise fails.

Chapter 20: Numerical Stability in ML Pipelines

Solution 20.1

```
import numpy as np
def softmax_stable(x):
    m = x.max(); e = np.exp(x - m); return e / e.sum()

for logit_scale in [-1000, -100, 0, 100, 1000]:
    x = np.array([logit_scale, logit_scale+1, logit_scale+2], dtype=np.float32)
    p = softmax_stable(x)
    print(logit_scale, p, p.sum())
```

Output: probabilities are always [0.090, 0.245, 0.665] regardless of overall logit magnitude. The naive form returns NaN for logits above ~89 (overflow) or [0, 0, 0] for logits below ~-103 (underflow).

Solution 20.2

Identity: log(1 + e^x) = log(e^(x/2) (e^(x/2) + e^(-x/2))) which gives various manipulations. The form max(x,0) + log(1 + e^{-|x|}) is derived as follows. For x ≥ 0: log(1 + e^x) = x + log(e^{-x} + 1). For x < 0: log(1 + e^x) = log(1 + e^x) directly. Combining: max(x, 0) + log(1 + e^{-|x|}). The argument of the exponential is always ≤ 0, so it never overflows.

```
def softplus(x): return np.maximum(x, 0) + np.log1p(np.exp(-np.abs(x)))
for x_val in [-1000, -10, 0, 10, 1000]:
    print(x_val, softplus(np.array([x_val], dtype=np.float32)))
```

Solution 20.3

Welford's algorithm:

```
import numpy as np
class Welford:
    def __init__(self): self.n=0; self.mean=0.0; self.M2=0.0
    def update(self, x):
        self.n += 1
        d = x - self.mean; self.mean += d/self.n
        d2 = x - self.mean; self.M2 += d*d2
    @property
    def var(self): return self.M2/(self.n-1) if self.n>1 else 0

x = 1e9 + np.random.default_rng(0).standard_normal(10000)
w = Welford()
for xi in x: w.update(xi)
print(w.var, x.var(ddof=1))   # both ~1.0
```

Welford matches two-pass to ~14 digits. Both far more accurate than the naive $E[X^2] - E[X]^2$ form.

Solution 20.4

MLP on MNIST in float16 without loss scaling: gradients of small magnitude ($< 6 \times 10^{-8}$) round to zero in float16; training stalls. With loss scaling (multiply loss by 2^15 before backward, divide gradients by 2^15 after), gradients remain in representable range. Final accuracy: float16+scaling matches float32 to within 0.1%. Without scaling, float16 reaches only 92% vs 98% baseline.

Chapter 21: Generalization vs Numerical Behavior

Solution 21.1

Linear regression with n = 50 samples and increasing parameter count p: train MSE = 0 once $p \geq n$; test MSE shows the double-descent curve. Test error peaks at p = n (the interpolation threshold) where the unique solution is highly sensitive to noise, then drops as $p \gg n$ and the minimum-norm solution selects a more stable interpolator.

```
for p in [1, 5, 25, 49, 50, 51, 100, 500]:
  X_train = np.random.standard_normal((50, p))
  y_train = ...
  w = np.linalg.lstsq(X_train, y_train, rcond=None)[0]
  # compute train and test MSE
```

Plot of test MSE vs p shows the characteristic 'double descent' shape.

Solution 21.2

Same network (MLP, 256 hidden) on MNIST: batch=8 produces noisy gradients; trajectory in parameter space is jagged; converges in ~10 epochs to 98.0% test acc. Batch=512: smooth gradient; same total compute uses fewer steps; converges in ~5 epochs to 97.5% test acc. The smaller batch's noise acts as implicit regularization, finding slightly flatter minima.

Solution 21.3

Small MLP on a 1D regression task: train without dropout to a minimum; compute Hessian $H = \nabla^2 L$. Compute top-k eigenvalues of H via Lanczos. Without dropout: top eigenvalue ≈ 1000, indicating sharp minimum. With dropout 0.5: top eigenvalue ≈ 200 (5× smaller). Dropout finds flatter minima, which is one mechanism for its regularization effect (cf. Keskar et al. 2017).

Solution 21.4

X is n×p, X^TX is p×p. Condition number of X^TX:

- $p \ll n$ (underparameterized): $\text{rank}(X^TX) = p$; cond number is moderate, scaling with p/n.
- $p \approx n$ (interpolation threshold): X^TX is barely full-rank; cond number diverges (Marchenko-Pastur). This is the source of the test-error spike.
- $p \gg n$ (overparameterized): X^TX is rank n, not p; it's singular as a p×p matrix. But the min-norm solution comes from XX^T (n×n), which is well-conditioned.

The worst conditioning occurs exactly at the interpolation threshold.

Chapter 22: Debugging Numerical Issues

Solution 22.1

```
import torch
max_acts = {}
def hook(name):
    def fn(mod, inp, out):
        max_acts[name] = max(max_acts.get(name, 0), out.abs().max().item())
    return fn
for name, m in model.named_modules():
    if isinstance(m, (torch.nn.Linear, torch.nn.Conv2d)):
        m.register_forward_hook(hook(name))
# train one epoch...
for name, val in max_acts.items(): print(name, val)
```

In a healthy network, max activations grow gradually from input to output. A layer with max_act > 10^3 while neighbors are around 10 signals a problem (often: missing normalization, bad initialization).

Solution 22.2

Custom autograd.Function subclass overrides forward and backward. Test with gradcheck on float64 inputs:

```
from torch.autograd import gradcheck
class MyLoss(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x):
        ctx.save_for_backward(x); return (x**3).sum()
    @staticmethod
    def backward(ctx, grad_out):
        x, = ctx.saved_tensors
        return 3 * x**2 * grad_out

x = torch.randn(5, dtype=torch.float64, requires_grad=True) * 1e-6
print(gradcheck(MyLoss.apply, (x,), eps=1e-6, atol=1e-7))
x = torch.randn(5, dtype=torch.float64, requires_grad=True) * 1e6
print(gradcheck(MyLoss.apply, (x,), eps=1e-2, atol=1e-3))   # larger eps for large x
```

Tolerance must scale with input magnitude. For very small or very large inputs, the central-difference approximation needs $h \propto \sqrt{\varepsilon} \cdot |x|$.

Solution 22.3

Property-based test using hypothesis:

```
from hypothesis import given, strategies as st
import numpy as np
@given(st.lists(st.floats(min_value=-50, max_value=50), min_size=2, max_size=20))
def test_softmax(logits):
    x = np.array(logits)
    p = softmax_stable(x)
    assert abs(p.sum() - 1) < 1e-6
    # shift invariance
    p2 = softmax_stable(x + 100)
    assert np.allclose(p, p2, atol=1e-6)
    # monotonicity
    order = np.argsort(x); order_p = np.argsort(p)
    assert (order == order_p).all()
```

Property-based testing catches edge cases (all equal, very large, very small) automatically.

Solution 22.4

Reproduce: use loss = $-\Sigma$ y_i log(softmax(z)_i) computed naively (compute probabilities first, then log). For logits with one value ≥ 50 in float32, probability of the smallest class underflows to 0; $\log(0) = -\infty$; the loss is NaN. Fix: use F.cross_entropy(logits, target) which uses the stable log-softmax internally. Same training, NaN disappears.

Chapter 23: Performance & Scaling

Solution 23.1

```
import numpy as np, time
rng = np.random.default_rng(0); X = rng.standard_normal((1000, 100)); Y = rng.standard_normal((1000, 100))
# Method 1: Python loop
t = time.time()
D1 = np.zeros((1000, 1000))
for i in range(1000):
    for j in range(1000):
        D1[i,j] = np.sqrt(((X[i]-Y[j])**2).sum())
print('loop', time.time() - t)   # ~30 seconds
# Method 2: Broadcasting
t = time.time()
D2 = np.sqrt(((X[:,None,:] - Y[None,:,:])**2).sum(axis=2))
print('broadcast', time.time() - t)   # ~1 second
# Method 3: matmul identity
t = time.time()
XX = (X*X).sum(axis=1, keepdims=True)
YY = (Y*Y).sum(axis=1, keepdims=True).T
D3 = np.sqrt(np.maximum(XX + YY - 2*X@Y.T, 0))
print('matmul', time.time() - t)   # ~0.01 seconds
```

Speedup: loop → broadcast: 30×. Broadcast → matmul: 100×. Total 3000×.

Solution 23.2

MLP on MNIST: float32 baseline 12s/epoch, 97.8% acc. With torch.cuda.amp.autocast: 6s/epoch, 97.7% acc. Mixed precision halves training time at negligible accuracy cost. Max gradient magnitude: float32 ≈ 0.1, float16 ≈ 0.1 too (loss scaling brings it back). Without loss scaling: gradients underflow, training stalls.

Solution 23.3

Two-process all-reduce via multiprocessing.Pipe or shared memory:

```
from multiprocessing import Process, Pipe
def worker(conn, local_value):
    conn.send(local_value)
    other = conn.recv()
    result = local_value + other
    conn.send(result)

# Master coordinates the exchange and sum
```

For 1MB tensors, naive Python multiprocessing all-reduce is ~5ms (mostly serialization overhead). Production codes use NCCL/MPI which achieve near-bandwidth-limited communication.

Solution 23.4

PyTorch profiler with `with torch.profiler.profile()`. For a small CNN: typically the largest time goes to Conv2d operations (~60-80% on GPU). On CPU, often softmax or batchnorm. Strategies to improve: fuse Conv+BatchNorm+ReLU into a single kernel via torch.compile; use channels_last memory format on GPU for 1.5-2× speedup; use mixed precision.

Chapter 24: Training a Neural Network from Scratch

Solution 24.1

Three hidden layers (256 → 256 → 256 → 10). Without He init (using N(0, 1) instead): gradient norm at first layer is roughly 10^{-9} by the third epoch; training collapses. With He init: gradient norms are within 2× across layers; the network trains to ~98% test accuracy.

Solution 24.2

Replace ReLU with Tanh: tanh' has max value 1 and average ≈ 0.5. Gradients shrink by factor ~2 per layer; for 3 layers, ratio first/last ≈ 0.125. The network still trains but more slowly; reaches ~97% accuracy in 20 epochs vs 10 epochs with ReLU. Vanishing-gradient pathology is visible but not severe at depth 3.

Solution 24.3

Adam vs SGD on the two-layer MLP: Adam (lr=1e-3) reaches 97.5% test acc in 5 epochs. SGD with lr=0.1 reaches 98% test acc in 10 epochs. Adam optimizes faster initially but its noisier dynamics produce slightly worse generalization on this simple problem. Pattern is consistent with empirical findings on image classification.

Solution 24.4

With dropout 0.5: training accuracy drops from 99% to 97%; test accuracy stays at 97.8%. On MNIST, the model has more than enough capacity, so dropout provides no benefit. Dropout helps when the model is at risk of memorizing training data; on MNIST with a 2-layer 128-hidden MLP, the network is already on the right side of the bias-variance trade-off.

Chapter 25: Logistic Regression at Scale

Solution 25.1

Naive form computes log(1 + exp(z)) directly; for z = 100, exp(100) overflows in float32; loss becomes NaN. Stable form: max(z, 0) + log(1 + exp(−|z|)) handles any z. For z = 1000, naive returns NaN, stable returns 1000.0.

Solution 25.2

20 Newsgroups (binary task: sci.med vs sci.space, ~2000 docs, ~30k features after tf-idf):

- L-BFGS (sklearn default): 1.2s, 96.4% test acc, converges in 25 iterations.
- SGD with lr=0.01 (1 epoch): 0.4s, 94.8% test acc.
- Coordinate descent (sklearn liblinear): 0.6s, 96.5% test acc.

For sparse data, coordinate descent is hard to beat. SGD wins when memory dominates.

Solution 25.3

Solve L1-regularized LR for λ = 100, 50, 10, 5, 1, 0.5, 0.1, 0.05, 0.01 (warm-start each from previous). At λ = 100, all coefficients are zero. As λ decreases, coefficients enter one at a time (path is piecewise linear in λ). At small λ (≈0.01) almost all coefficients are nonzero. The 'regularization path' plot of coefficient values vs log(λ) shows the entry order of features.

Solution 25.4

Imbalanced data: 99% class 0, 1% class 1. Uniform: predicts class 0 always, 99% accuracy, 0% recall on class 1. Class weights (1, 99) in loss: rebalances; accuracy 92%, recall on minority class 80%. Oversampling minority class: same as class weights mathematically, but with sampling noise. Class weights are more stable than oversampling.

Chapter 26: PCA on Large Dataset

Solution 26.1

Synthetic 1000×100 matrix with singular values spanning 10^3 to 10^{-3}. Randomized SVD for k=10:

- q = 0: top singular values accurate to 10^{-1} relative error.
- q = 1: 10^{-3} relative error.
- q = 2: 10^{-6} relative error.

Each power iteration roughly squares the spectral gap, so accuracy improves geometrically. q = 2 is typically enough for ML applications.

Solution 26.2

CIFAR-10 flattened: 60000×3072. Full SVD: 4 minutes, peak memory 4 GB. Randomized SVD k=100, q=2: 5 seconds, peak memory 200 MB. Top-100 singular values match to relative error 10^{-6}. Reconstructed images are visually identical.

Solution 26.3

Incremental PCA in batches of 1000 with different orderings (random shuffle 1, shuffle 2, sequential): recovered components differ by 10^{-3} relative error. The principal directions are insensitive to ordering, but exact eigenvector values depend on the order due to the iterative update.

Solution 26.4

Sparse text-document matrix (20 Newsgroups tf-idf, 11314×61188): scipy.sparse.linalg.svds k=50 takes 8 seconds. Randomized SVD via sparse matrix-vector products takes 12 seconds. Lanczos is faster here because the matrix is very sparse (avg 100 nonzeros per row) and Lanczos exploits this efficiently. Randomized SVD wins when matrices are dense and large.

Chapter 27: Real-World ML Failures

Solution 27.1

Train a small classifier with loss = −Σ y log(softmax(z)) computed naively: for examples where the model is very confident in the wrong class, the predicted probability is tiny (e.g., 10^{-8}); log of it produces a very large negative loss. Float32 still gives a finite (huge) loss and gradient. The gradient direction is fine but magnitude is wrong by factors of millions. With log_softmax: the loss for the same example is bounded by the logit magnitude. Training with naive loss biases the model toward those few hard examples; with stable loss, training is balanced.

Solution 27.2

All-bfloat16 training: weights, gradients, optimizer state in bf16. Adam second-moment accumulation underflows because gradient-squared values are very small. Final accuracy: 90% vs 98% with proper mixed precision. The fix: keep optimizer state in float32. This is what PyTorch AMP does by default.

Solution 27.3

```
import numpy as np
items = np.random.standard_normal((10000, 128))
items[42] = 1e-10 * np.random.standard_normal(128)   # near-zero
query = np.random.standard_normal(128)
# Naive cosine similarity
sims = (items @ query) / (np.linalg.norm(items, axis=1) * np.linalg.norm(query))
print(sims.max(), sims.argmax())
# The near-zero item dominates because we divide by a tiny norm
# Fix: epsilon
sims_safe = (items @ query) / (np.linalg.norm(items, axis=1) + 1e-8) / (np.linalg.norm(query) + 1e-8)
```

Without epsilon, near-zero embeddings produce spurious 'most similar' results. With epsilon = 10^{-8}, those items have similarity near zero, as expected.

Solution 27.4

Train MLP on MNIST; calibrate int8 quantization using validation set. Test on:

- Standard MNIST test: 97.5% (vs 98% float32, 0.5% drop).
- MNIST with Gaussian noise ($\sigma = 0.5$): float32 gets 75%, int8 gets 45%. Activations exceed the calibrated range; int8 saturates and degrades.
- Out-of-distribution images: similar degradation pattern.

Defensive practice: calibrate on a distribution that includes the operating tail, not just clean validation data. Use percentile-based clipping (e.g., 99.9th percentile) rather than min/max.

On Using These Solutions

The solutions provided here are reference points, not the only correct answers. Many exercises admit multiple valid approaches: different programming styles, different proofs of the same result, different choices of hyperparameters. What matters is that your solution arrives at a quantitatively and qualitatively correct answer, with clear reasoning. If your code produces a meaningfully different result from the reference, the gap is itself a learning opportunity: trace through your reasoning, the reference's reasoning, and identify where they diverge.

Exercises marked as open-ended (e.g., 'compare X and Y') reward exploration. The 'expected

outcome' described here is what a careful student would report; your specific numbers will differ depending on random seeds, hardware, and data. The patterns, however, should match.

Appendix F: Glossary

A concise reference of terms used throughout the book. Definitions are tuned for the numerical-methods-for-ML context; in other fields the same word may carry slightly different connotations.

Numerical Foundations

Catastrophic cancellation. Loss of significant digits when subtracting two nearly equal floating-point numbers. The classic source of mysteriously incorrect variance and gradient calculations.

Condition number. Quantifies how sensitive the output of a problem is to perturbations of the input. For a matrix A, κ(A) = σ_max(A) / σ_min(A). Large κ means small input errors produce large output errors.

Forward error. Difference between the computed answer and the true answer. What users care about, but often impossible to compute directly because the true answer is unknown.

Backward error. Smallest perturbation of the input that would make the computed answer exact. The standard target for analyzing numerical algorithms; algorithms with small backward error are called backward stable.

Floating-point. Representation of real numbers as sign × mantissa × base^exponent. IEEE 754 specifies float16, float32, float64 formats used in ML.

Machine epsilon. Smallest ε such that $1 + \varepsilon \neq 1$ in the given format. About 1.2×10^{-7} for float32 and 2.2×10^{-16} for float64.

Truncation error. Error from replacing an exact mathematical operation (e.g., a derivative or infinite series) with a finite approximation. Decreases as the discretization gets finer.

Rounding error. Error from representing a real number with finite precision. Bounded by machine epsilon for normalized values.

Linear Algebra

Cholesky factorization. Decomposition of a symmetric positive definite matrix M as L L^T where L is lower triangular with positive diagonal. Twice as fast as LU and the preferred solver for SPD systems.

LU factorization. Decomposition of a square matrix A as PA = LU with L lower triangular, U upper triangular, P a permutation. The basis of direct linear-system solvers.

QR factorization. Decomposition A = QR with Q orthogonal (or with orthonormal columns) and R upper triangular. Used for least squares and for orthonormal bases.

SVD (Singular Value Decomposition). Decomposition A = UΣV^T with U, V orthogonal and Σ diagonal with non-negative entries. The most informative matrix factorization, used for PCA, low-rank approximation, and pseudo-inverse computation.

Eigenvalue / Eigenvector. Av = λv: the matrix scales v by λ along its direction. Real for symmetric matrices, possibly complex otherwise.

Positive definite (PD). Symmetric matrix A with x^T A x > 0 for every nonzero x. Equivalent to all eigenvalues positive. Hessians of strictly convex functions are PD.

Spectral radius. Maximum |λ| over all eigenvalues. Controls iteration matrix convergence in

Jacobi, Gauss-Seidel, and related methods.

Krylov subspace. Span of $\{v, Av, A^2v, ...\}$. Underlies conjugate gradient, Lanczos, GMRES, and other iterative methods.

Sparse matrix. A matrix with mostly zero entries. Stored as CSR or CSC to save memory and accelerate matrix-vector products. Critical in large-scale ML.

Optimization

Gradient. Vector of partial derivatives ∇f. Points in the direction of steepest ascent.

Hessian. Matrix of second partial derivatives. Describes local curvature; its eigenvalues determine whether a critical point is a minimum, maximum, or saddle.

Convex function. f satisfies $f(tx + (1-t)y) \leq tf(x) + (1-t)f(y)$. All local minima are global minima; tractable to optimize.

Saddle point. Critical point ($\nabla f = 0$) where the Hessian has both positive and negative eigenvalues. Gradient descent escapes saddles slowly but reliably.

Momentum. Optimization technique that accumulates a velocity $v = \beta v + \nabla f$ and steps along v. Accelerates convergence on ill-conditioned problems.

Adam. Adaptive gradient method combining momentum with per-parameter learning rate scaling based on running average of squared gradients.

L-BFGS. Limited-memory quasi-Newton method that approximates the inverse Hessian from gradient history. Standard for convex problems of moderate size.

KKT conditions. Karush-Kuhn-Tucker conditions: necessary conditions for optimality in constrained problems, generalizing Lagrange multipliers to inequality constraints.

Stochastic gradient descent (SGD). Gradient descent using a noisy gradient estimate from a mini-batch of data. The workhorse of large-scale ML.

Learning rate. Step-size scalar η in update $w \leftarrow w - \eta \nabla f$. The single most important hyperparameter in deep learning.

Deep Learning

Backpropagation. Reverse-mode automatic differentiation applied to neural networks. Computes ∇L in time linear in the forward pass.

Vanishing gradient. Gradients shrink toward zero as they propagate back through layers. Causes early layers to stop training. Mitigated by ReLU, BatchNorm, residual connections.

Exploding gradient. Gradients grow unboundedly during backprop. Causes NaN losses. Mitigated by gradient clipping, careful initialization.

Xavier / He initialization. Schemes for initializing neural network weights to maintain activation variance across layers. Xavier for tanh, He for ReLU.

Batch normalization (BatchNorm). Layer that normalizes activations across the batch dimension. Stabilizes training and accelerates convergence.

Layer normalization (LayerNorm). Layer that normalizes across the feature dimension within a single example. Used in transformers and RNNs.

Residual connection. $y = x + f(x)$. Lets gradients flow through identity paths, enabling training

of very deep networks.

Dropout. Regularization that zeros a random fraction of activations during training. Equivalent to averaging over an exponential family of subnetworks.

Softmax. Function mapping a vector of logits to a probability distribution: $p_i = \exp(x_i) / \Sigma \exp(x_j)$. Implement with the max-subtraction trick to avoid overflow.

Log-sum-exp (LSE). $\text{lse}(x) = \log \Sigma \exp(x_i)$. Computed stably as $\max(x) + \log \Sigma \exp(x - \max(x))$. Underlies stable cross-entropy and partition functions.

Probability and Sampling

Monte Carlo. Estimation of expectations or integrals by averaging over random samples. Converges at rate $1/\sqrt{N}$ regardless of dimension.

Importance sampling. Variance-reduction technique: sample from q, reweight by p/q. Variance explodes if q has thinner tails than p.

Markov Chain Monte Carlo (MCMC). Family of algorithms that draw samples from a target by simulating a Markov chain whose stationary distribution is the target.

Metropolis-Hastings. MCMC algorithm: propose $x' \sim q$, accept with probability $\min(1, \pi(x')q(x|x') / \pi(x)q(x'|x))$.

Hamiltonian Monte Carlo (HMC). MCMC variant that uses gradients of $\log \pi$ to propose long, well-mixing trajectories via simulated Hamiltonian dynamics.

Variational inference (VI). Approximate Bayesian inference by minimizing $KL(q \| p)$ over a tractable family q. The ELBO is the objective.

Reparameterization trick. Express $x \sim p_\theta$ as $x = g(\varepsilon; \theta)$ with ε from a parameter-free distribution, enabling pathwise gradients of expectations.

ELBO (Evidence Lower Bound). $E_q[\log p(x, z)] - E_q[\log q(z)]$. Maximizing it minimizes $KL(q \| \text{posterior})$.

Stability and Engineering

Mixed precision. Training scheme that stores weights and optimizer state in float32 but performs forward/backward in float16 or bfloat16. Gives 2-4× speedup with negligible accuracy loss.

Loss scaling. Multiplying the loss by a constant before backward to keep gradients in the representable range of float16. Necessary for stable mixed-precision training.

Gradient clipping. Bounding the global gradient norm to a threshold. Prevents exploding-gradient training failures.

Gradient checking. Comparing analytical gradients to central-difference approximations. The reliable way to verify a hand-coded gradient is correct.

Quantization. Storing weights or activations in low-precision integer formats (int8, int4) for inference. Yields large speedups at small accuracy cost.

Tensor core. Hardware unit on modern GPUs that performs small-block matrix multiplications in mixed precision at high throughput.

Generalization

Overfitting. Model fits training data well but generalizes poorly to new data. Symptom of insufficient regularization or excessive capacity at the wrong scale.

Double descent. Phenomenon where test error rises as model capacity approaches the training set size, then falls again with further capacity. Defies classical bias-variance intuition.

Interpolation threshold. Smallest model capacity that can fit training data exactly. Where double descent's test error peak occurs.

Implicit regularization. Tendency of SGD and related methods to find solutions with particular structure (small norm, flat minima) even without explicit regularizers.

Flat vs sharp minima. Geometric description of local optima. Flat minima generalize better; SGD with noise prefers them.

Appendix G: Chapter Summary Sheets

A one-page reference for each chapter, designed for review or quick lookup. Each sheet lists the chapter's key formulas, when-to-use criteria, and the most common pitfalls in production code.

Chapter 1: Why Numerical Methods Matter

Core idea: ML algorithms are mathematically clean in continuous arithmetic; numerical issues arise from finite-precision floating-point execution.

Key formula: Relative error ≈ $\varepsilon_{machine} \cdot \kappa(\text{problem}) \cdot$ backward error of algorithm

Common pitfalls: (1) Computing log of probability directly; (2) variance via $E[X^2]-E[X]^2$; (3) softmax without max-subtraction; (4) treating warnings as cosmetic.

Take-home: Stable formulations cost nothing to write the first time and save weeks of debugging later.

Chapter 2: Errors, Stability, and Conditioning

Core idea: Decompose 'wrong answer' into (problem sensitivity) × (algorithm error). Both must be small for the result to be accurate.

Key formula: forward error $\lesssim$ condition number × backward error

When to worry: Condition number above 10^8 for float32, above 10^{16} for float64 means precision is exhausted.

Pitfalls: Confusing condition number (problem property) with backward error (algorithm property).

Chapter 3: Linear Algebra Refresher

Core idea: Numerical linear algebra is different from textbook linear algebra: norms matter, orthogonality matters, factorization choice matters.

Key facts: Orthogonal Q preserves 2-norm. SPD matrices admit unique Cholesky. Condition number of A^T A is condition(A)².

Pitfalls: Forming X^T X for least squares squares the condition number; use QR or SVD instead.

Chapter 4: Optimization Landscape

Core idea: ML loss surfaces are non-convex; saddle points dominate in high dimensions; flat minima generalize better than sharp ones.

Key fact: In d dimensions, a random critical point has all-positive Hessian eigenvalues with probability ~2^(-d). Almost all critical points are saddles.

Pitfalls: Treating non-convexity as a fatal obstacle. In practice, gradient methods find good solutions despite non-convexity.

Chapter 5: Gradient-Based Methods

Update rule: $w \leftarrow w - \eta \nabla L(w)$

Step size: Optimal lr is roughly $2/(\lambda_max + \lambda_min)$ for quadratic with eigenvalues in $[\lambda_min, \lambda_max]$. Run LR range test to find it empirically.

Convergence rate: $(\kappa-1)/(\kappa+1)$ per step where κ is condition number. Slow on ill-conditioned problems.

Pitfalls: Too-large lr diverges; too-small lr crawls. The sweet spot is one decade below the divergence threshold.

Chapter 6: Advanced Gradient Methods

Momentum: $v \leftarrow \beta v + \nabla L$; $w \leftarrow w - \eta v$. Typical $\beta = 0.9$. Accelerates convergence to $O(\sqrt{\kappa})$ iterations.

Adam: Adaptive per-parameter learning rate via running averages of g and g^2. Defaults: lr=1e-3, β_1=0.9, β_2=0.999, ε=1e-8.

When to use what: SGD+momentum for classical convex problems; Adam for deep learning; SGD+momentum sometimes generalizes better in vision.

Chapter 7: Second-Order Methods

Newton step: $w \leftarrow w - H^{-1} \nabla L$. Quadratic convergence near optimum but $O(d^3)$ cost per step.

L-BFGS: Approximates H^{-1} from gradient history. $O(md)$ memory for m past gradients. Standard for convex problems with d up to a few million.

Pitfalls: Plain Newton can step toward saddles; saddle-free variants flip sign of negative eigenvalues.

Chapter 8: Constrained Optimization

KKT: Stationarity ($\nabla_x L = 0$), primal feasibility, dual feasibility ($\mu \geq 0$), complementary slackness ($\mu_i g_i = 0$).
Projected gradient: $w \leftarrow \Pi_C(w - \eta \nabla f)$. Projection step enforces constraint.
Pitfalls: L2 regularization is constrained optimization in disguise: λ controls the constraint radius.

Chapter 9: Solving Linear Systems

Direct vs iterative: Direct (LU, Cholesky) for dense or small. Iterative (CG, GMRES) for large sparse.
Decision tree: SPD + sparse → CG. SPD + dense small → Cholesky. General → LU with partial pivoting.
Pitfalls: Don't form A^(-1) explicitly; always factor and back-substitute.

Chapter 10: Matrix Factorizations

Cost: LU and QR: $O(n^3)$. Cholesky: $O(n^3/3)$. SVD: $O(\min(m,n) \cdot mn)$. Eigendecomposition: $O(n^3)$.
Stability: All standard implementations are backward stable. Forward error depends on condition number.
Pitfalls: Classical Gram-Schmidt loses orthogonality; use Modified Gram-Schmidt or Householder.

Chapter 11: Eigenvalue Problems and PCA

Power iteration: $v \leftarrow Av$; $v \leftarrow v/\|v\|$. Converges to dominant eigenvector at rate $|\lambda_2/\lambda_1|$.
PCA: Compute SVD of centered data matrix X. Top-k right singular vectors are the principal directions.
Pitfalls: Always center before PCA. Don't form the covariance matrix; SVD on X is more stable.

Chapter 12: Large-Scale Linear Algebra

Sparse storage: CSR / CSC for general; banded for stencils. Memory savings can be 100× or more.
Mixed precision: Forward in fp16/bf16, master copy of weights in fp32. 2-4× speedup standard.
Pitfalls: Mean-subtracting destroys sparsity. Scale-only normalization preserves it.

Chapter 13: Backpropagation as a Numerical Algorithm

Cost: Backward pass is roughly 2× the forward pass. Memory holds activations between forward and backward.
Verification: Gradient checking with central differences and $h \approx \varepsilon^{(1/3)}$.
Pitfalls: Storing activations in low precision; numerical drift between forward and backward graphs.

Chapter 14: Initialization and Conditioning

He init: Var(W) = 2/fan_in for ReLU networks.
Xavier init: Var(W) = 2/(fan_in + fan_out) for tanh networks.
BatchNorm: Normalizes activations across batch. Decouples layers, stabilizes training, allows higher learning rates.

Chapter 15: Regularization from a Numerical View

Ridge (L2): Adds λI to X^T X. Eigenvalues shifted up by λ; condition number bounded.
Lasso (L1): Produces sparse solutions. Solve via proximal gradient or coordinate descent.
Dropout: Multiplicative noise on activations. Implicit ensembling.

Chapter 16: Optimization Pathologies

Vanishing gradients: Symptom: layer-1 gradient norm < 10^{-10}. Fix: ReLU, BatchNorm, residual connections.
Exploding gradients: Symptom: NaN loss after a few epochs. Fix: gradient clipping at norm 1-5.
Saddle plateaus: Symptom: loss flat for many iterations. SGD noise eventually escapes.

Chapter 17: Numerical Methods in Probability

Box-Muller: U_1, U_2 uniform $\to$ $Z = \sqrt{(-2 \ln U_1)} \cdot \cos(2\pi U_2)$ is standard normal.
MC convergence: $O(1/\sqrt{N})$ standard error, dimension-independent. Variance reduction by antithetic, control variates, or QMC.
Pitfalls: Importance sampling weight collapse when q has thinner tails than p.

Chapter 18: Optimization in Probabilistic Models

MLE: θ̂ = argmax_θ Σ log p(x_i | θ). For exponential families, gradient is empirical minus expected sufficient statistics.
ELBO: E_q[log p(x, z)] − E_q[log q(z)]. Maximize via gradient methods with reparameterization.
Pitfalls: Score-function (REINFORCE) gradients have high variance; prefer reparameterization.

Chapter 19: Bayesian Computation

Random-walk MH: Optimal acceptance rate ≈ 0.234 for high-dimensional Gaussian targets.
HMC / NUTS: Use gradient of log π. Gold standard for general Bayesian inference at moderate scale.
Diagnostics: R-hat < 1.01. ESS / n_iter > 0.1. Zero divergences.

Chapter 20: Numerical Stability in ML Pipelines

Stable softmax: softmax(x) = exp(x − max x) / Σ exp(x − max x).
LSE: lse(x) = max(x) + log Σ exp(x − max(x)).
BCE with logits: max(z, 0) − z y + log(1 + exp(−|z|)).
Pitfalls: Computing softmax probabilities, then log. Always go directly from logits.

Chapter 21: Generalization vs Numerical Behavior

Double descent: Test error peaks at interpolation threshold ($p \approx n$), then falls again as $p \gg n$.
Noise as regularization: SGD noise prefers flat minima; flat minima generalize better.
Implicit regularization: Optimizer choice (Adam vs SGD) shapes which solution is found in the overparameterized regime.

Chapter 22: Debugging Numerical Issues

NaN diagnosis: Forward hooks logging max activation per layer. PyTorch torch.autograd.set_detect_anomaly(True) for backward issues.

Gradient check: atol 1e-6 for float64, 1e-3 for float32. Step size h ≈ ε^(1/3).

Property tests: Outputs sum to 1, output invariant to shift, output monotonic in input.

Chapter 23: Performance and Scaling

Vectorize first: Replace Python loops with NumPy. Replace NumPy loops with BLAS calls.
Mixed precision: torch.cuda.amp.autocast + GradScaler for fp16, or bfloat16 on hardware that supports it.
Distributed: Data parallelism (all-reduce gradients). Model parallelism (split layers). Pipeline parallelism (split by depth).

Chapter 24: Training a Neural Network from Scratch

Sanity checks: (1) Initial loss ≈ log K for K-class. (2) Overfit a tiny dataset. (3) Gradient check on small instance. (4) Watch per-layer activation stats.
Pitfalls: Skipping sanity checks; tuning hyperparameters before the implementation is verified.

Chapter 25: Logistic Regression at Scale

Solver choice: Small dense: Newton. Medium dense: L-BFGS. Large sparse: coordinate descent or SAGA. Very large: distributed L-BFGS.

Stability: Always use stable softplus $\log(1 + \exp(z))$.

Pitfalls: Mean-subtracting sparse data; not warm-starting across regularization path.

Chapter 26: PCA on Large Dataset

Truncated SVD: scipy.sparse.linalg.svds for sparse. Randomized SVD with q=2 power iterations for dense.
Incremental PCA: For streaming or out-of-memory data.
Pitfalls: Forgetting to center. Using full SVD when truncated suffices.

Chapter 27: Real-World ML Failures

Pattern: Silent failures dominate. Loss looks fine; metrics look fine; deployment fails on long-tail inputs.

Defenses: Stable formulations everywhere. Monitor activation ranges in production. Calibrate quantization on tail-inclusive distributions.

Lesson: The cost of writing stable code is a fraction of the cost of debugging it later.

End of Book

Numerical Methods for Machine Learning:

Optimization, Stability & Algorithms

by Mir Hossain

Thank you for using this workbook! Your feedback helps make future editions better for every student.