DEEP LEARNING 101

A Comprehensive Guide to Deep Learning Theory and Practice

VU HUNG NGUYEN

First Edition: 15 Oct 2025

Deep Learning 101

Nguyễn Vũ Hưng – 阮武興

A Comprehensive Guide to
Deep Learning Theory and Practice

18th October 2025

Deep Learning 101

First Edition: 18th October 2025

Preface

Dear ML/AI learners,

Welcome to *Deep Learning 101*, a comprehensive guide designed to take you from the fundamentals to advanced concepts in deep learning. This book is crafted for students, researchers, and practitioners who want to build a solid foundation in one of the most transformative fields of our time.

I started this book as study notes for myself, because I couldn't find any book that fit my background. As I delved deeper into the field, I realised that many existing resources were either too basic or too advanced, leaving a gap for learners with a technical foundation seeking comprehensive yet accessible coverage.

This book is a "compressed version" of the deep learning books out there—you'll find the most important concepts and mathematical formulae presented in a structured, coherent manner. Rather than overwhelming you with every detail, I've distilled the essential knowledge that forms the foundation of modern deep learning.

This book assumes you have a fair understanding of mathematics, algorithms, and programming. But don't worry—you'll learn along the way if you miss any of them. The early chapters provide mathematical foundations, and examples throughout the book will help reinforce concepts you may need to review.

The target audience includes those who want to learn the basics of deep learning with a focus on mathematics. If you're interested in research or want to dive a little deeper into the maths and algorithms underlying neural networks, this book will guide you through both theory and practice.

Whether you're just beginning your journey into machine learning or looking to deepen your understanding of neural networks, this book provides the mathematical foundations, practical insights, and hands-on knowledge you need to succeed.

I hope this resource serves you well in your learning journey.

Best regards,
Vu Hung Nguyen

Contact Information:
GitHub: https://github.com/vuhung16au
LinkedIn: https://www.linkedin.com/in/nguyenvuhung/

Website: `https://vuhung16au.github.io/`

Contents

Acknowledgements

I would like to express my deepest gratitude to the deep learning community for their invaluable contributions to this rapidly evolving field. Special thanks to the pioneers whose groundbreaking work laid the foundation for modern deep learning.

I am grateful to my colleagues, students, and collaborators who have provided feedback, suggestions, and encouragement throughout the writing of this book. Your insights have been instrumental in shaping the content and presentation of this material.

This book draws inspiration from the excellent resources available in the deep learning community, including the seminal work by Goodfellow, Bengio, and Courville, as well as the emerging literature on understanding deep learning.

I would also like to thank my family for their unwavering support and patience during the many hours spent writing and revising this manuscript.

Finally, I am grateful to all readers who engage with this material and contribute to the advancement of deep learning through their research, applications, and education.

Any errors or omissions in this book are entirely my own responsibility.

Vu Hung Nguyen
18th October 2025

Notation

This book uses the following notation throughout:

General Notation

- a, b, c — scalars (lowercase italic letters)
- $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}$ — vectors (bold lowercase letters)
- $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}$ — matrices (bold uppercase letters)
- $\mathcal{A}, \mathcal{B}, \mathcal{C}$ — sets (calligraphic uppercase)
- a_i — the i-th element of vector $\boldsymbol{a}$
- A_{ij} or $\boldsymbol{A}_{ij}$ — element at row i, column j of matrix $\boldsymbol{A}$
- $\boldsymbol{A}^\top$ — transpose of matrix $\boldsymbol{A}$
- $\boldsymbol{A}^{-1}$ — inverse of matrix $\boldsymbol{A}$
- $\|\boldsymbol{x}\|$ — norm of vector $\boldsymbol{x}$ (typically L^2 norm)
- $\|\boldsymbol{x}\|_p$ — L^p norm of vector $\boldsymbol{x}$
- $|x|$ — absolute value of scalar x
- $\mathbb{R}$ — set of real numbers
- $\mathbb{R}^n$ — n-dimensional real vector space
- $\mathbb{R}^{m \times n}$ — set of real $m \times n$ matrices

Probability and Statistics

- $P(X)$ — probability distribution over discrete variable X
- $p(x)$ — probability density function over continuous variable x
- $P(X = x)$ or $P(x)$ — probability that X takes value x
- $P(X|Y)$ — conditional probability of X given Y
- $\mathbb{E}_{x\sim P}[f(x)]$ — expectation of $f(x)$ with respect to distribution P
- $\text{Var}(X)$ — variance of random variable X
- $\text{Cov}(X, Y)$ — covariance of random variables X and Y
- $\mathcal{N}(\mu, \sigma^2)$ — Gaussian distribution with mean μ and variance σ^2

Calculus and Optimization

- $\frac{dy}{dx}$ or $\frac{\partial y}{\partial x}$ — derivative of y with respect to x
- $\nabla_{\boldsymbol{x}} f$ — gradient of function f with respect to $\boldsymbol{x}$
- $\nabla^2 f$ or $\boldsymbol{H}$ — Hessian matrix (matrix of second derivatives)
- $\arg\min_x f(x)$ — value of x that minimizes $f(x)$
- $\arg\max_x f(x)$ — value of x that maximizes $f(x)$

Machine Learning

- $\mathcal{D}$ — dataset
- $\mathcal{D}_{\text{train}}$ — training dataset
- $\mathcal{D}_{\text{val}}$ — validation dataset
- $\mathcal{D}_{\text{test}}$ — test dataset
- n — number of examples in dataset
- m — mini-batch size

- $\boldsymbol{x}$ — input vector or feature vector
- y — target or label
- $\hat{y}$ — prediction or estimated output
- $\boldsymbol{\theta}$ or $\boldsymbol{w}$ — parameters or weights
- $\mathcal{L}$ — loss function
- J — cost function (sum or average of losses)
- α or η — learning rate
- λ — regularization coefficient

Neural Networks

- L — number of layers in a neural network
- $n^{[l]}$ — number of units in layer l
- $\boldsymbol{a}^{[l]}$ — activations of layer l
- $\boldsymbol{z}^{[l]}$ — pre-activation values of layer l
- $\boldsymbol{W}^{[l]}$ — weight matrix for layer l
- $\boldsymbol{b}^{[l]}$ — bias vector for layer l
- f or σ — activation function
- g — general function or transformation

Difficulty Legend

💫 **Beginner** Intuition-first explanations; minimal prerequisites.

☒ **Intermediate** Assumes fundamentals; technical details and derivations.

🌟 **Advanced** Research-level topics or heavier mathematics.

Chapter 1

Introduction

This chapter introduces the fundamental concepts of deep learning and its historical context, providing a foundation for understanding the field.

Learning Objectives

After studying this chapter, you will be able to:

1. **Define deep learning** and understand how it differs from traditional machine learning approaches.
2. **Trace the historical development** of neural networks and deep learning, including key milestones and breakthroughs.
3. **Identify major application domains** where deep learning has achieved significant success.
4. **Understand the key factors** that enabled the rise of deep learning in the 21st century.
5. **Recognize the challenges and limitations** of deep learning approaches.

1.1 What is Deep Learning? 💫

Deep learning is a subfield of machine learning that focuses on learning hierarchical representations of data through artificial neural networks with multiple layers. These networks, inspired by the structure and function of the human brain, have revolutionized numerous fields including computer vision, natural language processing, speech recognition, and many others.

1.1.1 The Rise of Deep Learning

The resurgence of neural networks, now known as deep learning, can be attributed to several key factors:

1. **Availability of Large Datasets:** The digital age has produced massive amounts of data, providing the fuel needed to train complex models effectively.
2. **Computational Power:** The advent of Graphics Processing Units (GPUs) and specialized hardware has enabled the training of much larger networks than was previously possible.
3. **Algorithmic Innovations:** Improvements in optimization algorithms, regularization techniques, and network architectures have made it possible to train very deep networks.
4. **Open-Source Software:** Frameworks like TensorFlow, PyTorch, and others have democratized access to deep learning tools.

1.1.2 Key Characteristics

Deep learning differs from traditional machine learning in several important ways:

- **Automatic Feature Learning:** Unlike traditional approaches that require manual feature engineering, deep learning models automatically learn relevant features from raw data.
- **Hierarchical Representations:** Deep networks learn multiple levels of representation, from low-level features (e.g., edges in images) to high-level concepts (e.g., object categories).
- **End-to-End Learning:** Deep learning often enables end-to-end learning, where the entire system is trained jointly rather than in separate stages.
- **Scalability:** Deep learning models can continue to improve with more data and computational resources.

1.1.3 Applications

Deep learning has achieved remarkable success in numerous domains:

Computer Vision: Image classification, object detection, semantic segmentation, facial recognition, and image generation.

Natural Language Processing: Machine translation, sentiment analysis, question answering, text generation, and language understanding.

Speech and Audio: Speech recognition, speaker identification, music generation, and audio synthesis.

Healthcare: Medical image analysis, drug discovery, disease prediction, and personalized medicine.

Robotics: Autonomous navigation, manipulation, and decision-making.

Game Playing: Achieving superhuman performance in complex games like Go, Chess, and video games.

The impact of deep learning extends far beyond these applications, touching virtually every aspect of modern technology and scientific research.

1.2 Historical Context 💫

The history of deep learning is intertwined with the broader history of artificial intelligence and neural networks. Understanding this context helps us appreciate the current state of the field and its future directions.

1.2.1 The Perceptron Era (1940s-1960s)

The foundations of neural networks were laid in the 1940s with the work of Warren McCulloch and Walter Pitts, who created a computational model of a neuron. In 1958, Frank Rosenblatt invented the Perceptron, an algorithm for learning a binary classifier. The Perceptron showed promise but faced significant limitations. In 1969, Marvin Minsky and Seymour Papert's book *Perceptrons* demonstrated that single-layer perceptrons could not solve non-linearly separable problems like XOR, leading to the first "AI winter."

1.2.2 The Backpropagation Revolution (1980s)

The field was revitalized in the 1980s with the rediscovery and popularization of the backpropagation algorithm by David Rumelhart, Geoffrey Hinton, and Ronald Williams. This algorithm enabled the training of multi-layer networks, overcoming the limitations of single-layer perceptrons.
Key developments during this period include:

- Convolutional Neural Networks (CNNs) by Yann LeCun
- Recurrent Neural Networks (RNNs) for sequential data
- Improved optimization techniques

1.2.3 The Second AI Winter (1990s-2000s)

Despite theoretical advances, neural networks fell out of favor in the 1990s due to:

- Limited computational resources
- Difficulty training deep networks (vanishing gradient problem)
- Success of alternative methods like Support Vector Machines (SVMs)
- Lack of large labeled datasets

During this period, the term “deep learning” was coined to distinguish multi-layer neural networks from shallow architectures.

1.2.4 The Deep Learning Renaissance (2006-Present)

The modern era of deep learning began around 2006 with several breakthrough papers:

1. **2006:** Geoffrey Hinton and colleagues introduced Deep Belief Networks (DBNs) and showed that deep networks could be trained using layer-wise pretraining.

2. **2009:** Large-scale GPU computing for neural networks became practical, dramatically reducing training times.

3. **2012:** AlexNet won the ImageNet competition by a large margin, demonstrating the power of deep CNNs trained on GPUs.

4. **2014-2016:** Sequence-to-sequence models and attention mechanisms revolutionized NLP.

5. **2017-Present:** Transformer architectures and large language models like GPT and BERT have achieved unprecedented performance.

1.2.5 Key Milestones

Table 1.1: Major milestones in deep learning history

Year	**Milestone**
1943	McCulloch-Pitts neuron model
1958	Rosenblatt’s Perceptron
1986	Backpropagation popularized
1989	LeCun’s CNN for handwritten digits
1997	LSTM networks introduced
2006	Deep Belief Networks
2012	AlexNet wins ImageNet
2014	Generative Adversarial Networks (GANs)
2017	Transformer architecture
2018	BERT for NLP
2020	GPT-3 and large language models

This historical perspective shows that deep learning is built on decades of research, with periods of both enthusiasm and skepticism. The current success is the result of persistent research, technological advances, and the convergence of multiple enabling factors.

1.3 Fundamental Concepts 💫

Before diving into the technical details, it is essential to understand several fundamental concepts that underpin deep learning.

1.3.1 Learning from Data

At its core, deep learning is about learning from data. Given a dataset $\mathcal{D} = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \ldots, (\boldsymbol{x}_n, y_n)\}$, where $\boldsymbol{x}_i$ represents input features and y_i represents corresponding targets, the goal is to learn a function $f : \mathcal{X} \to \mathcal{Y}$ that maps inputs to outputs.

Definition 1.1 (Supervised Learning)**.** In supervised learning, we have access to labeled examples where both inputs and desired outputs are known. The model learns to predict outputs for new, unseen inputs.

Definition 1.2 (Unsupervised Learning)**.** In unsupervised learning, we only have inputs without explicit labels. The model learns to discover patterns, structure, or representations in the data.

Definition 1.3 (Reinforcement Learning)**.** In reinforcement learning, an agent learns to make decisions by interacting with an environment and receiving rewards or penalties.

1.3.2 The Learning Process

The learning process in deep learning typically involves:

1. **Model Definition:** Specify the architecture of the neural network, including the number of layers, types of layers, and activation functions.

2. **Loss Function:** Define a loss function $\mathcal{L}(\hat{y}, y)$ that measures the discrepancy between predictions $\hat{y}$ and true targets y.

3. **Optimization:** Use an optimization algorithm (typically gradient descent variants) to adjust the model parameters $\boldsymbol{\theta}$ to minimize the loss:

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(f(\boldsymbol{x}_i; \boldsymbol{\theta}), y_i) \tag{1.1}$$

4. **Evaluation:** Assess the model's performance on held-out test data to estimate generalization.

1.3.3 Neural Networks as Universal Approximators

One of the remarkable properties of neural networks is their ability to approximate a wide range of functions.

Theorem 1.4 (Universal Approximation Theorem (informal))**.** *A neural network with a single hidden layer containing a sufficient number of neurons can approximate any continuous function on a compact subset of* $\mathbb{R}^n$ *to arbitrary accuracy.*

While this theorem provides theoretical justification for using neural networks, in practice, deep networks with multiple layers are often more efficient and effective than shallow but wide networks.

1.3.4 Representation Learning

A key advantage of deep learning is automatic feature learning, also known as representation learning.

- **Lower Layers:** Learn simple, general features (e.g., edges, textures in images)
- **Middle Layers:** Combine simple features into more complex patterns (e.g., object parts)
- **Higher Layers:** Learn abstract, task-specific representations (e.g., object categories)

This hierarchical feature learning is what makes deep networks particularly powerful for complex tasks.

1.3.5 Generalization and Overfitting

A critical challenge in machine learning is ensuring that models generalize well to new data.

Definition 1.5 (Overfitting)**.** Overfitting occurs when a model learns the training data too well, including noise and spurious patterns, leading to poor performance on new data.

Definition 1.6 (Underfitting)**.** Underfitting occurs when a model is too simple to capture the underlying patterns in the data, resulting in poor performance on both training and test data.

The goal is to find the right balance between model complexity and generalization ability, often visualized by the bias-variance tradeoff:

$$\text{Expected Error} = \text{Bias}^2 + \text{Variance} + \text{Irreducible Error} \tag{1.2}$$

Understanding these fundamental concepts provides a solid foundation for exploring the technical details of deep learning in subsequent chapters.

1.4 Structure of This Book 💫

This book is organized into three main parts, each building upon the previous one to provide a comprehensive understanding of deep learning.

1.4.1 Part I: Basic Math and Machine Learning Foundation

The first part establishes the mathematical and machine learning foundations necessary for understanding deep learning:

Chapter 2: Linear Algebra Covers vectors, matrices, and operations essential for understanding neural network computations.

Chapter 3: Probability and Information Theory Introduces probability distributions, expectation, information theory concepts, and their relevance to machine learning.

Chapter 4: Numerical Computation Discusses numerical optimization, gradient-based optimization, and computational considerations.

Chapter 5: Classical Machine Learning Algorithms Reviews traditional machine learning methods that provide context and motivation for deep learning approaches.

1.4.2 Part II: Practical Deep Networks

The second part focuses on practical aspects of designing, training, and deploying deep neural networks:

Chapter 6: Deep Feedforward Networks Introduces the fundamental building blocks of deep learning, including multilayer perceptrons and activation functions.

Chapter 7: Regularization for Deep Learning Explores techniques to improve generalization and prevent overfitting.

Chapter 8: Optimization for Training Deep Models Covers modern optimization algorithms and training strategies.

Chapter 9: Convolutional Networks Details architectures specifically designed for processing grid-structured data like images.

Chapter 10: Sequence Modeling Examines recurrent and recursive networks for sequential and temporal data.

Chapter 11: Practical Methodology Provides guidelines for successfully applying deep learning to real-world problems.

Chapter 12: Applications Showcases deep learning applications across various domains.

1.4.3 Part III: Deep Learning Research

The third part delves into advanced topics and current research directions:

Chapter 13: Linear Factor Models Introduces probabilistic models with linear structure.

Chapter 14: Autoencoders Explores unsupervised learning through reconstruction-based models.

Chapter 15: Representation Learning Discusses learning meaningful representations from data.

Chapter 16: Structured Probabilistic Models Covers graphical models and their integration with deep learning.

Chapter 17: Monte Carlo Methods Introduces sampling-based approaches for probabilistic inference.

Chapter 18: Confronting the Partition Function Addresses computational challenges in probabilistic models.

Chapter 19: Approximate Inference Explores methods for tractable inference in complex models.

Chapter 20: Deep Generative Models Examines modern approaches to generating new data samples.

1.4.4 How to Use This Book

This book is designed to accommodate different learning paths:

- **For Beginners:** Start with Part I to build a strong foundation, then proceed sequentially through Part II.
- **For Practitioners:** If you have a solid mathematical background, you may skip or skim Part I and focus on Parts II and III.
- **For Researchers:** Part III provides advanced material relevant to current research directions in deep learning.
- **For Specific Topics:** Each chapter is relatively self-contained, allowing you to focus on topics most relevant to your interests or needs.

Throughout the book, we balance theoretical rigor with practical insights, providing both mathematical foundations and intuitive explanations. Code examples and exercises (when available) help reinforce concepts and develop practical skills.

1.5 Prerequisites and Resources 💫

To get the most out of this book, certain prerequisites are helpful, though not absolutely necessary. This section outlines the assumed background and provides resources for filling any gaps.

1.5.1 Mathematical Prerequisites

While we introduce key concepts in Part I, familiarity with the following topics will be beneficial:

- **Linear Algebra:** Vectors, matrices, eigenvalues, and eigenvectors
- **Calculus:** Derivatives, partial derivatives, chain rule, and basic optimization
- **Probability:** Basic probability theory, random variables, and common distributions
- **Statistics:** Mean, variance, covariance, and basic statistical inference

For readers needing to review these topics, we recommend:

- “Linear Algebra Done Right” by Sheldon Axler
- “All of Statistics” by Larry Wasserman
- Online resources: Khan Academy, MIT OpenCourseWare

1.5.2 Programming and Machine Learning Background

Basic programming knowledge is helpful for implementing and experimenting with the concepts:

- **Python Programming:** Understanding of basic syntax, data structures, and functions
- **NumPy:** Familiarity with array operations is useful
- **Machine Learning Basics:** General understanding of supervised learning, training/test splits, and evaluation metrics

Recommended resources:

- “Python for Data Analysis” by Wes McKinney
- “Hands-On Machine Learning” by Aurélien Géron
- scikit-learn documentation and tutorials

1.5.3 Deep Learning Frameworks

While this book focuses on concepts rather than specific implementations, familiarity with a deep learning framework is valuable:

- **PyTorch:** Popular for research and prototyping
- **TensorFlow/Keras:** Widely used in industry
- **JAX:** Emerging framework for research

Official documentation and tutorials for these frameworks provide excellent hands-on learning opportunities.

1.5.4 Additional Resources

To complement this book, consider exploring:

Classic Textbooks:

- “Deep Learning” by Goodfellow, Bengio, and Courville
- “Pattern Recognition and Machine Learning” by Christopher Bishop

Online Courses:

- Coursera: Deep Learning Specialization (Andrew Ng)

- Fast.ai: Practical Deep Learning for Coders
- Stanford CS231n, CS224n (lecture notes and videos)

Research Papers:

- ArXiv.org for latest research
- Papers with Code for implementations
- Conference proceedings: NeurIPS, ICML, ICLR, CVPR

Community Resources:

- Distill.pub for interactive explanations
- Towards Data Science (Medium)
- Reddit: r/MachineLearning
- Twitter/X: Follow leading researchers

1.5.5 A Note on Exercises

Throughout this book, we include exercises at the end of chapters (when available) to help reinforce understanding. We encourage readers to:

1. Work through exercises actively rather than just reading solutions
2. Implement concepts in code to deepen understanding
3. Experiment with variations to explore the behavior of models
4. Collaborate with others and discuss concepts

Remember that deep learning is best learned through a combination of theoretical understanding and practical experience. Don't be discouraged if some concepts take time to fully grasp—this is normal and part of the learning process.

1.5.6 Getting Help

If you encounter difficulties or have questions:

- Review the notation section and relevant earlier chapters
- Consult the bibliography for additional perspectives
- Engage with online communities for discussions
- Implement toy examples to build intuition

- Be patient—deep learning is a rapidly evolving field with many subtleties

With these prerequisites and resources in mind, you are well-equipped to begin your deep learning journey. Let us now proceed to the mathematical foundations in Part I.

Key Takeaways

Key Takeaways 1

- **Deep learning** is a subset of machine learning that uses neural networks with multiple layers to learn hierarchical representations.
- **Historical milestones** include the perceptron (1950s), backpropagation (1980s), and the deep learning renaissance (2010s) enabled by data, compute, and algorithms.
- **Key success factors** include large datasets, GPU acceleration, improved algorithms, and better understanding of training dynamics.
- **Major applications** span computer vision, natural language processing, speech recognition, game playing, and scientific discovery.
- **Limitations exist**: Deep learning requires substantial data and compute, can be brittle to distribution shifts, and lacks interpretability.

Problems

Easy

Problem 1.7 (Historical Milestones)**.** List three key breakthroughs that enabled the rise of deep learning in the 21st century and explain their significance.
Hint: Consider computational advances, data availability, and algorithmic innovations.

Problem 1.8 (Deep Learning vs Traditional ML)**.** Explain the main difference between deep learning and traditional machine learning approaches in terms of feature engineering.
Hint: Think about automatic feature learning versus manual feature extraction.

Problem 1.9 (Application Domains)**.** Name three real-world domains where deep learning has achieved significant success and briefly describe one application in each domain.
Hint: Consider computer vision, natural language processing, and speech recognition.

Medium

Problem 1.10 (Enabling Factors)**.** Analyse how the availability of large datasets and computational resources (GPUs) together enabled the practical success of deep learning. Why wasn't one factor alone sufficient?
Hint: Consider the computational requirements of training deep networks and the need for diverse training examples.

Problem 1.11 (Challenges and Limitations)**.** Identify two major challenges or limitations of current deep learning approaches and propose potential research directions to address them.
Hint: Think about interpretability, data efficiency, generalisation, or robustness to adversarial examples.

Part I

Basic Math and Machine Learning Foundation

Chapter 2

Linear Algebra

This chapter covers the linear algebra foundations essential for understanding deep learning algorithms. Topics include vectors, matrices, eigenvalues, and linear transformations.

Learning Objectives

After studying this chapter, you will be able to:

1. **Work with vectors and matrices** and perform basic operations including addition, multiplication, and transposition.

2. **Understand linear transformations** and how matrices represent them in neural networks.

3. **Compute eigenvalues and eigenvectors** and understand their significance in optimization and data analysis.

4. **Apply matrix decompositions** such as eigendecomposition and singular value decomposition (SVD).

5. **Calculate norms and distances** in vector spaces, which are fundamental for optimization and loss functions.

6. **Solve linear systems** and understand when solutions exist and are unique.

2.1 Scalars, Vectors, Matrices, and Tensors 💫

Linear algebra provides the mathematical framework for understanding and implementing deep learning algorithms. We begin with the basic objects that form the foundation of this framework.

2.1.1 Scalars

A *scalar* is a single number, in contrast to objects that contain multiple numbers. We typically denote scalars with lowercase italic letters, such as a, n, or x.

Example 2.1. The learning rate $\alpha = 0.01$ is a scalar. The number of training examples $n = 1000$ is also a scalar.

In deep learning, scalars are often real numbers ($a \in \mathbb{R}$), but they can also be integers, complex numbers, or elements of other fields depending on the context.

2.1.2 Vectors

A *vector* is an array of numbers arranged in order. We identify each individual number in the vector by its position in the ordering. We denote vectors with bold lowercase letters, such as $\boldsymbol{x}$, $\boldsymbol{y}$, or $\boldsymbol{w}$.

Definition 2.2 (Vector)**.** A vector $\boldsymbol{x} \in \mathbb{R}^n$ is an ordered collection of n real numbers:

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \tag{2.1}$$

where x_i denotes the i-th element of $\boldsymbol{x}$.

Example 2.3. A feature vector for a house might be:

$$\boldsymbol{x} = \begin{bmatrix} 2000 \\ 3 \\ 2 \\ 50 \end{bmatrix} \tag{2.2}$$

representing square footage, number of bedrooms, number of bathrooms, and age in years.

2.1.3 Matrices

A *matrix* is a 2-D array of numbers, where each element is identified by two indices. We denote matrices with bold uppercase letters such as $\boldsymbol{A}$, $\boldsymbol{W}$, or $\boldsymbol{X}$.

Definition 2.4 (Matrix)**.** A matrix $\boldsymbol{A} \in \mathbb{R}^{m\times n}$ is a rectangular array of real numbers with m rows and n columns:

$$\boldsymbol{A} = \begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1n} \\ A_{21} & A_{22} & \cdots & A_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{m1} & A_{m2} & \cdots & A_{mn} \end{bmatrix} \tag{2.3}$$

where A_{ij} denotes the element at row i and column j.

Example 2.5. A matrix of training examples where each row is a feature vector:

$$\boldsymbol{X} = \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix} \tag{2.4}$$

Here, $\boldsymbol{X} \in \mathbb{R}^{3\times 3}$ contains 3 examples with 3 features each.

2.1.4 Tensors

A *tensor* is an array with more than two axes. While scalars are 0-D tensors, vectors are 1-D tensors, and matrices are 2-D tensors, we typically reserve the term "tensor" for arrays with three or more dimensions.

Definition 2.6 (Tensor)**.** A tensor $\mathcal{A} \in \mathbb{R}^{n_1\times n_2\times\cdots\times n_k}$ is a k-dimensional array where elements are identified by k indices: $\mathcal{A}_{i_1,i_2,\ldots,i_k}$.

Example 2.7. A batch of color images can be represented as a 4-D tensor:

$$\mathcal{X} \in \mathbb{R}^{B\times H\times W\times C} \tag{2.5}$$

where B is the batch size, H and W are height and width, and C is the number of color channels (e.g., 3 for RGB).

2.1.5 Notation Conventions

Throughout this book, we adopt the following conventions:

- Scalars: lowercase italic (a, b, x)
- Vectors: bold lowercase ($\boldsymbol{a}$, $\boldsymbol{x}$, $\boldsymbol{w}$)
- Matrices: bold uppercase ($\boldsymbol{A}$, $\boldsymbol{X}$, $\boldsymbol{W}$)
- Tensors: calligraphic uppercase ($\mathcal{A}$, $\mathcal{X}$)

Understanding these fundamental objects and their properties is essential for working with the mathematical formulations of deep learning algorithms.

2.2 Matrix Operations 💫

Matrix operations form the computational backbone of neural networks. Understanding these operations is crucial for implementing and analyzing deep learning algorithms.

2.2.1 Matrix Addition and Scalar Multiplication

Matrices of the same dimensions can be added element-wise:

Definition 2.8 (Matrix Addition)**.** Given $\boldsymbol{A}, \boldsymbol{B} \in \mathbb{R}^{m \times n}$, their sum $\boldsymbol{C} = \boldsymbol{A} + \boldsymbol{B}$ is defined as:

$$C_{ij} = A_{ij} + B_{ij} \tag{2.6}$$

for all $i = 1, \ldots, m$ and $j = 1, \ldots, n$.

Definition 2.9 (Scalar Multiplication)**.** Given a scalar $\alpha \in \mathbb{R}$ and a matrix $\boldsymbol{A} \in \mathbb{R}^{m \times n}$, the product $\boldsymbol{B} = \alpha \boldsymbol{A}$ is:

$$B_{ij} = \alpha A_{ij} \tag{2.7}$$

2.2.2 Matrix Transpose

The transpose is a fundamental operation that exchanges rows and columns.

Definition 2.10 (Transpose)**.** The transpose of a matrix $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ is a matrix $\boldsymbol{A}^\top \in \mathbb{R}^{n \times m}$ where:

$$(\boldsymbol{A}^\top)_{ij} = A_{ji} \tag{2.8}$$

Properties of transpose:

$$\begin{align}
(\boldsymbol{A}^\top)^\top &= \boldsymbol{A} \tag{2.9}\\
(\boldsymbol{A} + \boldsymbol{B})^\top &= \boldsymbol{A}^\top + \boldsymbol{B}^\top \tag{2.10}\\
(\alpha \boldsymbol{A})^\top &= \alpha \boldsymbol{A}^\top \tag{2.11}\\
(\boldsymbol{A}\boldsymbol{B})^\top &= \boldsymbol{B}^\top \boldsymbol{A}^\top \tag{2.12}
\end{align}$$

2.2.3 Matrix Multiplication

Matrix multiplication is central to neural network computations.

Definition 2.11 (Matrix Multiplication)**.** Given $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ and $\boldsymbol{B} \in \mathbb{R}^{n \times p}$, their product $\boldsymbol{C} = \boldsymbol{A}\boldsymbol{B} \in \mathbb{R}^{m \times p}$ is defined as:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj} \tag{2.13}$$

Example 2.12. Consider the multiplication:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix} \tag{2.14}$$

where $C_{11} = 1 \cdot 5 + 2 \cdot 7 = 19$.

Important properties:

- **Associative:** $(\boldsymbol{A}\boldsymbol{B})\boldsymbol{C} = \boldsymbol{A}(\boldsymbol{B}\boldsymbol{C})$
- **Distributive:** $\boldsymbol{A}(\boldsymbol{B} + \boldsymbol{C}) = \boldsymbol{A}\boldsymbol{B} + \boldsymbol{A}\boldsymbol{C}$
- **Not commutative:** Generally $\boldsymbol{A}\boldsymbol{B} \neq \boldsymbol{B}\boldsymbol{A}$

2.2.4 Element-wise (Hadamard) Product

The element-wise product is denoted by $\odot$ and operates on corresponding elements.

Definition 2.13 (Hadamard Product)**.** Given $\boldsymbol{A}, \boldsymbol{B} \in \mathbb{R}^{m \times n}$, the Hadamard product $\boldsymbol{C} = \boldsymbol{A} \odot \boldsymbol{B}$ is:

$$C_{ij} = A_{ij} B_{ij} \tag{2.15}$$

This operation is common in neural networks, particularly in activation functions and gating mechanisms.

2.2.5 Matrix-Vector Products

When multiplying a matrix by a vector, we can view it as a special case of matrix multiplication:

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \tag{2.16}$$

where $\boldsymbol{A} \in \mathbb{R}^{m \times n}$, $\boldsymbol{x} \in \mathbb{R}^n$, and $\boldsymbol{b} \in \mathbb{R}^m$.
This operation is fundamental in neural networks, where it represents the linear transformation:

$$b_i = \sum_{j=1}^{n} A_{ij} x_j \tag{2.17}$$

2.2.6 Dot Product

The dot product (or inner product) of two vectors is a special case of matrix multiplication:

Definition 2.14 (Dot Product)**.** For vectors $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$, their dot product is:

$$\boldsymbol{x} \cdot \boldsymbol{y} = \boldsymbol{x}^\top \boldsymbol{y} = \sum_{i=1}^{n} x_i y_i \tag{2.18}$$

The dot product has geometric interpretation:

$$\boldsymbol{x} \cdot \boldsymbol{y} = \|\boldsymbol{x}\| \, \|\boldsymbol{y}\| \cos\theta \tag{2.19}$$

where θ is the angle between the vectors.

2.2.7 Computational Complexity

Understanding computational costs is important for efficient implementation:

- Matrix-matrix multiplication $\boldsymbol{A} \in \mathbb{R}^{m \times n}$, $\boldsymbol{B} \in \mathbb{R}^{n \times p}$: $O(mnp)$ operations

- Matrix-vector multiplication: $O(mn)$ operations
- Element-wise operations: $O(mn)$ operations

These operations can be efficiently parallelized on modern hardware (GPUs, TPUs), which is one reason deep learning has become practical.

2.3 Identity and Inverse Matrices 💫

Special matrices play important roles in linear algebra and deep learning. The identity matrix and matrix inverses are among the most fundamental.

2.3.1 Identity Matrix

The identity matrix is the matrix analog of the number 1.

Definition 2.15 (Identity Matrix)**.** The identity matrix $\boldsymbol{I}_n \in \mathbb{R}^{n\times n}$ is a square matrix with ones on the diagonal and zeros elsewhere:

$$\boldsymbol{I}_n = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix} \tag{2.20}$$

Formally, $(\boldsymbol{I}_n)_{ij} = \delta_{ij}$ where δ_{ij} is the Kronecker delta:

$$\delta_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} \tag{2.21}$$

The key property of the identity matrix is:

$$\boldsymbol{I}_n\boldsymbol{A} = \boldsymbol{A}\boldsymbol{I}_n = \boldsymbol{A} \tag{2.22}$$

for any matrix $\boldsymbol{A} \in \mathbb{R}^{n\times n}$.

2.3.2 Matrix Inverse

The inverse of a matrix, when it exists, allows us to solve systems of linear equations.

Definition 2.16 (Matrix Inverse)**.** A square matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ is *invertible* (or *non-singular*) if there exists a matrix $\boldsymbol{A}^{-1} \in \mathbb{R}^{n \times n}$ such that:

$$\boldsymbol{A}^{-1}\boldsymbol{A} = \boldsymbol{A}\boldsymbol{A}^{-1} = \boldsymbol{I}_n \tag{2.23}$$

Example 2.17. The matrix $\boldsymbol{A} = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix}$ has inverse:

$$\boldsymbol{A}^{-1} = \begin{bmatrix} 1 & -1 \\ -1 & 2 \end{bmatrix} \tag{2.24}$$

We can verify: $\boldsymbol{A}\boldsymbol{A}^{-1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \boldsymbol{I}_2$.

2.3.3 Properties of Inverses

If $\boldsymbol{A}$ and $\boldsymbol{B}$ are invertible, then:

$$(\boldsymbol{A}^{-1})^{-1} = \boldsymbol{A} \tag{2.25}$$

$$(\boldsymbol{A}\boldsymbol{B})^{-1} = \boldsymbol{B}^{-1}\boldsymbol{A}^{-1} \tag{2.26}$$

$$(\boldsymbol{A}^{\top})^{-1} = (\boldsymbol{A}^{-1})^{\top} \tag{2.27}$$

2.3.4 Solving Linear Systems

The inverse allows us to solve systems of linear equations. Given $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$, if $\boldsymbol{A}$ is invertible:

$$\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b} \tag{2.28}$$

However, computing inverses is expensive ($O(n^3)$ for dense matrices) and numerically unstable. In practice, we often use more efficient methods like LU decomposition or iterative solvers.

2.3.5 Conditions for Invertibility

A matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ is invertible if and only if:

- Its determinant is non-zero: $\det(\boldsymbol{A}) \neq 0$

- Its columns (and rows) are linearly independent
- It has full rank: $\text{rank}(\boldsymbol{A}) = n$
- Its null space contains only the zero vector

2.3.6 Singular Matrices

Matrices that are not invertible are called *singular* or *degenerate*.

Example 2.18. The matrix $\boldsymbol{A} = \begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix}$ is singular because its rows are linearly dependent (the second row is twice the first). Its determinant is $\det(\boldsymbol{A}) = 4 - 4 = 0$.

Singular matrices arise in deep learning when:

- Features are perfectly correlated
- The model is overparameterized
- Numerical precision issues occur

2.3.7 Pseudo-inverse

For non-square or singular matrices, we can use the Moore-Penrose pseudo-inverse $\boldsymbol{A}^+$, which provides a generalized notion of inversion. The pseudo-inverse is particularly useful in least squares problems and is discussed further in later chapters.

2.3.8 Practical Considerations

In deep learning implementations:

- Avoid explicitly computing matrix inverses when possible
- Use numerically stable algorithms (e.g., QR decomposition, SVD)
- Add regularization to ensure invertibility (e.g., $(\boldsymbol{A}^\top \boldsymbol{A} + \lambda \boldsymbol{I})^{-1}$)
- Leverage optimized linear algebra libraries (BLAS, LAPACK, cuBLAS)

2.4 Linear Dependence and Span 💫

Understanding linear independence and span is crucial for analyzing the capacity and expressiveness of neural networks.

2.4.1 Linear Combinations

A *linear combination* of vectors $\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n$ is any vector of the form:

$$\boldsymbol{v} = a_1\boldsymbol{v}_1 + a_2\boldsymbol{v}_2 + \cdots + a_n\boldsymbol{v}_n \tag{2.29}$$

where $a_1, a_2, \ldots, a_n$ are scalars called *coefficients*.

Example 2.19. If $\boldsymbol{v}_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and $\boldsymbol{v}_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$, then any vector in $\mathbb{R}^2$ can be written as a linear combination:

$$\begin{bmatrix} x \\ y \end{bmatrix} = x\boldsymbol{v}_1 + y\boldsymbol{v}_2 \tag{2.30}$$

2.4.2 Span

Definition 2.20 (Span)**.** The *span* of a set of vectors $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n\}$ is the set of all possible linear combinations of these vectors:

$$\text{span}(\{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n\}) = \left\{ \sum_{i=1}^{n} a_i\boldsymbol{v}_i \;\middle|\; a_i \in \mathbb{R} \right\} \tag{2.31}$$

The span defines all vectors that can be reached by scaling and adding the given vectors.

Example 2.21. In $\mathbb{R}^3$, the span of $\boldsymbol{v}_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$ and $\boldsymbol{v}_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$ is the xy-plane:

$$\text{span}(\{\boldsymbol{v}_1, \boldsymbol{v}_2\}) = \left\{ \begin{bmatrix} x \\ y \\ 0 \end{bmatrix} \;\middle|\; x, y \in \mathbb{R} \right\} \tag{2.32}$$

2.4.3 Linear Independence

Definition 2.22 (Linear Independence)**.** A set of vectors $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n\}$ is *linearly independent* if no vector can be written as a linear combination of the others. Formally, the only solution to:

$$a_1\boldsymbol{v}_1 + a_2\boldsymbol{v}_2 + \cdots + a_n\boldsymbol{v}_n = \boldsymbol{0} \tag{2.33}$$

is $a_1 = a_2 = \cdots = a_n = 0$.

If a set of vectors is not linearly independent, it is *linearly dependent*.

Example 2.23 (Linear Dependence)**.** The vectors $\boldsymbol{v}_1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$, $\boldsymbol{v}_2 = \begin{bmatrix} 2 \\ 4 \end{bmatrix}$ are linearly dependent because $\boldsymbol{v}_2 = 2\boldsymbol{v}_1$.

Example 2.24 (Linear Independence)**.** The standard basis vectors $\boldsymbol{e}_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and $\boldsymbol{e}_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$ are linearly independent.

2.4.4 Basis

Definition 2.25 (Basis)**.** A *basis* for a vector space V is a set of linearly independent vectors that span V. Every vector in V can be uniquely expressed as a linear combination of basis vectors.

Example 2.26 (Standard Basis)**.** The standard basis for $\mathbb{R}^3$ is:

$$\boldsymbol{e}_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad \boldsymbol{e}_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad \boldsymbol{e}_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \tag{2.34}$$

2.4.5 Dimension and Rank

Definition 2.27 (Dimension)**.** The *dimension* of a vector space is the number of vectors in any basis for that space. We write $\dim(V)$ for the dimension of space V.

Definition 2.28 (Rank)**.** The *rank* of a matrix $\boldsymbol{A}$ is the dimension of the space spanned by its columns (column rank) or rows (row rank). For any matrix, column rank equals row rank, so we simply refer to "the rank."

Properties of rank:

- $\text{rank}(\boldsymbol{A}) \leq \min(m, n)$ for $\boldsymbol{A} \in \mathbb{R}^{m \times n}$
- $\text{rank}(\boldsymbol{AB}) \leq \min(\text{rank}(\boldsymbol{A}), \text{rank}(\boldsymbol{B}))$
- $\boldsymbol{A}$ is invertible if and only if $\text{rank}(\boldsymbol{A}) = n$ (full rank)

2.4.6 Column Space and Null Space

Definition 2.29 (Column Space)**.** The *column space* (or *range*) of a matrix $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ is the span of its columns:

$$\text{Col}(\boldsymbol{A}) = \{\boldsymbol{Ax} \mid \boldsymbol{x} \in \mathbb{R}^n\} \tag{2.35}$$

The dimension of the column space is the rank of $\boldsymbol{A}$.

Definition 2.30 (Null Space)**.** The *null space* (or *kernel*) of $\boldsymbol{A}$ is the set of all vectors that map to zero:

$$\text{Null}(\boldsymbol{A}) = \{\boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{Ax} = \boldsymbol{0}\} \tag{2.36}$$

2.4.7 Relevance to Deep Learning

These concepts are fundamental to understanding:

- **Model Capacity:** The expressiveness of a layer depends on the rank of its weight matrix
- **Redundancy:** Linear dependence in features indicates redundant information
- **Dimensionality Reduction:** Methods like PCA seek low-dimensional representations
- **Network Design:** Understanding which transformations are possible with given architectures

2.5 Norms 💫

Norms are functions that measure the size or length of vectors. They are essential for regularization, optimization, and measuring distances in deep learning.

2.5.1 Definition of a Norm

Definition 2.31 (Norm)**.** A function $f : \mathbb{R}^n \to \mathbb{R}$ is a norm if it satisfies the following properties for all $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$ and $\alpha \in \mathbb{R}$:

1. **Non-negativity:** $f(\boldsymbol{x}) \geq 0$, with equality if and only if $\boldsymbol{x} = \boldsymbol{0}$
2. **Homogeneity:** $f(\alpha\boldsymbol{x}) = |\alpha| f(\boldsymbol{x})$

3. **Triangle inequality:** $f(\boldsymbol{x}+\boldsymbol{y}) \leq f(\boldsymbol{x}) + f(\boldsymbol{y})$

We typically denote norms using the notation $\|\boldsymbol{x}\|$.

2.5.2 L^p Norms

The most common family of norms are the L^p norms.

Definition 2.32 (L^p Norm)**.** For $p \geq 1$, the L^p norm of a vector $\boldsymbol{x} \in \mathbb{R}^n$ is:

$$\|\boldsymbol{x}\|_p = \left(\sum_{i=1}^{n} |x_i|^p\right)^{1/p} \tag{2.37}$$

2.5.3 Common Norms

L^1 Norm (Manhattan Distance)

The L^1 norm is the sum of absolute values:

$$\|\boldsymbol{x}\|_1 = \sum_{i=1}^{n} |x_i| \tag{2.38}$$

Example 2.33. For $\boldsymbol{x} = \begin{bmatrix} 3 \\ -4 \\ 2 \end{bmatrix}$, we have $\|\boldsymbol{x}\|_1 = 3 + 4 + 2 = 9$.

The L^1 norm is used in:

- Lasso regularization (encourages sparsity)
- Robust statistics
- Compressed sensing

L^2 Norm (Euclidean Distance)

The L^2 norm is the most common norm, corresponding to Euclidean distance:

$$\|\boldsymbol{x}\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2} = \sqrt{\boldsymbol{x}^\top \boldsymbol{x}} \tag{2.39}$$

Example 2.34. For $\boldsymbol{x} = \begin{bmatrix} 3 \\ -4 \\ 2 \end{bmatrix}$, we have $\|\boldsymbol{x}\|_2 = \sqrt{9+16+4} = \sqrt{29} \approx 5.39$.

The L^2 norm is used in:

- Ridge regularization (weight decay)
- Gradient descent
- Distance metrics

The squared L^2 norm is often used in optimization because it has simpler derivatives:

$$\|\boldsymbol{x}\|_2^2 = \boldsymbol{x}^\top \boldsymbol{x} = \sum_{i=1}^{n} x_i^2 \tag{2.40}$$

L^∞ Norm (Maximum Norm)

The L^∞ norm is defined as:

$$\|\boldsymbol{x}\|_\infty = \max_i |x_i| \tag{2.41}$$

This can be viewed as the limit of L^p norms as $p \to \infty$.

Example 2.35. For $\boldsymbol{x} = \begin{bmatrix} 3 \\ -4 \\ 2 \end{bmatrix}$, we have $\|\boldsymbol{x}\|_\infty = \max(3,4,2) = 4$.

2.5.4 Frobenius Norm

For matrices, the Frobenius norm is analogous to the L^2 norm for vectors.

Definition 2.36 (Frobenius Norm)**.** For a matrix $\boldsymbol{A} \in \mathbb{R}^{m \times n}$:

$$\|\boldsymbol{A}\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} A_{ij}^2} = \sqrt{\text{trace}(\boldsymbol{A}^\top \boldsymbol{A})} \tag{2.42}$$

The Frobenius norm is used for regularizing weight matrices in neural networks.

2.5.5 Unit Vectors and Normalization

A vector with unit norm ($\|\boldsymbol{x}\| = 1$) is called a *unit vector*.

Definition 2.37 (Normalization)**.** To normalize a vector $\boldsymbol{x}$, we divide by its norm:

$$\hat{\boldsymbol{x}} = \frac{\boldsymbol{x}}{\|\boldsymbol{x}\|} \tag{2.43}$$

resulting in a unit vector pointing in the same direction.

Normalization is commonly used in deep learning:

- Batch normalization
- Layer normalization
- Input feature scaling
- Weight normalization

2.5.6 Distance Metrics

Norms induce distance metrics. The distance between vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ is:

$$d(\boldsymbol{x}, \boldsymbol{y}) = \|\boldsymbol{x} - \boldsymbol{y}\| \tag{2.44}$$

Different norms lead to different notions of distance:

- L^1: Manhattan distance (sum of coordinate differences)
- L^2: Euclidean distance (straight-line distance)
- L^∞: Chebyshev distance (maximum coordinate difference)

2.5.7 Regularization in Deep Learning

Norms are central to regularization techniques:

- L^1 **Regularization:** Adds $\lambda \|\boldsymbol{w}\|_1$ to loss, promoting sparsity
- L^2 **Regularization:** Adds $\lambda \|\boldsymbol{w}\|_2^2$ to loss, preventing large weights
- **Elastic Net:** Combines L^1 and L^2: $\lambda_1 \|\boldsymbol{w}\|_1 + \lambda_2 \|\boldsymbol{w}\|_2^2$

Understanding norms and their properties is essential for designing effective regularization strategies and analyzing model behavior.

2.6 Eigendecomposition 💫

Eigendecomposition is a powerful tool for understanding and analyzing linear transformations, with important applications in deep learning.

2.6.1 Eigenvalues and Eigenvectors

Definition 2.38 (Eigenvector and Eigenvalue)**.** An *eigenvector* of a square matrix $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ is a non-zero vector $\boldsymbol{v}$ such that:

$$\boldsymbol{A}\boldsymbol{v} = \lambda \boldsymbol{v} \tag{2.45}$$

where $\lambda \in \mathbb{R}$ (or $\mathbb{C}$) is the corresponding *eigenvalue*.

The eigenvector's direction is preserved under the transformation $\boldsymbol{A}$, with only its magnitude scaled by λ.

Example 2.39. Consider $\boldsymbol{A} = \begin{bmatrix} 3 & 1 \\ 0 & 2 \end{bmatrix}$. We can verify that $\boldsymbol{v}_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ is an eigenvector:

$$\boldsymbol{A}\boldsymbol{v}_1 = \begin{bmatrix} 3 & 1 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 3 \\ 0 \end{bmatrix} = 3\boldsymbol{v}_1 \tag{2.46}$$

So $\lambda_1 = 3$ is an eigenvalue.

2.6.2 Finding Eigenvalues

To find eigenvalues, we solve the *characteristic equation*:

$$\det(\boldsymbol{A} - \lambda \boldsymbol{I}) = 0 \tag{2.47}$$

This gives a polynomial of degree n called the characteristic polynomial, which has n roots (counting multiplicities) in $\mathbb{C}$.

Example 2.40. For $\boldsymbol{A} = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$:

$$\det \begin{bmatrix} 2-\lambda & 1 \\ 1 & 2-\lambda \end{bmatrix} = (2-\lambda)^2 - 1 = \lambda^2 - 4\lambda + 3 = 0 \tag{2.48}$$

Solving gives $\lambda_1 = 3$ and $\lambda_2 = 1$.

2.6.3 Eigendecomposition

If a matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ has n linearly independent eigenvectors, it can be decomposed as:

$$\boldsymbol{A} = \boldsymbol{V}\boldsymbol{\Lambda}\boldsymbol{V}^{-1} \tag{2.49}$$

where:

- $\boldsymbol{V}$ is the matrix whose columns are eigenvectors: $\boldsymbol{V} = [\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n]$
- $\boldsymbol{\Lambda}$ is a diagonal matrix of eigenvalues: $\boldsymbol{\Lambda} = \text{diag}(\lambda_1, \lambda_2, \ldots, \lambda_n)$

This is called the *eigendecomposition* or *spectral decomposition*.

2.6.4 Symmetric Matrices

Symmetric matrices have particularly nice properties.

Theorem 2.41 (Spectral Theorem for Symmetric Matrices). *If $\boldsymbol{A}$ is a real symmetric matrix ($\boldsymbol{A} = \boldsymbol{A}^\top$), then:*

1. *All eigenvalues are real*
2. *Eigenvectors corresponding to different eigenvalues are orthogonal*
3. *$\boldsymbol{A}$ can be decomposed as:*
$$\boldsymbol{A} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top \tag{2.50}$$
where $\boldsymbol{Q}$ is an orthogonal matrix ($\boldsymbol{Q}^\top\boldsymbol{Q} = \boldsymbol{I}$) of eigenvectors.

This decomposition is fundamental in many algorithms, including Principal Component Analysis (PCA).

2.6.5 Properties of Eigenvalues

For a matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}$:

- $\text{trace}(\boldsymbol{A}) = \sum_{i=1}^{n} A_{ii} = \sum_{i=1}^{n} \lambda_i$
- $\det(\boldsymbol{A}) = \prod_{i=1}^{n} \lambda_i$
- If $\boldsymbol{A}$ is invertible, eigenvalues of $\boldsymbol{A}^{-1}$ are $1/\lambda_i$
- Eigenvalues of $\boldsymbol{A}^k$ are λ_i^k

2.6.6 Positive Definite Matrices

Definition 2.42 (Positive Definite)**.** A symmetric matrix $\boldsymbol{A}$ is *positive definite* if for all non-zero $\boldsymbol{x} \in \mathbb{R}^n$:

$$\boldsymbol{x}^\top \boldsymbol{A}\boldsymbol{x} > 0 \tag{2.51}$$

Equivalently, all eigenvalues of $\boldsymbol{A}$ are positive.

Definition 2.43 (Positive Semi-definite)**.** $\boldsymbol{A}$ is *positive semi-definite* if $\boldsymbol{x}^\top \boldsymbol{A}\boldsymbol{x} \geq 0$ for all $\boldsymbol{x}$, i.e., all eigenvalues are non-negative.

Positive definite matrices are crucial in optimization, as they ensure that local minima are global minima for quadratic functions.

2.6.7 Applications in Deep Learning

Eigendecomposition has several important applications:

1. **Principal Component Analysis (PCA):** Finds directions of maximum variance by computing eigenvectors of the covariance matrix.

2. **Optimization:** The Hessian matrix's eigenvalues determine the curvature of the loss surface. Positive definite Hessians indicate convexity.

3. **Spectral Normalization:** Constrains the largest eigenvalue of weight matrices to stabilize training of GANs.

4. **Graph Neural Networks:** Graph Laplacian eigendecomposition defines spectral graph convolutions.

5. **Understanding Dynamics:** Eigenvalues of recurrent weight matrices affect gradient flow and stability.

2.6.8 Computational Considerations

Computing eigendecomposition:

- Full eigendecomposition: $O(n^3)$ for dense matrices

- Power iteration for dominant eigenvector: $O(kn^2)$ for k iterations

- Iterative methods (e.g., Lanczos) for sparse matrices

For large-scale deep learning, we often use:

- Approximations (e.g., power iteration)
- Focus on top-k eigenvalues/eigenvectors
- Specialized algorithms for specific structures (e.g., symmetric, sparse)

Understanding eigendecomposition provides insight into the geometric properties of linear transformations and is essential for many advanced deep learning techniques.

Key Takeaways

Key Takeaways 2

- **Vectors and matrices** are fundamental building blocks for representing data and transformations in neural networks.
- **Matrix operations** (multiplication, transposition, inversion) enable efficient computation of forward passes and gradients.
- **Eigendecomposition and SVD** reveal structure in data and are crucial for PCA, matrix factorisation, and understanding dynamics.
- **Norms and distances** provide metrics for measuring similarity, regularisation, and convergence in optimisation.
- **Linear systems** underpin many machine learning algorithms and their solvability determines model identifiability.

Problems

Easy

Problem 2.44 (Matrix Multiplication)**.** Given $\boldsymbol{A} = \begin{bmatrix} 2 & 1 \\ 0 & 3 \end{bmatrix}$ and $\boldsymbol{B} = \begin{bmatrix} 1 & 2 \\ 3 & 1 \end{bmatrix}$, compute $\boldsymbol{AB}$.

Hint: Remember that $(\boldsymbol{AB})_{ij} = \sum_k a_{ik} b_{kj}$.

Problem 2.45 (Vector Norms). Calculate the L1, L2, and L_∞ norms of the vector $\boldsymbol{v} = [3, -4, 0]$.
Hint: L1 norm is sum of absolute values, L2 norm is Euclidean length, L_∞ is maximum absolute value.

Problem 2.46 (Linear Independence). Determine whether the vectors $\boldsymbol{v}_1 = [1, 0, 1]$, $\boldsymbol{v}_2 = [0, 1, 0]$, and $\boldsymbol{v}_3 = [1, 1, 1]$ are linearly independent.
Hint: Check if $c_1\boldsymbol{v}_1 + c_2\boldsymbol{v}_2 + c_3\boldsymbol{v}_3 = \boldsymbol{0}$ has only the trivial solution $c_1 = c_2 = c_3 = 0$.

Problem 2.47 (Matrix Transpose Properties). Prove that $(\boldsymbol{AB})^\top = \boldsymbol{B}^\top\boldsymbol{A}^\top$ for any compatible matrices $\boldsymbol{A}$ and $\boldsymbol{B}$.
Hint: Use the definition of transpose and matrix multiplication element-wise.

Medium

Problem 2.48 (Eigenvalues and Eigenvectors). Find the eigenvalues and eigenvectors of the matrix $\boldsymbol{A} = \begin{bmatrix} 4 & 2 \\ 1 & 3 \end{bmatrix}$.
Hint: Solve $\det(\boldsymbol{A} - \lambda\boldsymbol{I}) = 0$ for eigenvalues, then find eigenvectors by solving $(\boldsymbol{A} - \lambda\boldsymbol{I})\boldsymbol{v} = \boldsymbol{0}$.

Problem 2.49 (SVD Application). Explain how Singular Value Decomposition (SVD) can be used for dimensionality reduction. Describe the relationship between SVD and Principal Component Analysis (PCA).
Hint: Consider which singular values and vectors to keep, and how this relates to variance in the data.

Hard

Problem 2.50 (Matrix Decomposition for Neural Networks). Show how the weight matrix in a neural network layer can be decomposed using SVD to reduce the number of parameters. Analyse the computational and memory trade-offs.
Hint: Consider low-rank approximation $\boldsymbol{W} \approx \boldsymbol{U}_k\boldsymbol{\Sigma}_k\boldsymbol{V}_k^\top$ where $k < \min(m, n)$.

Chapter 3

Probability and Information Theory

This chapter introduces fundamental concepts from probability theory and information theory that are essential for understanding machine learning and deep learning. Topics include probability distributions, conditional probability, expectation, variance, entropy, and mutual information.

Learning Objectives

After studying this chapter, you will be able to:

- **Understand probability foundations**: Grasp the intuitive meaning of probability distributions, both discrete and continuous, and how they model uncertainty in real-world scenarios.
- **Apply conditional probability**: Use Bayes' theorem to update beliefs with new evidence and understand its central role in machine learning algorithms.
- **Calculate statistical measures**: Compute expectation, variance, and covariance to characterize the behavior of random variables and their relationships.
- **Work with common distributions**: Recognize when to use Bernoulli, Gaussian, and other probability distributions in machine learning contexts.

- **Quantify information content**: Use entropy, cross-entropy, and KL divergence to measure uncertainty and information in data and models.

- **Apply information theory**: Connect information-theoretic concepts to loss functions, model selection, and representation learning in deep neural networks.

3.1 Probability Distributions 💫

3.1.1 Intuition: What is Probability?

Imagine you’re playing a game of dice. Before rolling, you know that each face (1 through 6) has an equal chance of appearing. This ”chance” is what we call **probability** - a number between 0 and 1 that quantifies how likely an event is to occur. In machine learning, we face uncertainty everywhere:

- **Data uncertainty**: Will the next customer click on an ad?

- **Model uncertainty**: How confident is our neural network in its prediction?

- **Parameter uncertainty**: What’s the best value for our model’s weights?

Probability distributions are mathematical tools that help us model and work with this uncertainty systematically.

3.1.2 Visualizing Probability

Consider a simple example: predicting whether it will rain tomorrow. We might say there’s a 30% chance of rain. This means:

- If we could repeat tomorrow 100 times, rain would occur about 30 times

- The probability of rain is 0.3

- The probability of no rain is 0.7

Probability theory provides a mathematical framework for quantifying uncertainty. In deep learning, we use probability distributions to model uncertainty in data, model parameters, and predictions.

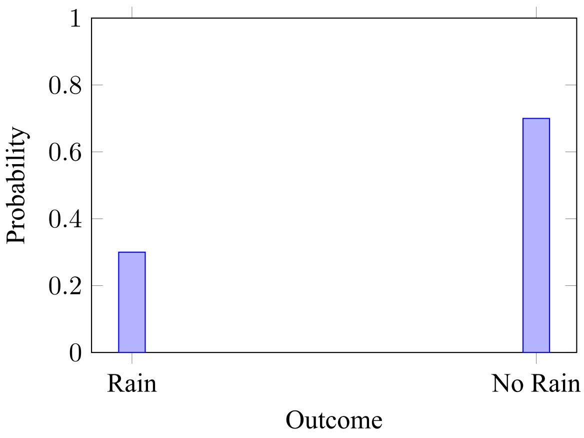

Figure 3.1: Probability distribution for rain prediction

3.1.3 Discrete Probability Distributions

Intuition: Counting Outcomes

Think of discrete probability as counting specific outcomes. For example:

- **Coin flip**: Heads (1) or Tails (0) - only 2 possible outcomes
- **Dice roll**: 1, 2, 3, 4, 5, or 6 - exactly 6 possible outcomes
- **Email classification**: Spam (1) or Not Spam (0) - binary outcome

A discrete random variable X takes values from a countable set. The **probability mass function** (PMF) $P(X = x)$ assigns probabilities to each possible value:

$$P(X = x) \geq 0 \quad \text{for all } x \tag{3.1}$$

$$\sum_x P(X = x) = 1 \tag{3.2}$$

Example: Fair Coin

For a fair coin, we have:

$$P(X = 0) = 0.5 \quad \text{(Tails)} \tag{3.3}$$

$$P(X = 1) = 0.5 \quad \text{(Heads)} \tag{3.4}$$

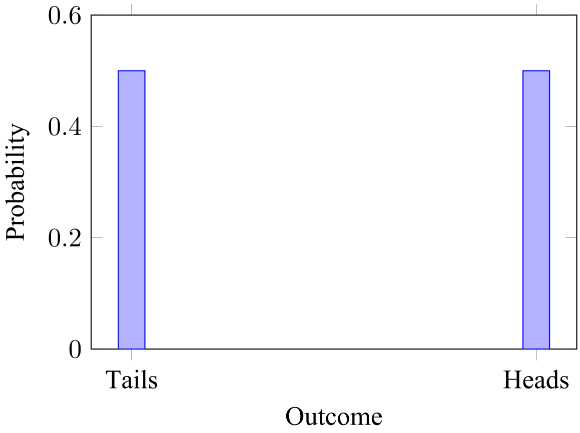

Figure 3.2: Probability mass function for a fair coin

3.1.4 Continuous Probability Distributions

Intuition: Measuring Instead of Counting

Unlike discrete variables, continuous variables can take any value within a range. Think of:

- **Height of people**: Can be 170 cm, 170.5 cm, 170.52 cm, etc.
- **Temperature**: Can be 22.3°C, 22.34°C, 22.341°C, etc.
- **Neural network weights**: Can be 0.1234, 0.12345, 0.123456, etc.

Since there are infinitely many possible values, we can't assign probabilities to individual points. Instead, we use **density** - how "concentrated" the probability is in different regions.
A continuous random variable can take any value in a continuous range. We describe it using a **probability density function** (PDF) $p(x)$:

$$p(x) \geq 0 \quad \text{for all } x \tag{3.5}$$

$$\int_{-\infty}^{\infty} p(x)\, dx = 1 \tag{3.6}$$

The probability that X falls in an interval $[a, b]$ is:

$$P(a \leq X \leq b) = \int_a^b p(x)\, dx \tag{3.7}$$

Example: Normal Distribution

The most common continuous distribution is the **normal (Gaussian) distribution**, which looks like a bell curve:

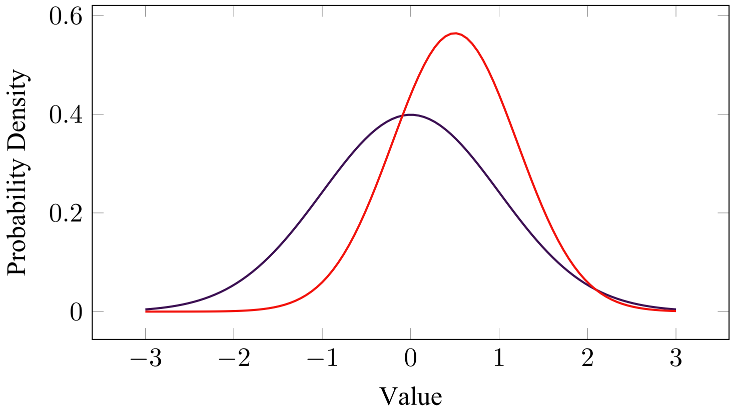

Figure 3.3: Two normal distributions with different means and standard deviations

The area under the curve between any two points gives the probability of the variable falling in that range.

3.1.5 Joint and Marginal Distributions

Intuition: Multiple Variables Together

In real-world scenarios, we often deal with multiple variables simultaneously:

- **Weather prediction**: Temperature AND humidity
- **Image classification**: Pixel values at different positions
- **Stock prices**: Multiple stocks in a portfolio

The **joint distribution** tells us about the probability of combinations of values, while **marginal distributions** tell us about individual variables when we ignore the others.

Example: Weather Data

Consider a simple weather dataset with two variables:

- X: Temperature (Hot/Cold)
- Y: Humidity (High/Low)

	$Y = \text{High}$	$Y = \text{Low}$	**Marginal**
$X = \text{Hot}$	0.3	0.2	**0.5**
$X = \text{Cold}$	0.1	0.4	**0.5**
Marginal	**0.4**	**0.6**	**1.0**

Table 3.1: Joint probability table for weather data

For multiple random variables X and Y, the **joint distribution** $P(X, Y)$ describes their combined behavior. The **marginal distribution** is obtained by summing (or integrating) over the other variable:

$$P(X = x) = \sum_y P(X = x, Y = y) \tag{3.8}$$

For continuous variables:

$$p(x) = \int p(x, y)\, dy \tag{3.9}$$

From our weather example:

- $P(X = \text{Hot}) = 0.3 + 0.2 = 0.5$ (marginal probability of hot weather)
- $P(Y = \text{High}) = 0.3 + 0.1 = 0.4$ (marginal probability of high humidity)

3.2 Conditional Probability and Bayes' Rule 💫

3.2.1 Intuition: Updating Beliefs with New Information

Imagine you're a doctor trying to diagnose a patient. Initially, you might think there's a 5% chance the patient has a rare disease. But then the patient tells you they have a specific symptom that's present in 80% of people with that disease. How should you update your belief?

This is exactly what **conditional probability** helps us do - it tells us how to update our beliefs when we get new information.

3.2.2 Conditional Probability

The **conditional probability** of X given Y is:

$$P(X|Y) = \frac{P(X,Y)}{P(Y)} \tag{3.10}$$

This quantifies how the probability of X changes when we know the value of Y.

Example: Medical Diagnosis

Let's make this concrete with our medical example:

- D: Patient has the disease (1 = yes, 0 = no)
- S: Patient has the symptom (1 = yes, 0 = no)

From medical records, we know:

- $P(D = 1) = 0.05$ (5% of population has the disease)
- $P(S = 1|D = 1) = 0.8$ (80% of diseased patients have the symptom)
- $P(S = 1|D = 0) = 0.1$ (10% of healthy patients have the symptom)

If a patient has the symptom, what's the probability they have the disease?
Using Bayes' theorem (which we'll derive next):

$$\begin{aligned}
P(D = 1|S = 1) &= \frac{P(S = 1|D = 1)P(D = 1)}{P(S = 1)} && (3.11)\\
&= \frac{0.8 \times 0.05}{0.8 \times 0.05 + 0.1 \times 0.95} && (3.12)\\
&= \frac{0.04}{0.04 + 0.095} && (3.13)\\
&= \frac{0.04}{0.135} \approx 0.296 && (3.14)
\end{aligned}$$

So even with the symptom, there's only about a 30% chance the patient has the disease!

3.2.3 Independence

Intuition: When Variables Don't Affect Each Other

Two events are **independent** if knowing one doesn't change our belief about the other. For example:

- **Independent**: Rolling two dice - the result of the first die doesn't affect the second
- **Dependent**: Weather and clothing choice - knowing it's raining affects the probability you'll wear a raincoat

Two random variables X and Y are **independent** if:

$$P(X, Y) = P(X)P(Y) \tag{3.15}$$

Equivalently, $P(X|Y) = P(X)$ and $P(Y|X) = P(Y)$.

Example: Independent vs Dependent Variables

Consider two scenarios:
Scenario 1 (Independent): Flipping two coins

- $P(\text{First coin = Heads}) = 0.5$
- $P(\text{Second coin = Heads}) = 0.5$
- $P(\text{Both Heads}) = 0.5 \times 0.5 = 0.25$ ✓

Scenario 2 (Dependent): Drawing cards without replacement

- $P(\text{First card = Ace}) = 4/52 = 1/13$
- $P(\text{Second card = Ace}) = 3/51$ (if first was Ace) or $4/51$ (if first wasn't Ace)
- The probability of the second card depends on what the first card was

3.2.4 Bayes' Theorem

Intuition: The Most Important Formula in Machine Learning

Bayes' theorem is like a "belief update machine." It tells us how to revise our initial beliefs (prior) when we observe new evidence, to get our updated beliefs (posterior). **Bayes' theorem** is fundamental to probabilistic inference:

$$P(X|Y) = \frac{P(Y|X)P(X)}{P(Y)} \tag{3.16}$$

Understanding Each Component

In machine learning terminology:

- $P(X)$ is the **prior** probability - what we believed before seeing the data
- $P(Y|X)$ is the **likelihood** - how likely the data is given our hypothesis
- $P(X|Y)$ is the **posterior** probability - what we believe after seeing the data
- $P(Y)$ is the **evidence** or marginal likelihood - the probability of observing the data

Visualizing Bayes' Theorem

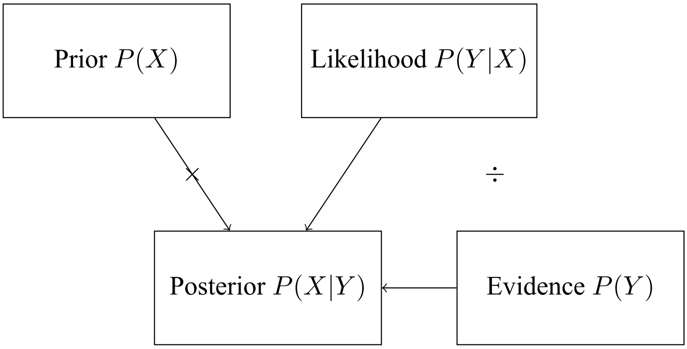

Figure 3.4: Bayes' theorem as a belief update process

The formula can be read as: "Posterior = (Likelihood × Prior) ÷ Evidence"

3.2.5 Application to Machine Learning

Bayes' theorem forms the basis of:

- Bayesian inference
- Naive Bayes classifiers
- Maximum a posteriori (MAP) estimation
- Bayesian neural networks

Given data $\mathcal{D}$ and model parameters θ:

$$P(\theta|\mathcal{D}) = \frac{P(\mathcal{D}|\theta)P(\theta)}{P(\mathcal{D})} \tag{3.17}$$

3.3 Expectation, Variance, and Covariance 💫

3.3.1 Intuition: Characterizing Random Variables

When we have a random variable, we often want to summarize its behavior with a few key numbers:

- **Expected value (mean)**: The ”center” or ”typical” value
- **Variance**: How much the values spread out from the center
- **Covariance**: How two variables move together

Think of it like describing a person:

- **Mean height**: The average height of people in a group
- **Variance in height**: How much heights vary (tall vs short people)
- **Covariance of height and weight**: Do taller people tend to weigh more?

3.3.2 Expectation

The **expected value** or **mean** of a function $f(x)$ with respect to distribution $P(x)$ is: For discrete variables:

$$\mathbb{E}_{x\sim P}[f(x)] = \sum_x P(x)f(x) \tag{3.18}$$

For continuous variables:

$$\mathbb{E}_{x\sim p}[f(x)] = \int p(x)f(x)\,dx \tag{3.19}$$

Example: Expected Value of Dice

For a fair six-sided die:

$$\mathbb{E}[X] = \sum_{x=1}^{6} x \cdot P(X = x) \tag{3.20}$$

$$= 1 \cdot \frac{1}{6} + 2 \cdot \frac{1}{6} + \cdots + 6 \cdot \frac{1}{6} \tag{3.21}$$

$$= \frac{1+2+3+4+5+6}{6} = \frac{21}{6} = 3.5 \tag{3.22}$$

The expected value is 3.5, even though we can never actually roll 3.5!

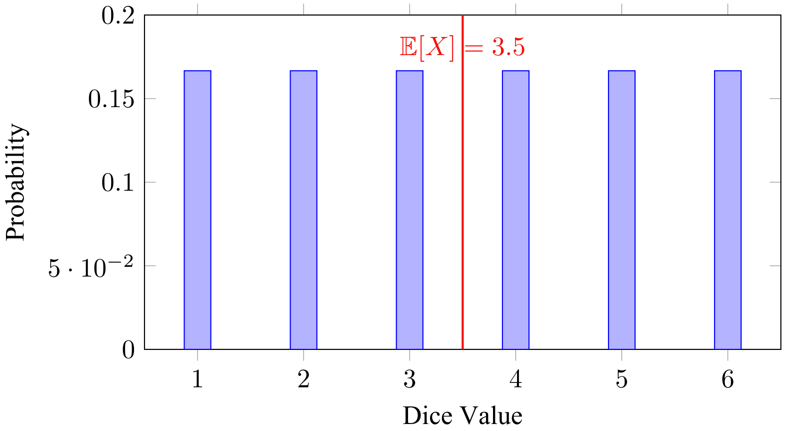

Figure 3.5: Probability distribution of a fair die with expected value marked

3.3.3 Variance

Intuition: Measuring Spread

Variance tells us how ”spread out” the values are around the mean. Think of two dart players:

- **Low variance**: All darts cluster tightly around the bullseye
- **High variance**: Darts are scattered all over the board

The **variance** measures the spread of a distribution:

$$\text{Var}(X) = \mathbb{E}[(X - \mathbb{E}[X])^2] = \mathbb{E}[X^2] - (\mathbb{E}[X])^2 \tag{3.23}$$

The **standard deviation** is $\sigma = \sqrt{\mathrm{Var}(X)}$.

Example: Variance of Dice

For our fair die:

$$\mathrm{Var}(X) = \mathbb{E}[X^2] - (\mathbb{E}[X])^2 \tag{3.24}$$

$$= \left(\frac{1^2 + 2^2 + \cdots + 6^2}{6}\right) - (3.5)^2 \tag{3.25}$$

$$= \frac{91}{6} - 12.25 = 15.17 - 12.25 = 2.92 \tag{3.26}$$

So $\sigma = \sqrt{2.92} \approx 1.71$.

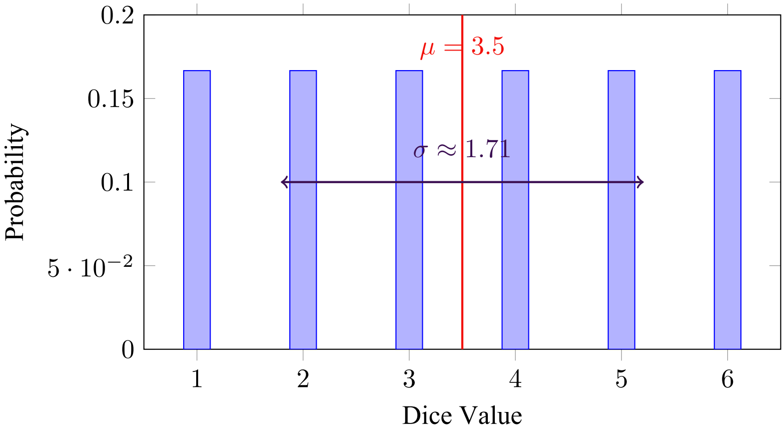

Figure 3.6: Probability distribution showing mean and standard deviation

3.3.4 Covariance

Intuition: How Variables Move Together

Covariance tells us whether two variables tend to move in the same direction or opposite directions:

- **Positive covariance**: When one goes up, the other tends to go up too
- **Negative covariance**: When one goes up, the other tends to go down
- **Zero covariance**: No clear relationship

Examples:

- **Height and weight**: Positive covariance (taller people tend to weigh more)
- **Price and demand**: Negative covariance (higher prices usually mean lower demand)
- **Height and IQ**: Near zero covariance (no clear relationship)

The **covariance** measures how two variables vary together:

$$\text{Cov}(X, Y) = \mathbb{E}[(X - \mathbb{E}[X])(Y - \mathbb{E}[Y])] \tag{3.27}$$

Positive covariance indicates that X and Y tend to increase together, while negative covariance indicates they tend to vary in opposite directions.

Example: Height and Weight

Consider a small dataset of people:

Height (cm)	Weight (kg)
152	54
165	64
178	73
191	82

The means are $\mu_X = 171.5$ and $\mu_Y = 68.25$. The covariance is:

$$\text{Cov}(X, Y) = \frac{1}{4}\sum_{i=1}^{4}(x_i - 171.5)(y_i - 68.25) \tag{3.28}$$

$$= \frac{1}{4}[(-19.5)(-14.25) + (-6.5)(-4.25) + (6.5)(4.75) + (19.5)(13.75)] \tag{3.29}$$

$$= \frac{1}{4}[277.875 + 27.625 + 30.875 + 268.125] = \frac{604.5}{4} = 151.125 \tag{3.30}$$

Positive covariance confirms that taller people tend to weigh more!

3.3.5 Correlation

The **correlation coefficient** normalizes covariance:

$$\rho(X, Y) = \frac{\text{Cov}(X, Y)}{\sqrt{\text{Var}(X)\text{Var}(Y)}} \tag{3.31}$$

Properties:

- $-1 \leq \rho \leq 1$
- $|\rho| = 1$ indicates perfect linear relationship
- $\rho = 0$ indicates no linear relationship (but variables may still be dependent)

3.4 Common Probability Distributions 💫

3.4.1 Bernoulli Distribution

Models a binary random variable (0 or 1):

$$P(X = 1) = \phi, \quad P(X = 0) = 1 - \phi \tag{3.32}$$

Used for binary classification problems.

3.4.2 Categorical Distribution

Generalizes Bernoulli to k discrete outcomes. If X can take values $\{1, 2, \ldots, k\}$:

$$P(X = i) = p_i \quad \text{where} \quad \sum_{i=1}^{k} p_i = 1 \tag{3.33}$$

3.4.3 Gaussian (Normal) Distribution

The most important continuous distribution in deep learning:

$$\mathcal{N}(x; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x - \mu)^2}{2\sigma^2}\right) \tag{3.34}$$

Properties:

- Mean: μ
- Variance: σ^2
- Central limit theorem: sums of independent variables approach Gaussian

The multivariate Gaussian with mean vector $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$ is:

$$\mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{\sqrt{(2\pi)^n |\boldsymbol{\Sigma}|}} \exp\left(-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\boldsymbol{x} - \boldsymbol{\mu})\right) \tag{3.35}$$

3.4.4 Exponential Distribution

Models the time between events in a Poisson process:

$$p(x; \lambda) = \lambda e^{-\lambda x} \quad \text{for } x \geq 0 \tag{3.36}$$

3.4.5 Laplace Distribution

Heavy-tailed alternative to Gaussian:

$$\text{Laplace}(x; \mu, b) = \frac{1}{2b} \exp\left(-\frac{|x - \mu|}{b}\right) \tag{3.37}$$

Used in robust statistics and L1 regularization.

3.4.6 Dirac Delta and Mixture Distributions

The **Dirac delta** $\delta(x)$ concentrates all probability at a single point:

$$p(x) = \delta(x - \mu) \tag{3.38}$$

Mixture distributions combine multiple distributions:

$$p(x) = \sum_{i=1}^{k} \alpha_i p_i(x), \quad \sum_{i=1}^{k} \alpha_i = 1 \tag{3.39}$$

Example: Gaussian Mixture Model (GMM).

3.5 Information Theory Basics 💫

Information theory provides tools for quantifying information and uncertainty, which are crucial for understanding learning and compression.

3.5.1 Self-Information

The **self-information** or **surprisal** of an event x is:

$$I(x) = -\log P(x) \tag{3.40}$$

Rare events have high information content, while certain events have zero information.

3.5.2 Entropy

The **Shannon entropy** measures the expected information in a distribution:

$$H(X) = \mathbb{E}_{x\sim P}[I(x)] = -\sum_x P(x)\log P(x) \tag{3.41}$$

For continuous distributions, we use **differential entropy**:

$$H(X) = -\int p(x)\log p(x)\,dx \tag{3.42}$$

Entropy is maximized when all outcomes are equally likely.

3.5.3 Cross-Entropy

The **cross-entropy** between distributions P and Q is:

$$H(P, Q) = -\mathbb{E}_{x\sim P}[\log Q(x)] = -\sum_x P(x)\log Q(x) \tag{3.43}$$

In deep learning, cross-entropy is commonly used as a loss function for classification.

3.5.4 Kullback-Leibler Divergence

The **KL divergence** measures how one distribution differs from another:

$$D_{KL}(P\|Q) = \mathbb{E}_{x\sim P}\left[\log\frac{P(x)}{Q(x)}\right] = \sum_x P(x)\log\frac{P(x)}{Q(x)} \tag{3.44}$$

Properties:

- $D_{KL}(P\|Q) \geq 0$ with equality if and only if $P = Q$
- Not symmetric: $D_{KL}(P\|Q) \neq D_{KL}(Q\|P)$
- Related to cross-entropy: $D_{KL}(P\|Q) = H(P, Q) - H(P)$

3.5.5 Mutual Information

The **mutual information** between X and Y quantifies how much knowing one reduces uncertainty about the other:

$$I(X;Y) = D_{KL}(P(X,Y) \| P(X)P(Y)) \tag{3.45}$$

Equivalently:

$$I(X;Y) = H(X) - H(X|Y) = H(Y) - H(Y|X) \tag{3.46}$$

Mutual information is symmetric and measures the dependence between variables.

3.5.6 Applications in Deep Learning

Information theory concepts are used in:

- Loss functions (cross-entropy loss)
- Model selection (AIC, BIC use information-theoretic principles)
- Variational inference (minimizing KL divergence)
- Information bottleneck theory
- Mutual information maximization in self-supervised learning

3.6 Problems 💫

This section provides exercises to reinforce your understanding of probability and information theory concepts. Problems are categorized by difficulty level.

3.6.1 Easy Problems (6 problems)

Problem 3.1 (Coin Flipping)**.** A fair coin is flipped 3 times. What is the probability of getting exactly 2 heads?

Problem 3.2 (Dice Probability)**.** A fair six-sided die is rolled. What is the probability of getting an even number?

Problem 3.3 (Basic Conditional Probability). In a class of 30 students, 18 are girls and 12 are boys. If a student is selected at random, what is the probability that they are a girl?

Problem 3.4 (Independent Events). Two fair coins are flipped. What is the probability that both show heads?

Problem 3.5 (Complement Rule). The probability that it will rain tomorrow is 0.3. What is the probability that it will not rain tomorrow?

Problem 3.6 (Basic Expectation). A fair die is rolled. What is the expected value of the outcome?

3.6.2 Medium Problems (5 problems)

Problem 3.7 (Bayes' Theorem Application). A medical test for a disease has a 95% accuracy rate (sensitivity) and a 90% specificity rate. If 2% of the population has the disease, what is the probability that a person who tests positive actually has the disease?

Problem 3.8 (Joint Probability). In a survey of 100 people, 60 like pizza, 40 like burgers, and 20 like both. If a person is selected at random, what is the probability that they like pizza or burgers (or both)?

Problem 3.9 (Variance Calculation). A random variable X has the following probability distribution:

x	$P(X = x)$
1	0.2
2	0.3
3	0.3
4	0.2

Calculate the variance of X.

Problem 3.10 (Normal Distribution). A random variable X follows a normal distribution with mean $\mu = 50$ and standard deviation $\sigma = 10$. What is the probability that X is between 40 and 60?

Problem 3.11 (Entropy Calculation). A random variable X can take values $\{1, 2, 3, 4\}$ with probabilities $\{0.1, 0.4, 0.3, 0.2\}$ respectively. Calculate the entropy $H(X)$.

3.6.3 Hard Problems (5 problems)

Problem 3.12 (Bayesian Inference)**.** You have two coins: one fair and one biased (with probability 0.8 of heads). You randomly select one coin and flip it 3 times, getting heads each time. What is the probability that you selected the biased coin?

Problem 3.13 (Covariance and Correlation)**.** Two random variables X and Y have the following joint probability distribution:

	$Y = 1$	$Y = 2$
$X = 1$	0.2	0.1
$X = 2$	0.3	0.4

Calculate the covariance and correlation coefficient between X and Y.

Problem 3.14 (KL Divergence)**.** Calculate the Kullback-Leibler divergence $D_{KL}(P\|Q)$ where:

$$P(x) = \begin{cases} 0.5 & \text{if } x = 0 \\ 0.5 & \text{if } x = 1 \end{cases} \tag{3.47}$$

$$Q(x) = \begin{cases} 0.3 & \text{if } x = 0 \\ 0.7 & \text{if } x = 1 \end{cases} \tag{3.48}$$

Problem 3.15 (Mutual Information)**.** Two random variables X and Y have joint distribution:

	$Y = 0$	$Y = 1$
$X = 0$	0.4	0.1
$X = 1$	0.2	0.3

Calculate the mutual information $I(X; Y)$.

Problem 3.16 (Maximum Likelihood and MAP)**.** Given observations $\{x_1, x_2, \ldots, x_n\}$ from a normal distribution $\mathcal{N}(\mu, \sigma^2)$, derive the maximum likelihood estimate of μ. Then, assuming a normal prior $\mathcal{N}(\mu_0, \sigma_0^2)$ for μ, derive the MAP estimate.

Hints

Easy Problems Hints

1. **Coin Flipping**: Use the binomial distribution or enumerate all possible outcomes.

2. **Dice Probability**: Count favorable outcomes (2, 4, 6) over total outcomes.

3. **Basic Conditional Probability**: This is just a ratio of favorable cases to total cases.

4. **Independent Events**: Multiply the individual probabilities.

5. **Complement Rule**: Use $P(\text{not A}) = 1 - P(A)$.

6. **Basic Expectation**: Use the definition $\mathbb{E}[X] = \sum x \cdot P(X = x)$.

Medium Problems Hints

1. **Bayes' Theorem**: Use the formula $P(\text{Disease}|\text{Positive}) = \frac{P(\text{Positive}|\text{Disease})P(\text{Disease})}{P(\text{Positive})}$.

2. **Joint Probability**: Use the inclusion-exclusion principle: $P(A \cup B) = P(A) + P(B) - P(A \cap B)$.

3. **Variance Calculation**: First find $\mathbb{E}[X]$ and $\mathbb{E}[X^2]$, then use $\text{Var}(X) = \mathbb{E}[X^2] - (\mathbb{E}[X])^2$.

4. **Normal Distribution**: Use the 68-95-99.7 rule or standardize and use normal tables.

5. **Entropy Calculation**: Use $H(X) = -\sum_i p_i \log p_i$.

Hard Problems Hints

1. **Bayesian Inference**: Use Bayes' theorem with the prior probability of selecting each coin.

2. **Covariance and Correlation**: First find marginal distributions, then use $\text{Cov}(X, Y) = \mathbb{E}[XY] - \mathbb{E}[X]\mathbb{E}[Y]$.

3. **KL Divergence**: Use $D_{KL}(P\|Q) = \sum_x P(x) \log \frac{P(x)}{Q(x)}$.

4. **Mutual Information**: Use $I(X;Y) = H(X) + H(Y) - H(X,Y)$ or $I(X;Y) = \sum_{x,y} P(x,y) \log \frac{P(x,y)}{P(x)P(y)}$.

5. **Maximum Likelihood and MAP**: Take derivatives of the log-likelihood and log-posterior respectively.

Key Takeaways

Key Takeaways 3

- **Probability distributions** model uncertainty and enable principled reasoning under incomplete information.
- **Bayes' theorem** provides a framework for updating beliefs with evidence, central to many machine learning algorithms.
- **Expectation and variance** characterise random variables and guide choices of loss functions and model architectures.
- **Common distributions** (Bernoulli, Gaussian, categorical) serve as building blocks for probabilistic models.
- **Information theory** quantifies uncertainty through entropy and divergence, directly connecting to loss functions and regularisation.

Problems

Easy

Problem 3.17 (Bayes' Theorem Application)**.** Given that P(Disease) = 0.01, P(Positive Test | Disease) = 0.95, and P(Positive Test | No Disease) = 0.05, calculate P(Disease | Positive Test).
Hint: Use Bayes' theorem: $P(A|B) = \frac{P(B|A)P(A)}{P(B)}$. Remember to compute P(Positive Test) first.

Problem 3.18 (Expectation and Variance)**.** A discrete random variable X takes values 1, 2, 3, 4 with probabilities 0.1, 0.2, 0.4, 0.3 respectively. Calculate $\mathbb{E}[X]$ and $\text{Var}(X)$.
Hint: $\mathbb{E}[X] = \sum_i x_i P(X = x_i)$ and $\text{Var}(X) = \mathbb{E}[X^2] - (\mathbb{E}[X])^2$.

Problem 3.19 (Entropy Calculation)**.** Calculate the entropy of a fair coin flip and compare it to the entropy of a biased coin with P(Heads) = 0.9.
Hint: Entropy $H(X) = -\sum_i p_i \log_2 p_i$. Higher entropy means more uncertainty.

Problem 3.20 (Independence Test)**.** Given P(A) = 0.3, P(B) = 0.4, and P(A ∩ B) = 0.12, determine if events A and B are independent.
Hint: Events are independent if P(A ∩ B) = P(A)P(B).

Medium

Problem 3.21 (KL Divergence for Model Comparison)**.** Explain why Kullback-Leibler (KL) divergence is not symmetric and discuss its implications when comparing probability distributions in machine learning.
Hint: Consider $D_{KL}(P||Q)$ versus $D_{KL}(Q||P)$ and their behaviour when P or Q is close to zero.

Problem 3.22 (Cross-Entropy Loss)**.** Show that minimising cross-entropy loss is equivalent to maximising the log-likelihood for classification tasks. Derive the relationship mathematically.
Hint: Start with the cross-entropy $H(p, q) = -\sum_i p_i \log q_i$ where p is the true distribution and q is the predicted distribution.

Hard

Problem 3.23 (Information Theory in Neural Networks)**.** Analyse how mutual information between layers in a neural network can be used to understand information flow during training. Discuss the information bottleneck principle.
Hint: Consider $I(X;Y) = H(X) - H(X|Y)$ and how it relates to representation learning.

Chapter 4

Numerical Computation

This chapter covers numerical methods and computational considerations essential for implementing deep learning algorithms. Topics include gradient-based optimization, numerical stability, and conditioning.

Learning Objectives

After studying this chapter, you will be able to:

- **Understand numerical precision issues**: Recognize how finite precision arithmetic affects deep learning computations and implement solutions for overflow, underflow, and numerical instability.

- **Master gradient-based optimization**: Apply gradient descent and understand the role of Jacobian and Hessian matrices in optimization landscapes, including critical points and saddle points.

- **Handle constrained optimization**: Use Lagrange multipliers and KKT conditions to solve optimization problems with constraints, relevant for regularization and fairness in deep learning.

- **Assess numerical stability**: Calculate condition numbers, identify ill-conditioned problems, and implement gradient checking and other numerical verification techniques.

- **Apply practical numerical techniques**: Use log-sum-exp tricks, mixed precision training, and other numerical stability methods in real deep learning implementations.

- **Debug numerical issues**: Recognize common numerical problems in deep learning and apply appropriate solutions for robust model training.

4.1 Overflow and Underflow 💫

4.1.1 Intuition: The Problem with Finite Precision

Imagine you're trying to measure the height of a building using a ruler that only has markings every meter. If the building is 10.7 meters tall, you might record it as 11 meters - you've lost some precision. Now imagine doing millions of calculations with this imprecise ruler, and you can see how small errors can compound into big problems.

Computers face a similar challenge. They can only represent a finite number of digits, so they must round numbers. This **finite precision** is like having a ruler with limited markings - some information is always lost.

In deep learning, this becomes critical because:

- **Neural networks perform millions of calculations** - small errors accumulate

- **Exponential functions are common** - they can produce extremely large or small numbers

- **Gradients can become very small** - they might round to zero, breaking training

Computers represent real numbers with finite precision, typically using floating-point arithmetic. This leads to **rounding errors** that can accumulate and cause problems.

4.1.2 Floating-Point Representation

The IEEE 754 standard defines floating-point numbers. For 32-bit floats:

- Smallest positive number: approximately 10^{-38}

- Largest number: approximately 10^{38}

- Machine epsilon: approximately 10^{-7}

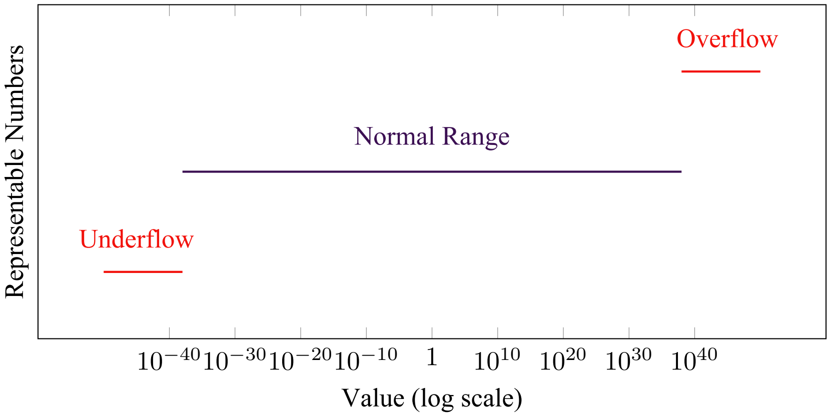

Figure 4.1: Representable range for 32-bit floating-point numbers

4.1.3 Underflow

Intuition: When Numbers Become Too Small

Think of underflow like trying to measure the width of a human hair with a ruler marked in meters. The hair is so thin that your ruler shows 0 meters - you've lost all information about the actual size.

In computers, **underflow** occurs when numbers become so small that they round to zero. This is like your ruler being too coarse to measure tiny objects.

Underflow occurs when numbers near zero are rounded to zero. This can be problematic when we need to compute ratios or logarithms. For example, the softmax function:

$$\text{softmax}(\boldsymbol{x})_i = \frac{\exp(x_i)}{\sum_j \exp(x_j)} \tag{4.1}$$

can underflow if all x_i are very negative.

4.1.4 Overflow

Intuition: When Numbers Become Too Large

Imagine trying to count the number of atoms in the universe using a calculator that can only display 8 digits. When you reach 99,999,999, the next number would be 100,000,000, but your calculator shows ”Error” or resets to 0.

Overflow occurs when large numbers exceed representable values. In the softmax example, overflow can occur if some x_i are very large.

4.1.5 Numerical Stability

To stabilize softmax, we use the identity:

$$\text{softmax}(\boldsymbol{x}) = \text{softmax}(\boldsymbol{x} - c) \tag{4.2}$$

where $c = \max_i x_i$. This prevents both overflow and underflow.
Similarly, when computing $\log(\sum_i \exp(x_i))$, we use the **log-sum-exp** trick:

$$\log\left(\sum_i \exp(x_i)\right) = c + \log\left(\sum_i \exp(x_i - c)\right) \tag{4.3}$$

Example: Softmax Numerical Issues

Consider computing softmax for $\boldsymbol{x} = [1000, 1001, 1002]$:
Naive approach:

$$\exp(1000) \approx \infty \quad \text{(overflow!)} \tag{4.4}$$
$$\exp(1001) \approx \infty \quad \text{(overflow!)} \tag{4.5}$$
$$\exp(1002) \approx \infty \quad \text{(overflow!)} \tag{4.6}$$
$$\text{softmax}(\boldsymbol{x}) = [\text{NaN}, \text{NaN}, \text{NaN}] \tag{4.7}$$

Stable approach:

$$c = \max(1000, 1001, 1002) = 1002 \tag{4.8}$$
$$\exp(1000 - 1002) = \exp(-2) \approx 0.135 \tag{4.9}$$
$$\exp(1001 - 1002) = \exp(-1) \approx 0.368 \tag{4.10}$$
$$\exp(1002 - 1002) = \exp(0) = 1.000 \tag{4.11}$$
$$\text{softmax}(\boldsymbol{x}) = [0.090, 0.245, 0.665] \tag{4.12}$$

4.1.6 Other Numerical Issues

Catastrophic cancellation: Loss of precision when subtracting nearly equal numbers.

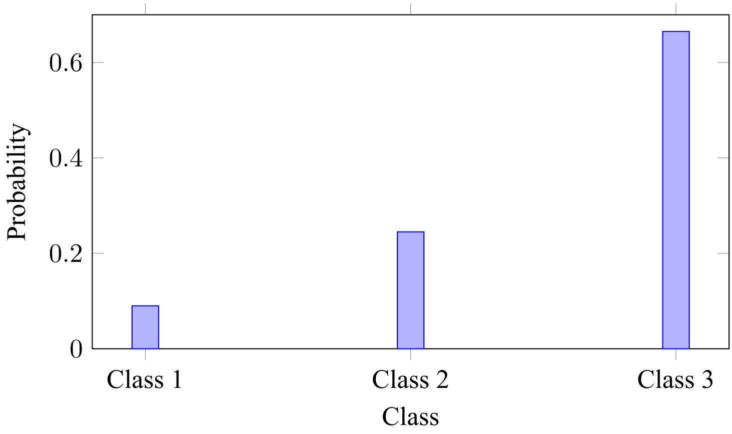

Figure 4.2: Stable softmax computation for large inputs

Accumulated rounding errors: Small errors compound through many operations.
Solutions:

- Use higher precision (64-bit floats)
- Algorithmic modifications (like log-sum-exp)
- Batch normalization
- Gradient clipping

4.2 Gradient-Based Optimization 💫

4.2.1 Intuition: Finding the Bottom of a Hill

Imagine you’re hiking in foggy mountains and need to find the lowest point in a valley. You can’t see far ahead, but you can feel the slope under your feet. The steepest downward direction tells you which way to walk to get lower.
This is exactly what gradient descent does:

- **The mountain** = the loss function we want to minimize
- **Your position** = current parameter values
- **The slope** = gradient (direction of steepest increase)

- **Your steps** = parameter updates

Most deep learning algorithms involve optimization: finding parameters that minimize or maximize an objective function.

4.2.2 Gradient Descent

For a function $f(\boldsymbol{\theta})$, **gradient descent** updates parameters as:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}_t) \tag{4.13}$$

where $\alpha > 0$ is the **learning rate**.

Example: Gradient Descent in 2D

Consider minimizing $f(x, y) = x^2 + 2y^2$ starting from $(3, 3)$:

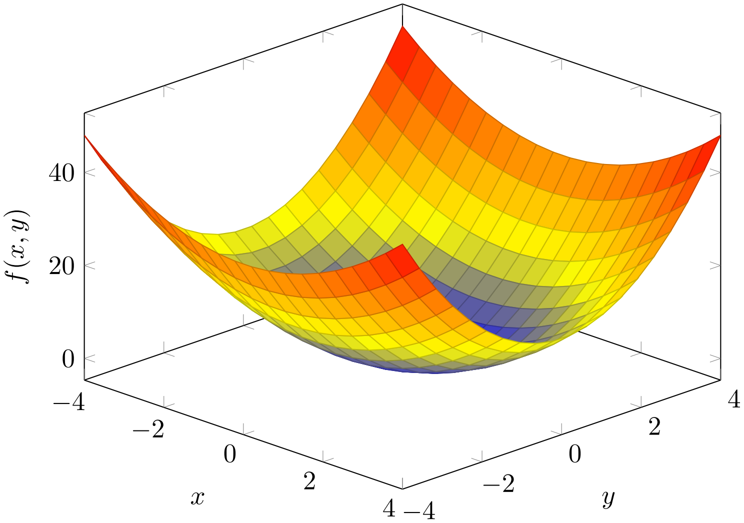

Figure 4.3: Contour plot of $f(x, y) = x^2 + 2y^2$ showing gradient descent path

The gradient is $\nabla f = [2x, 4y]$. Starting from $(3,3)$ with learning rate $\alpha = 0.1$:

$$\nabla f(3,3) = [6,12] \tag{4.14}$$

$$(x_1, y_1) = (3,3) - 0.1[6,12] = (2.4, 1.8) \tag{4.15}$$

$$\nabla f(2.4, 1.8) = [4.8, 7.2] \tag{4.16}$$

$$(x_2, y_2) = (2.4, 1.8) - 0.1[4.8, 7.2] = (1.92, 1.08) \tag{4.17}$$

4.2.3 Jacobian and Hessian Matrices

The **Jacobian matrix** contains all first-order partial derivatives. For $\boldsymbol{f} : \mathbb{R}^n \to \mathbb{R}^m$:

$$\boldsymbol{J}_{ij} = \frac{\partial f_i}{\partial x_j} \tag{4.18}$$

The **Hessian matrix** contains second-order derivatives:

$$\boldsymbol{H}_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j} \tag{4.19}$$

The Hessian characterizes the local curvature of the function.

4.2.4 Taylor Series Approximation

Near point $\boldsymbol{x}_0$, we can approximate $f(\boldsymbol{x})$ using Taylor series:

$$f(\boldsymbol{x}) \approx f(\boldsymbol{x}_0) + (\boldsymbol{x} - \boldsymbol{x}_0)^\top \nabla f(\boldsymbol{x}_0) + \frac{1}{2}(\boldsymbol{x} - \boldsymbol{x}_0)^\top \boldsymbol{H}(\boldsymbol{x}_0)(\boldsymbol{x} - \boldsymbol{x}_0) \tag{4.20}$$

This provides insight into optimization behavior.

4.2.5 Critical Points

Intuition: Different Types of Critical Points

Think of critical points as different types of terrain features:

- **Local minimum** = Bottom of a bowl - you can't go lower in any direction
- **Local maximum** = Top of a hill - you can't go higher in any direction
- **Saddle point** = Mountain pass - you can go down in some directions, up in others

At a **critical point**, $\nabla f(\boldsymbol{x}) = \mathbf{0}$. The Hessian determines the nature:

- **Local minimum:** Hessian is positive definite
- **Local maximum:** Hessian is negative definite
- **Saddle point:** Hessian has both positive and negative eigenvalues

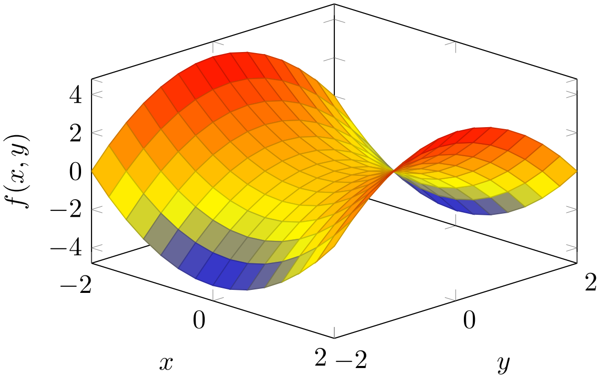

Figure 4.4: Example of a saddle point: $f(x, y) = x^2 - y^2$

Deep learning often encounters saddle points rather than local minima in high dimensions.

4.2.6 Directional Derivatives

The directional derivative in direction $\boldsymbol{u}$ (with $\|\boldsymbol{u}\| = 1$) is:

$$\frac{\partial}{\partial \alpha} f(\boldsymbol{x} + \alpha \boldsymbol{u}) \bigg|_{\alpha=0} = \boldsymbol{u}^\top \nabla f(\boldsymbol{x}) \tag{4.21}$$

To minimize f, we move in the direction $\boldsymbol{u} = -\frac{\nabla f(\boldsymbol{x})}{\|\nabla f(\boldsymbol{x})\|}$.

4.3 Constrained Optimization 💫

4.3.1 Intuition: Optimization with Rules

Imagine you're trying to find the best location for a new store, but you have constraints:

- Must be within 10 km of the city centre
- Must have parking for at least 50 cars
- Budget cannot exceed \$1 million

You can't just pick any location - you must follow these rules while still optimizing your objective (like maximizing customer traffic).
In deep learning, we often have similar constraints:

- **Weight constraints**: Keep weights small to prevent overfitting
- **Probability constraints**: Outputs must sum to 1 (like softmax)
- **Fairness constraints**: Model must treat different groups equally

Many problems require optimizing a function subject to constraints.

4.3.2 Lagrange Multipliers

For equality constraint $g(\boldsymbol{x}) = 0$, the **Lagrangian** is:

$$\mathcal{L}(\boldsymbol{x}, \lambda) = f(\boldsymbol{x}) + \lambda g(\boldsymbol{x}) \tag{4.22}$$

At the optimum, both:

$$\nabla_{\boldsymbol{x}} \mathcal{L} = \mathbf{0} \quad \text{and} \quad \frac{\partial \mathcal{L}}{\partial \lambda} = 0 \tag{4.23}$$

4.3.3 Inequality Constraints

For inequality constraint $g(\boldsymbol{x}) \leq 0$, we use the **Karush-Kuhn-Tucker (KKT)** conditions:

$$\begin{align} \nabla_{\boldsymbol{x}} \mathcal{L} &= \mathbf{0} \tag{4.24} \\ \lambda &\geq 0 \tag{4.25} \\ \lambda g(\boldsymbol{x}) &= 0 \quad \text{(complementary slackness)} \tag{4.26} \\ g(\boldsymbol{x}) &\leq 0 \tag{4.27} \end{align}$$

4.3.4 Projected Gradient Descent

For constraints defining a set $\mathcal{C}$, **projected gradient descent** applies:

$$\boldsymbol{x}_{t+1} = \text{Proj}_{\mathcal{C}}\left(\boldsymbol{x}_t - \alpha \nabla f(\boldsymbol{x}_t)\right) \tag{4.28}$$

where $\text{Proj}_{\mathcal{C}}$ projects onto the feasible set.

4.3.5 Applications in Deep Learning

Constrained optimization appears in:

- Weight constraints (e.g., unit norm constraints)
- Projection to valid probability distributions
- Adversarial training with bounded perturbations
- Fairness constraints

4.4 Numerical Stability and Conditioning 💫

4.4.1 Intuition: The Butterfly Effect in Computation

Imagine a house of cards. A tiny breeze can cause the entire structure to collapse. In numerical computation, we have a similar problem: small errors can grow into large ones.

This is especially problematic in deep learning because:

- **Deep networks** have many layers - errors compound
- **Matrix operations** can amplify small errors
- **Gradient computation** requires precise derivatives

The **condition number** tells us how ”sensitive” a computation is to small changes. A high condition number means small input errors become large output errors.

4.4.2 Condition Number

The **condition number** of matrix $\boldsymbol{A}$ is:

$$\kappa(\boldsymbol{A}) = \|\boldsymbol{A}\|\|\boldsymbol{A}^{-1}\| \tag{4.29}$$

For symmetric matrices with eigenvalues λ_i:

$$\kappa(\boldsymbol{A}) = \frac{\max_i |\lambda_i|}{\min_i |\lambda_i|} \tag{4.30}$$

High condition numbers indicate numerical instability: small changes in input lead to large changes in output.

Example: Well-Conditioned vs Ill-Conditioned Matrices

Consider two matrices:

$$\boldsymbol{A}_1 = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} \quad \text{(well-conditioned)} \tag{4.31}$$

$$\boldsymbol{A}_2 = \begin{bmatrix} 1 & 0.99 \\ 0.99 & 1 \end{bmatrix} \quad \text{(ill-conditioned)} \tag{4.32}$$

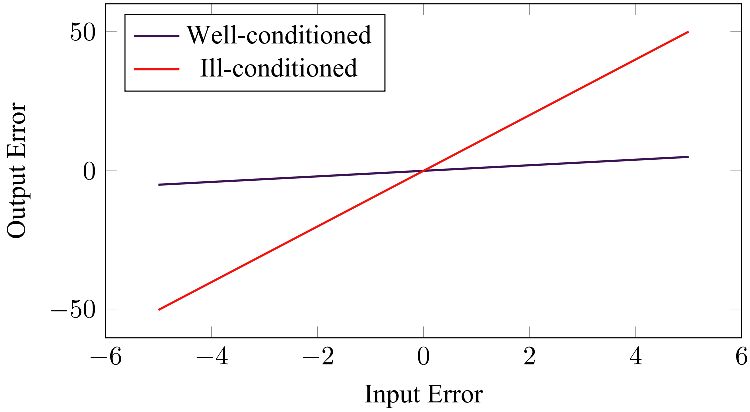

Figure 4.5: Error amplification for well-conditioned vs ill-conditioned matrices

4.4.3 Ill-Conditioned Matrices

In deep learning, ill-conditioned Hessians can make optimization difficult. This motivates techniques like:

- Batch normalization
- Careful weight initialization
- Adaptive learning rate methods
- Preconditioning

4.4.4 Gradient Checking

To verify gradient computations, we use **finite differences**:

$$\frac{\partial f}{\partial \theta_i} \approx \frac{f(\theta_i + \epsilon) - f(\theta_i - \epsilon)}{2\epsilon} \tag{4.33}$$

This is computationally expensive but useful for debugging.

4.4.5 Numerical Precision Trade-offs

Mixed precision training:

- Store weights in FP32
- Compute activations/gradients in FP16
- Use loss scaling to prevent underflow
- 2-3x speedup with minimal accuracy loss

4.4.6 Practical Tips

- Monitor gradient norms during training
- Use gradient clipping for RNNs
- Prefer numerically stable implementations (log-space computations)
- Be aware of precision limits in very deep networks

Key Takeaways

Key Takeaways 4

- **Numerical precision** matters: Finite precision arithmetic can cause overflow, underflow, and instability in deep learning computations.
- **Gradient-based optimisation** relies on Jacobian and Hessian matrices to navigate loss landscapes and find optimal parameters.
- **Constrained optimisation** uses Lagrange multipliers and KKT conditions to solve problems with constraints.
- **Numerical stability** is assessed via condition numbers; ill-conditioned problems require careful handling.
- **Practical techniques** like log-sum-exp tricks and gradient checking ensure robust implementations.

4.5 Problems

This section contains exercises to reinforce your understanding of numerical computation concepts. Problems are categorised by difficulty level.

4.5.1 Easy Problems

Problem 4.1 (Floating-Point Basics)**.** Consider a hypothetical 4-bit floating-point system with 1 sign bit, 2 exponent bits, and 1 mantissa bit. What is the smallest positive number that can be represented? What is the largest number?
Hint: Use the IEEE 754 format: $(-1)^s \times 2^{e-b} \times (1+m)$ where s is the sign bit, e is the exponent, b is the bias, and m is the mantissa.

Problem 4.2 (Softmax Stability)**.** Compute the softmax of $\boldsymbol{x} = [1000, 1001, 1002]$ using both the naive approach and the numerically stable approach. Show your work step by step.
Hint: Use the identity $\text{softmax}(\boldsymbol{x}) = \text{softmax}(\boldsymbol{x} - c)$ where $c = \max_i x_i$.

Problem 4.3 (Gradient Computation)**.** Compute the gradient of $f(x, y) = x^2 + 3xy + y^2$ at the point $(2, 1)$.

Hint: The gradient is $\nabla f = \left[\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right]$.

Problem 4.4 (Condition Number)**.** Calculate the condition number of the matrix $\boldsymbol{A} = \begin{bmatrix} 1 & 0.5 \\ 0.5 & 1 \end{bmatrix}$.

Hint: For a 2×2 matrix, $\kappa(\boldsymbol{A}) = \frac{\lambda_{\text{max}}}{\lambda_{\text{min}}}$ where λ_{max} and λ_{min} are the eigenvalues.

4.5.2 Medium Problems

Problem 4.5 (Log-Sum-Exp Trick)**.** Derive the log-sum-exp trick: $\log\left(\sum_{i=1}^{n} \exp(x_i)\right) = c + \log\left(\sum_{i=1}^{n} \exp(x_i - c)\right)$ where $c = \max_i x_i$.

Hint: Start by factoring out $\exp(c)$ from the sum.

Problem 4.6 (Lagrange Multipliers)**.** Find the maximum value of $f(x, y) = xy$ subject to the constraint $x^2 + y^2 = 1$ using Lagrange multipliers.

Hint: Set up the Lagrangian $\mathcal{L}(x, y, \lambda) = xy + \lambda(1 - x^2 - y^2)$ and solve the system of equations.

4.5.3 Hard Problems

Problem 4.7 (Numerical Stability of Matrix Inversion)**.** Consider the matrix $\boldsymbol{A} = \begin{bmatrix} 1 & 1 \\ 1 & 1 + \epsilon \end{bmatrix}$ where ϵ is small. Show that the condition number grows as $\epsilon \to 0$. Implement a numerical experiment to demonstrate this and show how the error in $\boldsymbol{A}^{-1}$ grows.

Hint: Use the formula $\kappa(\boldsymbol{A}) = \frac{\lambda_{\text{max}}}{\lambda_{\text{min}}}$ and compute the eigenvalues analytically.

Problem 4.8 (KKT Conditions Application)**.** Consider the optimisation problem:

$$\min_{x,y} \quad x^2 + y^2 \tag{4.34}$$

$$\text{subject to} \quad x + y \geq 1 \tag{4.35}$$

$$x \geq 0, y \geq 0 \tag{4.36}$$

Find the optimal solution using the KKT conditions and verify that all conditions are satisfied.

Hint: Set up the Lagrangian with multiple constraints and check the complementary slackness conditions.

Chapter 5

Classical Machine Learning Algorithms

This chapter reviews traditional machine learning methods that provide context and motivation for deep learning approaches. Understanding these classical algorithms helps appreciate the advantages and innovations of deep learning.

Learning Objectives

After studying this chapter, you will be able to:

1. **Understand the mathematical foundations** of classical machine learning algorithms including linear regression, logistic regression, and support vector machines

2. **Compare and contrast** different approaches to classification and regression problems

3. **Implement and optimize** classical algorithms using both closed-form solutions and iterative methods

4. **Apply ensemble methods** like random forests and gradient boosting to improve model performance

5. **Evaluate the trade-offs** between classical methods and deep learning approaches

6. **Choose appropriate algorithms** based on dataset characteristics, computational constraints, and interpretability requirements

7. **Understand the limitations** of classical methods that motivated the development of deep learning

8. **Apply regularization techniques** to prevent overfitting in classical machine learning models

This chapter assumes familiarity with linear algebra, probability theory, and basic optimization concepts from previous chapters.

5.1 Linear Regression ☒

Linear regression is one of the most fundamental and widely-used machine learning algorithms. It models the relationship between input features and a continuous output by finding the best linear function that minimizes prediction errors.

5.1.1 Intuition and Motivation

Imagine you're trying to predict house prices based on features like size, number of bedrooms, and location. Linear regression assumes that the price can be expressed as a weighted sum of these features plus a base price (bias). The algorithm learns the optimal weights that best explain the relationship between features and prices in your training data.

The key insight is that linear relationships are often sufficient for many real-world problems, and they have several advantages:

- **Interpretability:** Each weight tells us how much the output changes when a feature increases by one unit

- **Computational efficiency:** Fast training and prediction

- **Statistical properties:** Well-understood theoretical guarantees

5.1.2 Model Formulation

For input $\boldsymbol{x} \in \mathbb{R}^d$ and output $y \in \mathbb{R}$, linear regression models the relationship as:

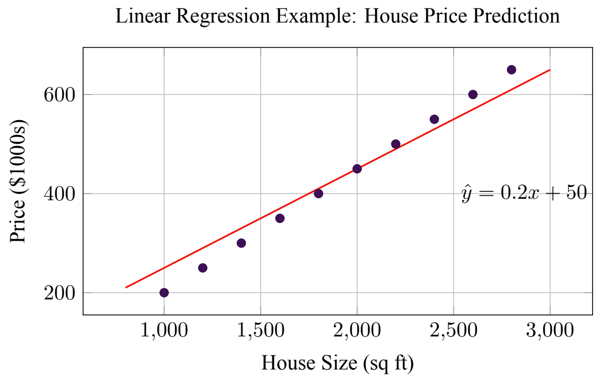

Figure 5.1: Linear regression finds the best line that fits the data points, minimizing the sum of squared errors.

$$\hat{y} = \boldsymbol{w}^\top \boldsymbol{x} + b \tag{5.1}$$

where:

- $\boldsymbol{w} \in \mathbb{R}^d$ are the **weights** (regression coefficients)
- $b \in \mathbb{R}$ is the **bias** (intercept term)
- $\hat{y}$ is the predicted output

5.1.3 Ordinary Least Squares

The goal is to find parameters that minimize the prediction error. We use the **mean squared error** (MSE) as our loss function:

$$L(\boldsymbol{w}, b) = \frac{1}{n}\sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2 = \frac{1}{n}\sum_{i=1}^{n}(y^{(i)} - \boldsymbol{w}^\top \boldsymbol{x}^{(i)} - b)^2 \tag{5.2}$$

Matrix Formulation

For computational efficiency, we can absorb the bias into the weight vector by adding a constant feature of 1 to each input. Let $\boldsymbol{X} \in \mathbb{R}^{n\times(d+1)}$ be the design matrix with an additional column of ones, and $\boldsymbol{w} \in \mathbb{R}^{d+1}$ include the bias term.

The closed-form solution (normal equation) is:

$$\boldsymbol{w}^* = (\boldsymbol{X}^\top \boldsymbol{X})^{-1} \boldsymbol{X}^\top \boldsymbol{y} \tag{5.3}$$

Remark 5.1. The normal equation requires $\boldsymbol{X}^\top \boldsymbol{X}$ to be invertible. This condition is satisfied when the features are linearly independent and we have at least as many training examples as features.

5.1.4 Regularized Regression

When we have many features or when features are correlated, the normal equation can become unstable. Regularization helps by adding a penalty term to prevent overfitting.

Ridge Regression (L2 Regularization)

Ridge regression adds an L2 penalty to the loss function:

$$L(\boldsymbol{w}) = \|\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}\|^2 + \lambda \|\boldsymbol{w}\|^2 \tag{5.4}$$

The solution becomes:

$$\boldsymbol{w}^* = (\boldsymbol{X}^\top \boldsymbol{X} + \lambda \boldsymbol{I})^{-1} \boldsymbol{X}^\top \boldsymbol{y} \tag{5.5}$$

where $\lambda > 0$ is the regularization strength.

Example 5.2. For a simple 2D case with features x_1 and x_2, ridge regression finds:

$$\hat{y} = w_1 x_1 + w_2 x_2 + b$$

The L2 penalty $\lambda(w_1^2 + w_2^2)$ encourages smaller weights, leading to a smoother, more stable solution.

Lasso Regression (L1 Regularization)

Lasso regression uses L1 regularization, which promotes sparsity:

$$L(\boldsymbol{w}) = \|\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}\|^2 + \lambda \|\boldsymbol{w}\|_1 \tag{5.6}$$

Unlike ridge regression, lasso can drive some weights to exactly zero, effectively performing automatic feature selection.

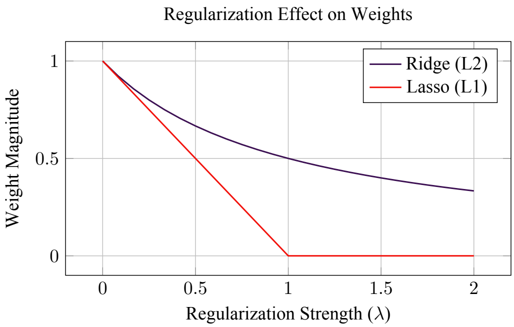

Figure 5.2: Comparison of L1 and L2 regularization effects. L1 can drive weights to zero, while L2 shrinks them smoothly.

5.1.5 Gradient Descent Solution

For large datasets, computing the inverse of $\boldsymbol{X}^\top \boldsymbol{X}$ can be computationally expensive. Gradient descent provides an iterative alternative:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \alpha \nabla_{\boldsymbol{w}} L(\boldsymbol{w}_t) \tag{5.7}$$

where the gradient is:

$$\nabla_{\boldsymbol{w}} L(\boldsymbol{w}) = \frac{2}{n} \boldsymbol{X}^\top (\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}) \tag{5.8}$$

Stochastic Gradient Descent

For very large datasets, we can use stochastic gradient descent (SGD), which updates weights using only a subset of the data at each iteration:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \alpha \nabla_{\boldsymbol{w}} L_i(\boldsymbol{w}_t) \tag{5.9}$$

where L_i is the loss for a single training example or a small batch.

5.1.6 Geometric Interpretation

Linear regression can be understood geometrically as finding the projection of the target vector $\boldsymbol{y}$ onto the column space of the design matrix $\boldsymbol{X}$. The residual vector $\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}^*$ is orthogonal to the column space of $\boldsymbol{X}$.

Theorem 5.3 (Orthogonality Principle)**.** *The optimal solution $\boldsymbol{w}^*$ satisfies:*

$$\boldsymbol{X}^\top(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}^*) = \boldsymbol{0}$$

This means the residual vector is orthogonal to all feature vectors.

5.2 Logistic Regression ☒

Logistic regression is a fundamental classification algorithm that models the probability of class membership using a logistic (sigmoid) function. Despite its name, it's actually a classification method, not a regression method.

5.2.1 Intuition and Motivation

Logistic regression extends linear regression to handle classification problems. Instead of predicting continuous values, it predicts probabilities that an input belongs to a particular class. The key insight is to use a sigmoid function to map linear combinations of features to probabilities between 0 and 1.

Think of logistic regression as answering: ”Given these features, what's the probability that this example belongs to the positive class?” For example, given a patient's symptoms, what's the probability they have a particular disease?

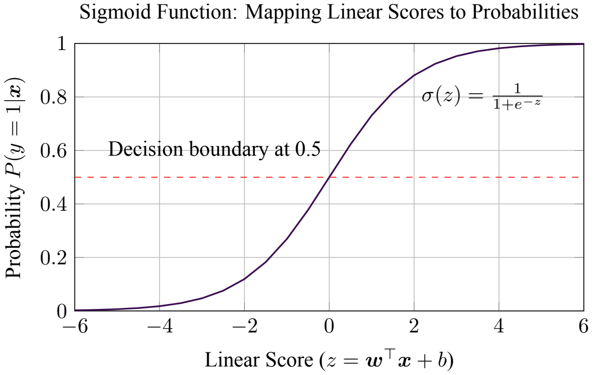

Figure 5.3: The sigmoid function smoothly maps any real number to a probability between 0 and 1. The decision boundary is typically at 0.5.

5.2.2 Binary Classification

For binary classification with classes $\{0, 1\}$, logistic regression models the probability of the positive class using the sigmoid function:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{5.10}$$

The prediction probability is:

$$P(y = 1|\boldsymbol{x}) = \sigma(\boldsymbol{w}^\top \boldsymbol{x} + b) = \frac{1}{1 + e^{-(\boldsymbol{w}^\top \boldsymbol{x} + b)}} \tag{5.11}$$

Properties of the Sigmoid Function

The sigmoid function has several important properties:

- **Range:** $\sigma(z) \in (0, 1)$ for all $z \in \mathbb{R}$
- **Monotonic:** $\sigma'(z) = \sigma(z)(1 - \sigma(z)) > 0$ for all z
- **Symmetric:** $\sigma(-z) = 1 - \sigma(z)$
- **Asymptotic:** $\lim_{z \to \infty} \sigma(z) = 1$ and $\lim_{z \to -\infty} \sigma(z) = 0$

5.2.3 Cross-Entropy Loss

Unlike linear regression, we can't use mean squared error for classification because the sigmoid function is non-linear. Instead, we use the **cross-entropy loss** (negative log-likelihood):

$$L(\boldsymbol{w}, b) = -\frac{1}{n} \sum_{i=1}^{n} \left[y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}) \right] \tag{5.12}$$

where $\hat{y}^{(i)} = P(y = 1|\boldsymbol{x}^{(i)})$.

Derivation of Cross-Entropy Loss

The cross-entropy loss comes from maximum likelihood estimation. For a single example, the likelihood is:

$$L_i = P(y^{(i)}|\boldsymbol{x}^{(i)}) = (\hat{y}^{(i)})^{y^{(i)}} (1 - \hat{y}^{(i)})^{1 - y^{(i)}}$$

Taking the negative log-likelihood:

$$-\log L_i = -y^{(i)} \log \hat{y}^{(i)} - (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})$$

5.2.4 Gradient Descent for Logistic Regression

The gradient of the cross-entropy loss with respect to the weights is:

$$\nabla_{\boldsymbol{w}} L = \frac{1}{n} \sum_{i=1}^{n} (\hat{y}^{(i)} - y^{(i)}) \boldsymbol{x}^{(i)} \tag{5.13}$$

The weight update rule becomes:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \alpha \frac{1}{n} \sum_{i=1}^{n} (\hat{y}^{(i)} - y^{(i)}) \boldsymbol{x}^{(i)} \tag{5.14}$$

Remark 5.4. The gradient has a simple form: it's the average of the prediction errors multiplied by the input features. This makes logistic regression particularly efficient to train.

5.2.5 Multiclass Classification

For K classes, we extend logistic regression to **softmax regression** (also called multinomial logistic regression). Instead of a single sigmoid function, we use the softmax function:

$$P(y = k|\boldsymbol{x}) = \frac{\exp(\boldsymbol{w}_k^\top \boldsymbol{x} + b_k)}{\sum_{j=1}^{K} \exp(\boldsymbol{w}_j^\top \boldsymbol{x} + b_j)} \tag{5.15}$$

Properties of Softmax

The softmax function has several key properties:

- **Probability distribution:** $\sum_{k=1}^{K} P(y = k|\boldsymbol{x}) = 1$
- **Non-negative:** $P(y = k|\boldsymbol{x}) \geq 0$ for all k
- **Monotonic:** Higher scores lead to higher probabilities
- **Scale invariant:** Adding a constant to all scores doesn't change probabilities

5.2.6 Categorical Cross-Entropy Loss

For multiclass classification, we use the categorical cross-entropy loss:

$$L = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K} y_k^{(i)} \log \hat{y}_k^{(i)} \tag{5.16}$$

where $y_k^{(i)}$ is 1 if example i belongs to class k, and 0 otherwise (one-hot encoding).

5.2.7 Decision Boundaries

Logistic regression creates linear decision boundaries. For binary classification, the decision boundary is the hyperplane where $P(y = 1|\boldsymbol{x}) = 0.5$, which occurs when $\boldsymbol{w}^\top \boldsymbol{x} + b = 0$.

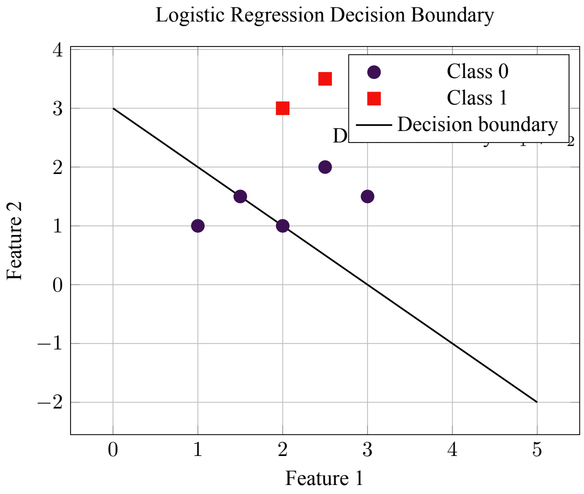

Figure 5.4: Logistic regression finds a linear decision boundary that separates the two classes. Points on one side are classified as class 0, points on the other side as class 1.

5.2.8 Regularization in Logistic Regression

Just like linear regression, logistic regression can benefit from regularization to prevent overfitting:

L2 Regularization (Ridge)

$$L(\boldsymbol{w}) = -\frac{1}{n}\sum_{i=1}^{n}\left[y^{(i)}\log\hat{y}^{(i)} + (1-y^{(i)})\log(1-\hat{y}^{(i)})\right] + \lambda\|\boldsymbol{w}\|^2 \tag{5.17}$$

L1 Regularization (Lasso)

$$L(\boldsymbol{w}) = -\frac{1}{n}\sum_{i=1}^{n}\left[y^{(i)}\log\hat{y}^{(i)} + (1-y^{(i)})\log(1-\hat{y}^{(i)})\right] + \lambda\|\boldsymbol{w}\|_1 \tag{5.18}$$

5.2.9 Advantages and Limitations

Advantages:

- Simple and interpretable
- Fast training and prediction
- Provides probability estimates
- Works well with small datasets
- No assumptions about feature distributions

Limitations:

- Assumes linear relationship between features and log-odds
- Sensitive to outliers
- May not work well with highly correlated features
- Limited to linear decision boundaries

5.3 Support Vector Machines ⊠

Support Vector Machines (SVMs) are powerful classification algorithms that find the optimal hyperplane that maximally separates different classes. The key insight is to maximize the margin between classes, leading to better generalization performance.

5.3.1 Intuition and Motivation

Imagine you have two groups of points on a plane that you want to separate with a line. There are many possible lines that could separate them, but SVM finds the line that maximizes the distance to the nearest points from each class. This ”maximum margin” approach leads to better generalization because the decision boundary is as far as possible from both classes.
The key concepts are:

- **Support vectors:** The training examples closest to the decision boundary
- **Margin:** The distance between the decision boundary and the nearest support vectors
- **Maximum margin principle:** Choose the hyperplane that maximizes this margin

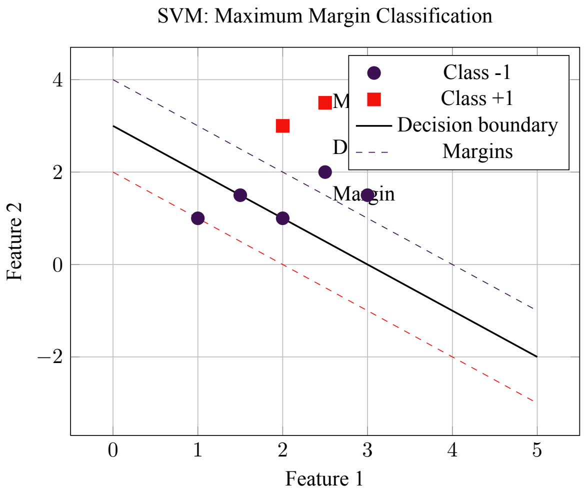

Figure 5.5: SVM finds the hyperplane (line in 2D) that maximizes the margin between classes. The support vectors are the points closest to the decision boundary.

5.3.2 Linear SVM

For binary classification with labels $y \in \{-1, +1\}$, the decision boundary is:

$$\boldsymbol{w}^\top \boldsymbol{x} + b = 0 \tag{5.19}$$

The **margin** is the distance between the decision boundary and the nearest support vectors. For a point $\boldsymbol{x}^{(i)}$, the distance to the hyperplane is:

$$\text{distance} = \frac{|y^{(i)}(\boldsymbol{w}^\top \boldsymbol{x}^{(i)} + b)|}{\|\boldsymbol{w}\|} \tag{5.20}$$

Since we want to maximize the margin, we can set the margin to be $\frac{2}{\|\boldsymbol{w}\|}$ by requiring:

$$y^{(i)}(\boldsymbol{w}^\top \boldsymbol{x}^{(i)} + b) \geq 1 \quad \forall i \tag{5.21}$$

Optimization Problem

Maximizing the margin is equivalent to minimizing $\|\boldsymbol{w}\|^2$ subject to the constraints:

$$\min_{\boldsymbol{w},b} \frac{1}{2}\|\boldsymbol{w}\|^2 \tag{5.22}$$

subject to:

$$y^{(i)}(\boldsymbol{w}^\top \boldsymbol{x}^{(i)} + b) \geq 1 \quad \forall i \tag{5.23}$$

This is a quadratic programming problem that can be solved using Lagrange multipliers.

5.3.3 Soft Margin SVM

In practice, data is rarely linearly separable. The **soft margin SVM** allows some training examples to be misclassified by introducing slack variables ξ_i:

$$\min_{\boldsymbol{w},b,\boldsymbol{\xi}} \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{i=1}^{n} \xi_i \tag{5.24}$$

subject to:

$$y^{(i)}(\boldsymbol{w}^\top \boldsymbol{x}^{(i)} + b) \geq 1 - \xi_i, \quad \xi_i \geq 0 \tag{5.25}$$

The parameter C controls the trade-off between:

- **Large margin:** Small C allows more slack, larger margin
- **Training accuracy:** Large C penalizes misclassifications more heavily

5.3.4 Dual Formulation

The SVM optimization problem can be reformulated in its dual form, which reveals the support vectors and enables the kernel trick:

$$\max_{\boldsymbol{\alpha}} \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y^{(i)} y^{(j)} \boldsymbol{x}^{(i)} \cdot \boldsymbol{x}^{(j)} \tag{5.26}$$

subject to:

$$\sum_{i=1}^{n} \alpha_i y^{(i)} = 0, \quad 0 \le \alpha_i \le C \tag{5.27}$$

The decision function becomes:

$$f(\boldsymbol{x}) = \sum_{i=1}^{n} \alpha_i y^{(i)} \boldsymbol{x}^{(i)} \cdot \boldsymbol{x} + b \tag{5.28}$$

Only examples with $\alpha_i > 0$ are support vectors.

5.3.5 Kernel Trick

For non-linear decision boundaries, we can map inputs to a higher-dimensional space using a **kernel function** $k(\boldsymbol{x}, \boldsymbol{x}')$:

$$f(\boldsymbol{x}) = \sum_{i=1}^{n} \alpha_i y^{(i)} k(\boldsymbol{x}^{(i)}, \boldsymbol{x}) + b \tag{5.29}$$

Common Kernels

Linear kernel:

$$k(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{x}^\top \boldsymbol{x}' \tag{5.30}$$

Polynomial kernel:

$$k(\boldsymbol{x}, \boldsymbol{x}') = (\boldsymbol{x}^\top \boldsymbol{x}' + c)^d \tag{5.31}$$

RBF (Gaussian) kernel:

$$k(\boldsymbol{x}, \boldsymbol{x}') = \exp(-\gamma \|\boldsymbol{x} - \boldsymbol{x}'\|^2) \tag{5.32}$$

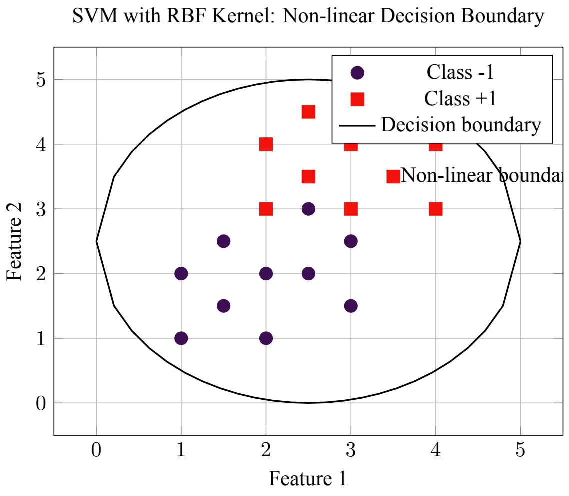

Figure 5.6: SVM with RBF kernel can learn non-linear decision boundaries. The decision boundary here is approximately circular.

5.3.6 Kernel Properties

A function $k(\boldsymbol{x}, \boldsymbol{x}')$ is a valid kernel if and only if it is:

- **Symmetric:** $k(\boldsymbol{x}, \boldsymbol{x}') = k(\boldsymbol{x}', \boldsymbol{x})$
- **Positive semi-definite:** For any set of points $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(n)}\}$, the kernel matrix $K_{ij} = k(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)})$ is positive semi-definite

5.3.7 Advantages and Limitations

Advantages:

- Effective in high-dimensional spaces
- Memory efficient (only stores support vectors)
- Versatile (different kernel functions)
- Strong theoretical foundation
- Works well with small to medium datasets

Limitations:

- Poor performance on large datasets
- Sensitive to feature scaling
- No probabilistic output
- Kernel selection can be tricky
- Computationally expensive for very large datasets

5.3.8 SVM for Regression

SVM can also be extended to regression problems (Support Vector Regression, SVR). Instead of finding a hyperplane that separates classes, SVR finds a hyperplane that fits the data within an ϵ-tube:

$$\min_{\boldsymbol{w},b,\boldsymbol{\xi},\boldsymbol{\xi}^*} \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{i=1}^{n}(\xi_i + \xi_i^*) \tag{5.33}$$

subject to:

$$y^{(i)} - \boldsymbol{w}^\top \boldsymbol{x}^{(i)} - b \leq \epsilon + \xi_i \tag{5.34}$$

$$\boldsymbol{w}^\top \boldsymbol{x}^{(i)} + b - y^{(i)} \leq \epsilon + \xi_i^* \tag{5.35}$$

$$\xi_i, \xi_i^* \geq 0 \tag{5.36}$$

5.4 Decision Trees and Ensemble Methods ⊠

Decision trees are intuitive, interpretable models that make predictions by asking a series of yes/no questions about the input features. While individual trees can be prone to overfitting, combining multiple trees through ensemble methods often leads to much better performance.

5.4.1 Intuition and Motivation

Think of a decision tree as a flowchart for making decisions. For example, to classify whether someone will buy a product, you might ask: ”Is their income > $50k?” If yes, ask ”Are they under 30?” If no, ask ”Do they have children?” Each question splits the data into smaller, more homogeneous groups.

The key advantages of decision trees are:

- **Interpretability:** Easy to understand and explain
- **No feature scaling:** Works with mixed data types
- **Handles missing values:** Can deal with incomplete data
- **Non-parametric:** No assumptions about data distribution

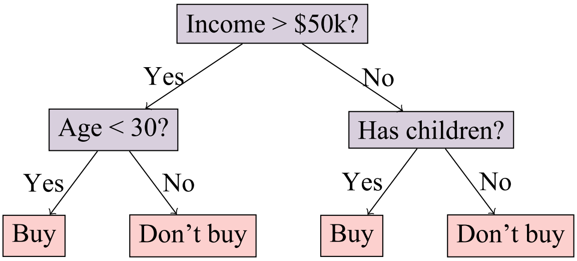

Figure 5.7: A simple decision tree for predicting product purchases. Each internal node represents a decision, and leaf nodes represent the final prediction.

5.4.2 Decision Trees

A **decision tree** recursively partitions the input space based on feature values. The algorithm works by:

1. Starting with all training examples at the root
2. Finding the best feature and threshold to split on
3. Creating child nodes for each split
4. Repeating recursively until a stopping criterion is met
5. Assigning a prediction to each leaf node

Splitting Criteria

The key question is: ”Which feature and threshold should we use to split the data?” We want splits that create the most homogeneous child nodes.

For classification:

Gini impurity:

$$\text{Gini} = 1 - \sum_{k=1}^{K} p_k^2 \tag{5.37}$$

Entropy:

$$\text{Entropy} = -\sum_{k=1}^{K} p_k \log p_k \tag{5.38}$$

For regression:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y^{(i)} - \bar{y})^2 \tag{5.39}$$

where p_k is the proportion of class k examples in a node, and $\bar{y}$ is the mean target value.

Information Gain

The **information gain** measures how much a split reduces impurity:

$$\text{IG} = \text{Impurity(parent)} - \sum_{j} \frac{n_j}{n}\text{Impurity(child}_j) \tag{5.40}$$

We choose the split that maximizes information gain.

5.4.3 Random Forests

Random forests address the overfitting problem of individual trees by combining multiple trees trained on different subsets of the data.

Bootstrap Aggregating (Bagging)

Random forests use two sources of randomness:

1. **Bootstrap sampling:** Each tree is trained on a random sample (with replacement) of the training data

2. **Random feature selection:** At each split, only a random subset of features is considered

Prediction is made by averaging (regression) or voting (classification):

$$\hat{y} = \frac{1}{B}\sum_{b=1}^{B} f_b(\boldsymbol{x}) \tag{5.41}$$

where B is the number of trees.

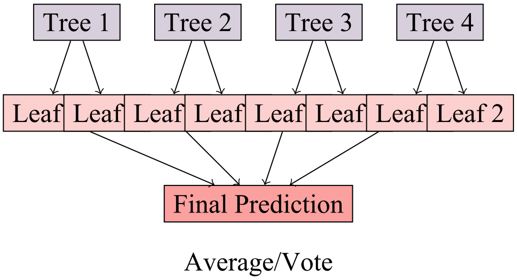

Figure 5.8: Random forests combine multiple decision trees. Each tree is trained on a different bootstrap sample and makes its own prediction. The final prediction is the average (regression) or majority vote (classification) of all trees.

Advantages of Random Forests

- **Reduced overfitting:** Multiple trees reduce variance
- **Feature importance:** Can measure which features are most important
- **Robust to outliers:** Bootstrap sampling reduces outlier impact
- **Parallelizable:** Trees can be trained independently
- **No feature scaling needed:** Works with mixed data types

5.4.4 Gradient Boosting

Gradient boosting builds an ensemble sequentially, where each new tree corrects the errors of the previous ensemble. Unlike random forests, trees are trained one after another.

Algorithm

For iteration m:

1. Compute residuals: $r_i^{(m)} = y^{(i)} - \hat{y}^{(m-1)}(\boldsymbol{x}^{(i)})$

2. Fit tree f_m to residuals
3. Update: $\hat{y}^{(m)} = \hat{y}^{(m-1)} + \nu f_m(\boldsymbol{x}^{(i)})$

where ν is the learning rate (shrinkage parameter).

Intuition

Gradient boosting works by:

1. Start with a simple model (e.g., mean for regression)
2. Calculate the errors (residuals) of the current model
3. Train a new model to predict these errors
4. Add the new model to the ensemble (with a small weight)
5. Repeat until convergence

Example 5.5. For regression with target values $[10, 20, 30, 40]$:

1. Initial prediction: $\hat{y}^{(0)} = 25$ (mean)
2. Residuals: $[-15, -5, 5, 15]$
3. Train tree on residuals
4. Add tree to ensemble: $\hat{y}^{(1)} = 25 + \nu \cdot \text{tree}_1(\boldsymbol{x})$
5. Repeat with new residuals

5.4.5 Advanced Ensemble Methods

AdaBoost

AdaBoost (Adaptive Boosting) is an early boosting algorithm that:

- Assigns higher weights to misclassified examples
- Combines weak learners with weights based on their performance
- Focuses on the hardest examples

XGBoost and LightGBM

Modern gradient boosting implementations like XGBoost and LightGBM add:

- **Regularization:** L1 and L2 penalties
- **Pruning:** Remove splits that don't improve performance
- **Feature subsampling:** Random feature selection at each split
- **Efficient implementation:** Optimized for speed and memory

5.4.6 Comparison of Ensemble Methods

Method	Bias	Variance	Interpretability
Single Tree	Low	High	High
Random Forest	Low	Medium	Medium
Gradient Boosting	Low	Low	Low

Table 5.1: Comparison of tree-based methods. Random forests reduce variance through averaging, while gradient boosting reduces both bias and variance through sequential learning.

5.4.7 Advantages and Limitations

Advantages:

- **Interpretable:** Easy to understand and explain
- **Flexible:** Handles mixed data types and missing values
- **No assumptions:** Works with any data distribution
- **Feature importance:** Can identify important features
- **Robust:** Less sensitive to outliers than linear methods

Limitations:

- **Overfitting:** Individual trees can overfit easily
- **Instability:** Small changes in data can lead to very different trees

- **Computational cost:** Ensemble methods can be slow to train
- **Memory usage:** Storing many trees requires significant memory
- **Less effective with high-dimensional data:** Performance can degrade with many features

5.5 k-Nearest Neighbors ☒

k-Nearest Neighbors (k-NN) is a simple yet powerful non-parametric algorithm that makes predictions based on the similarity to training examples. It's called ”lazy learning” because it doesn't build a model during training—all computation happens at prediction time.

5.5.1 Intuition and Motivation

The k-NN algorithm is based on a simple principle: ”similar things are close to each other.” If you want to predict whether someone will like a movie, look at what movies similar people (with similar tastes) liked. If you want to predict house prices, look at prices of similar houses in the neighborhood.
The key insight is that we can make good predictions by finding the most similar examples in our training data and using their outcomes as a guide.

5.5.2 Algorithm

For a query point $\boldsymbol{x}$:

1. Find the k closest training examples based on a distance metric
2. For classification: return the majority class among the k neighbors
3. For regression: return the average of the target values of the k neighbors

Mathematical Formulation

For classification, the predicted class is:

$$\hat{y} = \arg\max_{c} \sum_{i=1}^{k} \mathbb{I}(y^{(i)} = c) \tag{5.42}$$

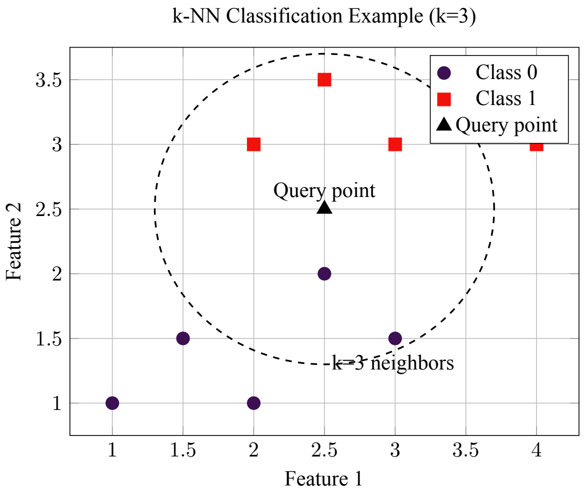

Figure 5.9: k-NN finds the k nearest neighbors to a query point and makes predictions based on their labels. Here, with k=3, the query point would be classified as class 0 (2 out of 3 neighbors are class 0).

For regression, the predicted value is:

$$\hat{y} = \frac{1}{k} \sum_{i=1}^{k} y^{(i)} \tag{5.43}$$

where $y^{(i)}$ are the target values of the k nearest neighbors.

5.5.3 Distance Metrics

The choice of distance metric significantly affects k-NN performance. Here are the most common ones:

Euclidean Distance

$$d(\boldsymbol{x}, \boldsymbol{x}') = \sqrt{\sum_{i=1}^{d} (x_i - x_i')^2} \tag{5.44}$$

This is the most common choice, measuring the straight-line distance between points.

Manhattan Distance

$$d(\boldsymbol{x}, \boldsymbol{x}') = \sum_{i=1}^{d} |x_i - x'_i| \tag{5.45}$$

Also known as L1 distance, it measures the sum of absolute differences. Useful when features have different scales.

Minkowski Distance

$$d(\boldsymbol{x}, \boldsymbol{x}') = \left(\sum_{i=1}^{d} |x_i - x'_i|^p \right)^{1/p} \tag{5.46}$$

This is a generalization where:

- $p = 1$: Manhattan distance
- $p = 2$: Euclidean distance
- $p \to \infty$: Chebyshev distance (maximum coordinate difference)

5.5.4 Choosing k

The choice of k is crucial and involves a bias-variance trade-off:

- **Small k (e.g., k=1):**
  - Low bias, high variance
  - Flexible decision boundary
  - Sensitive to noise and outliers
  - May overfit to training data
- **Large k (e.g., k=n):**
  - High bias, low variance
  - Smooth decision boundary
  - May miss local patterns
  - Underfits the data

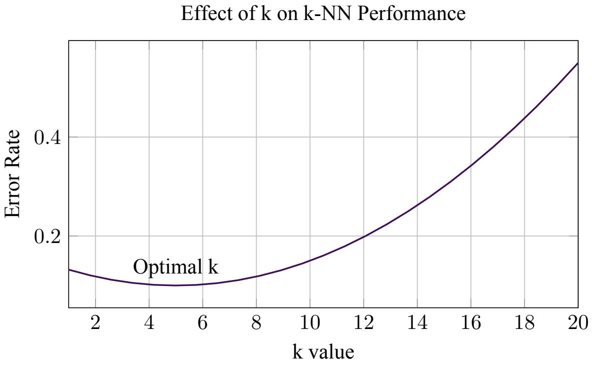

Figure 5.10: The effect of k on k-NN performance. Small k leads to high variance (overfitting), while large k leads to high bias (underfitting). The optimal k is typically found through cross-validation.

5.5.5 Weighted k-NN

Instead of giving equal weight to all k neighbors, we can weight them by their distance:

$$\hat{y} = \frac{\sum_{i=1}^{k} w_i y^{(i)}}{\sum_{i=1}^{k} w_i} \tag{5.47}$$

where $w_i = \frac{1}{d(\boldsymbol{x}, \boldsymbol{x}^{(i)})}$ is the weight based on distance.

5.5.6 Computational Considerations

k-NN has several computational characteristics:

Training Time

- **No training time:** k-NN is a lazy learner
- **Memory usage:** Must store all training examples
- **Scalability:** Performance degrades with large datasets

Prediction Time

- **Naive approach:** $O(n \cdot d)$ for each prediction
- **Optimized approaches:** Can be reduced using data structures

Speedup Techniques

KD-Trees:

- Partition space into regions
- Reduce search time to $O(\log n)$ in low dimensions
- Less effective in high dimensions

Ball Trees:

- Use hyperspheres instead of hyperplanes
- Better for high-dimensional data
- More complex to implement

Locality Sensitive Hashing (LSH):

- Approximate nearest neighbor search
- Significant speedup for very large datasets
- May sacrifice some accuracy

5.5.7 Advantages and Limitations

Advantages:

- **Simple:** Easy to understand and implement
- **No assumptions:** Works with any data type
- **Non-parametric:** No model to fit
- **Local patterns:** Can capture complex decision boundaries
- **Incremental:** Easy to add new training examples

Limitations:

- **Computational cost:** Slow for large datasets
- **Memory usage:** Must store all training data
- **Sensitive to irrelevant features:** All features are treated equally
- **Curse of dimensionality:** Performance degrades in high dimensions
- **No interpretability:** Hard to understand why a prediction was made

5.5.8 Curse of Dimensionality

In high-dimensional spaces, k-NN faces the **curse of dimensionality**:

- **Distance concentration:** All points become roughly equidistant
- **Empty space:** Most of the space is empty
- **Irrelevant features:** Performance degrades with irrelevant features

Example 5.6. In a 1000-dimensional space, even if only 10 features are relevant, the 990 irrelevant features can dominate the distance calculations, making k-NN ineffective.

5.5.9 Feature Selection and Scaling

To improve k-NN performance:

- **Feature selection:** Remove irrelevant features
- **Feature scaling:** Normalize features to the same scale
- **Dimensionality reduction:** Use PCA or other techniques
- **Distance weighting:** Weight features by importance

5.6 Comparison with Deep Learning ☒

Understanding the relationship between classical machine learning methods and deep learning is crucial for choosing the right approach for your problem. While deep learning has achieved remarkable success in many domains, classical methods still have important advantages in certain scenarios.

5.6.1 When Classical Methods Excel

Classical machine learning methods have several advantages that make them preferable in certain situations:

Interpretability and Debugging

- **Linear models:** Coefficients directly show feature importance
- **Decision trees:** Rules are human-readable
- **SVM:** Support vectors provide insight into decision boundary
- **Easier debugging:** Can trace predictions step by step

Small to Medium Datasets

- **Less prone to overfitting:** Classical methods have fewer parameters
- **Faster training:** Require less computational resources
- **Better generalization:** Often perform better with limited data
- **No need for data augmentation:** Work well with original data

Computational Efficiency

- **Lower memory requirements:** Don't need to store large networks
- **Faster inference:** Simple mathematical operations
- **No GPU required:** Can run on standard hardware
- **Real-time applications:** Suitable for embedded systems

5.6.2 When Deep Learning Excels

Deep learning addresses several fundamental limitations of classical methods:

Automatic Feature Learning

- **No manual feature engineering:** Networks learn relevant features automatically
- **Hierarchical representations:** Lower layers learn simple features, higher layers learn complex combinations
- **End-to-end learning:** Single model handles feature extraction and classification
- **Adaptive features:** Features adapt to the specific problem

Scalability with Data and Model Size

- **Data scalability:** Performance typically improves with more data
- **Model capacity:** Can handle very large models with millions of parameters
- **Distributed training:** Can leverage multiple GPUs/TPUs
- **Transfer learning:** Pre-trained models can be fine-tuned for new tasks

Handling Complex Data Types

- **Images:** Convolutional networks excel at computer vision
- **Text:** Recurrent and transformer networks handle natural language
- **Audio:** Can process raw audio signals
- **Multimodal:** Can combine different data types

5.6.3 Performance Comparison

Aspect	Classical ML	Deep Learning	Best Use Case
Interpretability	High	Low	Medical diagnosis, finance
Training Speed	Fast	Slow	Prototyping, small datasets
Inference Speed	Fast	Medium	Real-time applications
Data Requirements	Low	High	Small companies, research
Computational Cost	Low	High	Resource-constrained environments
Feature Engineering	Manual	Automatic	Complex domains

Table 5.2: Comparison of classical machine learning and deep learning across different aspects. The choice depends on the specific requirements of your problem.

5.6.4 Hybrid Approaches

In practice, the best solutions often combine classical and deep learning methods:

Feature Engineering with Deep Learning

- Use deep networks to extract features
- Apply classical methods (SVM, random forest) on extracted features

- Combine interpretability of classical methods with representation power of deep learning

Ensemble Methods

- Combine predictions from classical and deep learning models
- Use classical methods for interpretable components
- Use deep learning for complex pattern recognition

Two-Stage Approaches

- Use classical methods for initial screening
- Apply deep learning for final classification
- Balance efficiency and accuracy

5.6.5 Choosing the Right Approach

The choice between classical and deep learning methods depends on several factors:

Data Characteristics

- **Small dataset (< 10k examples):** Classical methods often better
- **Medium dataset (10k-100k examples):** Both approaches viable
- **Large dataset (> 100k examples):** Deep learning typically better
- **High-dimensional data:** Deep learning excels
- **Structured data:** Classical methods often sufficient

Problem Requirements

- **Interpretability needed:** Classical methods preferred
- **Real-time inference:** Classical methods often faster
- **Complex patterns:** Deep learning better
- **Unstructured data:** Deep learning necessary

Resource Constraints

- **Limited computational resources:** Classical methods
- **Limited labeled data:** Classical methods or transfer learning
- **Need for quick prototyping:** Classical methods
- **Production deployment:** Consider inference costs

5.6.6 Future Directions

The field continues to evolve with several promising directions:

Automated Machine Learning (AutoML)

- **Neural Architecture Search (NAS):** Automatically design network architectures
- **Hyperparameter optimization:** Automatically tune classical methods
- **Model selection:** Automatically choose between classical and deep learning

Explainable AI

- **SHAP values:** Explain predictions from any model
- **LIME:** Local interpretable model-agnostic explanations
- **Attention mechanisms:** Understand what deep networks focus on

Efficient Deep Learning

- **Model compression:** Reduce model size while maintaining performance
- **Quantization:** Use lower precision arithmetic
- **Knowledge distillation:** Transfer knowledge from large to small models

5.6.7 Conclusion

Classical machine learning methods and deep learning are not competing approaches but complementary tools in the machine learning toolkit. The key is to understand the strengths and limitations of each approach and choose the right tool for your specific problem.

- **Start simple:** Begin with classical methods for baseline performance
- **Consider complexity:** Only use deep learning if classical methods are insufficient
- **Think about requirements:** Consider interpretability, speed, and resource constraints
- **Combine approaches:** Use hybrid methods when appropriate
- **Stay updated:** The field continues to evolve rapidly

The goal is not to use the most complex method, but to use the most appropriate method for your specific problem and constraints.

Key Takeaways

Key Takeaways 5

- **Classical algorithms** like linear regression, logistic regression, and SVMs provide interpretable baselines for machine learning tasks.
- **Trade-offs exist** between model complexity, interpretability, and performance; classical methods excel in low-data regimes.
- **Regularisation** (L1/L2) prevents overfitting and enables feature selection in high-dimensional settings.
- **Ensemble methods** combine weak learners to improve accuracy and robustness through variance reduction.
- **Understanding limitations** of classical methods motivates deep learning's hierarchical representation learning.

Problems

Easy Problems

Problem 5.7 (Linear Regression Basics)**.** Given the dataset $\{(1,2),(2,4),(3,6),(4,8)\}$, find the linear regression model $\hat{y} = wx + b$ that minimises the mean squared error.
Hint: Use the normal equation $\boldsymbol{w}^* = (\boldsymbol{X}^\top \boldsymbol{X})^{-1}\boldsymbol{X}^\top \boldsymbol{y}$ where $\boldsymbol{X}$ includes a column of ones for the bias term.

Problem 5.8 (Logistic Regression Decision Boundary)**.** For a logistic regression model with weights $\boldsymbol{w} = [2, -1]$ and bias $b = 0$, find the decision boundary equation and classify the point $(1, 1)$.
Hint: The decision boundary occurs where $P(y = 1|\boldsymbol{x}) = 0.5$, which means $\boldsymbol{w}^\top \boldsymbol{x} + b = 0$.

Problem 5.9 (Decision Tree Splitting)**.** Given a node with 10 examples: 6 belong to class A and 4 to class B, calculate the Gini impurity and entropy.
Hint: Gini impurity = $1 - \sum_k p_k^2$ and entropy = $-\sum_k p_k \log p_k$ where p_k is the proportion of class k.

Problem 5.10 (Regularisation Effect)**.** Explain why L1 regularisation (Lasso) can drive some weights to exactly zero, while L2 regularisation (Ridge) cannot.
Hint: Consider the shape of the L1 and L2 penalty functions and their derivatives at zero.

Medium Problems

Problem 5.11 (Ridge Regression Derivation)**.** Derive the closed-form solution for ridge regression: $\boldsymbol{w}^* = (\boldsymbol{X}^\top \boldsymbol{X} + \lambda \boldsymbol{I})^{-1}\boldsymbol{X}^\top \boldsymbol{y}$.
Hint: Start with the ridge regression objective function $L(\boldsymbol{w}) = \|\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}\|^2 + \lambda\|\boldsymbol{w}\|^2$ and take the gradient with respect to $\boldsymbol{w}$.

Problem 5.12 (Random Forest Bias-Variance)**.** Explain how random forests reduce variance compared to a single decision tree. What happens to bias?
Hint: Consider how averaging multiple models affects the bias and variance of the ensemble.

Hard Problems

Problem 5.13 (SVM Kernel Trick)**.** Prove that the polynomial kernel $k(\boldsymbol{x}, \boldsymbol{x}') = (\boldsymbol{x}^\top \boldsymbol{x}' + c)^d$ is a valid kernel function by showing it corresponds to an inner product in some feature space.
Hint: Expand the polynomial and show it can be written as $\phi(\boldsymbol{x})^\top \phi(\boldsymbol{x}')$ for some feature map ϕ.

Problem 5.14 (Ensemble Methods Theory)**.** Prove that for an ensemble of B independent models with error rate $p < 0.5$, the ensemble error rate approaches 0 as $B \to \infty$.
Hint: Use the binomial distribution and the fact that the ensemble makes an error only when more than half of the models are wrong.

Part II

Practical Deep Networks

Chapter 6

Deep Feedforward Networks

This chapter introduces deep feedforward neural networks, also known as multilayer perceptrons (MLPs). These are the fundamental building blocks of deep learning.

Learning Objectives

After studying this chapter, you will be able to:

1. **Understand the architecture** of feedforward neural networks and how they process information from input to output.

2. **Master activation functions** and their role in introducing non-linearity, including the trade-offs between different choices.

3. **Design appropriate output layers** and loss functions for different types of machine learning tasks (regression, binary classification, multiclass classification).

4. **Derive and implement backpropagation** to efficiently compute gradients for training neural networks.

5. **Make informed design choices** about network architecture, initialization, and training procedures.

6. **Apply theoretical knowledge** to solve practical problems involving feedforward networks.

6.1 Introduction to Feedforward Networks ☒

6.1.1 Intuition: What is a Feedforward Network?

Imagine you’re trying to recognize handwritten digits. A feedforward neural network is like having a team of experts, each specialized in detecting different features:

- **First layer experts** look for basic patterns like edges, curves, and lines
- **Middle layer experts** combine these basic patterns to detect more complex shapes like loops, corners, and curves
- **Final layer experts** combine these complex shapes to make the final decision: ”This is a 3” or ”This is a 7”

The key insight is that each layer builds upon the previous one, creating increasingly sophisticated representations. This hierarchical approach mirrors how our own visual system works, from detecting simple edges to recognizing complex objects.
A **feedforward neural network** approximates a function f^*. For input $\boldsymbol{x}$, the network computes $y = f(\boldsymbol{x}; \boldsymbol{\theta})$ and learns parameters $\boldsymbol{\theta}$ such that $f \approx f^*$.

6.1.2 Network Architecture

A feedforward network consists of layers:

- **Input layer:** receives raw features $\boldsymbol{x}$
- **Hidden layers:** intermediate representations $\boldsymbol{h}^{(1)}, \boldsymbol{h}^{(2)}, \ldots$
- **Output layer:** produces predictions $\hat{y}$

For a network with L layers:

$$\boldsymbol{h}^{(l)} = \sigma(\boldsymbol{W}^{(l)} \boldsymbol{h}^{(l-1)} + \boldsymbol{b}^{(l)}) \tag{6.1}$$

where $\boldsymbol{h}^{(0)} = \boldsymbol{x}$, $\boldsymbol{W}^{(l)}$ are weights, $\boldsymbol{b}^{(l)}$ are biases, and σ is an activation function.

6.1.3 Forward Propagation

The computation proceeds from input to output:

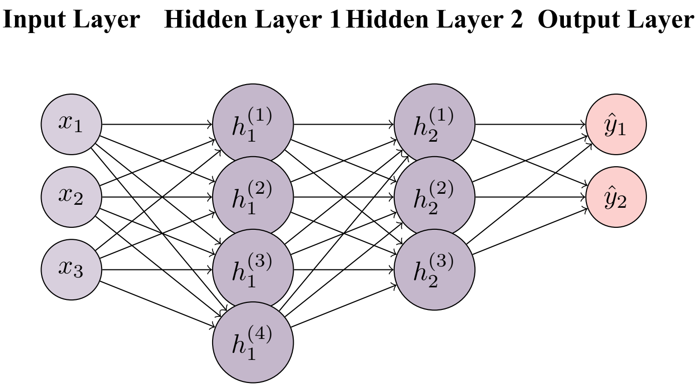

Figure 6.1: Architecture of a feedforward neural network with 2 hidden layers. Each circle represents a neuron, and arrows show the flow of information from input to output.

$$\boldsymbol{z}^{(1)} = \boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)} \tag{6.2}$$

$$\boldsymbol{h}^{(1)} = \sigma(\boldsymbol{z}^{(1)}) \tag{6.3}$$

$$\boldsymbol{z}^{(2)} = \boldsymbol{W}^{(2)}\boldsymbol{h}^{(1)} + \boldsymbol{b}^{(2)} \tag{6.4}$$

$$\vdots \tag{6.5}$$

$$\hat{y} = \boldsymbol{h}^{(L)} \tag{6.6}$$

Example 6.1 (Simple Forward Pass)**.** Consider a simple network with 2 inputs, 2 hidden neurons, and 1 output for binary classification:

- Input: $\boldsymbol{x} = [1, 0.5]$
- Weights to hidden layer: $\boldsymbol{W}^{(1)} = \begin{bmatrix} 0.5 & -0.3 \\ 0.8 & 0.2 \end{bmatrix}$
- Bias: $\boldsymbol{b}^{(1)} = [0.1, -0.2]$
- Activation: ReLU

Step 1: Compute pre-activation

$$\begin{aligned}
\boldsymbol{z}^{(1)} &= \boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)} && (6.7)\\
&= \begin{bmatrix} 0.5 & -0.3 \\ 0.8 & 0.2 \end{bmatrix} \begin{bmatrix} 1 \\ 0.5 \end{bmatrix} + \begin{bmatrix} 0.1 \\ -0.2 \end{bmatrix} && (6.8)\\
&= \begin{bmatrix} 0.5 \cdot 1 + (-0.3) \cdot 0.5 + 0.1 \\ 0.8 \cdot 1 + 0.2 \cdot 0.5 + (-0.2) \end{bmatrix} && (6.9)\\
&= \begin{bmatrix} 0.45 \\ 0.7 \end{bmatrix} && (6.10)
\end{aligned}$$

Step 2: Apply activation function

$$\boldsymbol{h}^{(1)} = \text{ReLU}(\boldsymbol{z}^{(1)}) = \begin{bmatrix} \max(0, 0.45) \\ \max(0, 0.7) \end{bmatrix} = \begin{bmatrix} 0.45 \\ 0.7 \end{bmatrix} \tag{6.11}$$

The hidden layer has learned to represent the input in a transformed space where both neurons are active (positive values).

6.1.4 Universal Approximation

The **universal approximation theorem** states that a feedforward network with a single hidden layer containing a finite number of neurons can approximate any continuous function on a compact subset of $\mathbb{R}^n$, given appropriate activation functions. However, deeper networks often learn more efficiently.

6.2 Activation Functions ☒

6.2.1 Intuition: Why Do We Need Activation Functions?

Imagine you’re building a house with only straight lines and right angles. No matter how many rooms you add, you can only create rectangular spaces. But what if you want curved walls, arches, or domes? You need curved tools!

Similarly, without activation functions, neural networks can only learn linear relationships, no matter how many layers you add. Activation functions are the ”curved tools” that allow networks to learn non-linear patterns.

Think of an activation function as a decision maker:

- **Input:** A weighted sum of information from other neurons
- **Decision:** How much should this neuron ”fire” or contribute to the next layer?
- **Output:** A transformed value that becomes input to the next layer

Activation functions introduce non-linearity, enabling networks to learn complex patterns.

6.2.2 Sigmoid

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{6.12}$$

Properties:

- Range: $(0, 1)$
- Saturates for large $|z|$ (vanishing gradients)
- Not zero-centered
- Historically important but less common in hidden layers

6.2.3 Hyperbolic Tangent (tanh)

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{6.13}$$

Properties:

- Range: $(-1, 1)$
- Zero-centered
- Still suffers from saturation

6.2.4 Rectified Linear Unit (ReLU)

$$\text{ReLU}(z) = \max(0, z) \tag{6.14}$$

Properties:

- Simple and computationally efficient
- No saturation for positive values

- Can cause ”dead neurons” (always output 0)
- Most widely used activation

6.2.5 Leaky ReLU and Variants

Leaky ReLU:

$$\text{LeakyReLU}(z) = \max(\alpha z, z), \quad \alpha \ll 1 \tag{6.15}$$

Parametric ReLU (PReLU):

$$\text{PReLU}(z) = \max(\alpha z, z) \tag{6.16}$$

where α is learned.

Exponential Linear Unit (ELU):

$$\text{ELU}(z) = \begin{cases} z & \text{if } z > 0 \\ \alpha(e^z - 1) & \text{if } z \leq 0 \end{cases} \tag{6.17}$$

6.2.6 Swish and GELU

Swish:

$$\text{Swish}(z) = z \cdot \sigma(z) \tag{6.18}$$

Gaussian Error Linear Unit (GELU):

$$\text{GELU}(z) = z \cdot \Phi(z) \tag{6.19}$$

where Φ is the Gaussian CDF. Used in modern transformers.

6.3 Output Units and Loss Functions ☒

6.3.1 Intuition: Matching Output to Task

Think of the output layer as the ”final decision maker” in your network. Just like different jobs require different tools, different machine learning tasks require different output formats:

- **Regression (predicting prices):** You want a real number. ”This house costs $250,000”

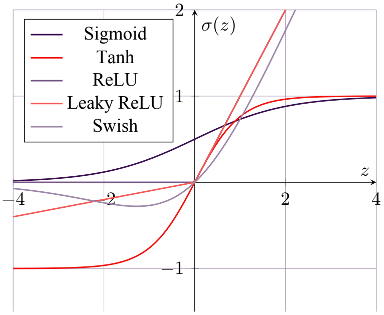

Figure 6.2: Comparison of common activation functions. Each function has different properties: sigmoid and tanh saturate at extreme values, ReLU is simple but can ”die”, while Swish provides smooth gradients.

- **Binary Classification (spam detection):** You want a probability. ”This email is 95% likely to be spam”
- **Multiclass Classification (image recognition):** You want probabilities for each class. ”This image is 80% cat, 15% dog, 5% bird”

The loss function is like a ”teacher” that tells the network how wrong it is. A good teacher gives clear, helpful feedback that guides learning in the right direction.
The choice of output layer and loss function depends on the task.

6.3.2 Linear Output for Regression

For regression, use linear output:

$$\hat{y} = \boldsymbol{W}^\top \boldsymbol{h} + b \tag{6.20}$$

with MSE loss:

$$L = \frac{1}{n}\sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2 \tag{6.21}$$

6.3.3 Sigmoid Output for Binary Classification

For binary classification:

$$\hat{y} = \sigma(\boldsymbol{W}^\top \boldsymbol{h} + b) \tag{6.22}$$

with binary cross-entropy loss:

$$L = -\frac{1}{n}\sum_{i=1}^{n}[y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})] \tag{6.23}$$

6.3.4 Softmax Output for Multiclass Classification

For K classes:

$$\hat{y}_k = \frac{\exp(z_k)}{\sum_{j=1}^{K} \exp(z_j)} \tag{6.24}$$

with categorical cross-entropy loss:

$$L = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K} y_k^{(i)} \log \hat{y}_k^{(i)} \tag{6.25}$$

6.4 Backpropagation ☒

6.4.1 Intuition: Learning from Mistakes

Imagine you’re learning to play basketball. After each shot, you need to know:

- How far off was your shot? (the error)
- Which part of your technique needs adjustment? (which parameters to change)
- How much should you adjust each part? (how much to change each parameter)

Backpropagation is like having a coach who watches your shot and tells you exactly what to adjust:

- ”Your elbow was too high” (gradient for elbow angle)
- ”You need to follow through more” (gradient for follow-through)
- ”Your timing was off” (gradient for release timing)

The key insight is that we can efficiently compute how much each parameter contributed to the final error by working backwards through the network. **Backpropagation** efficiently computes gradients using the chain rule.

6.4.2 The Chain Rule

For composition $f(g(x))$:

$$\frac{df}{dx} = \frac{df}{dg}\frac{dg}{dx} \tag{6.26}$$

For vectors, we use the Jacobian:

$$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} = \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{z}}\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}} \tag{6.27}$$

6.4.3 Backward Pass

Starting from the loss L, we compute gradients layer by layer:

$$\delta^{(L)} = \nabla_{\boldsymbol{h}^{(L)}} L \tag{6.28}$$

$$\delta^{(l)} = (\boldsymbol{W}^{(l+1)})^\top \delta^{(l+1)} \odot \sigma'(\boldsymbol{z}^{(l)}) \tag{6.29}$$

where $\odot$ denotes element-wise multiplication.
Parameter gradients:

$$\frac{\partial L}{\partial \boldsymbol{W}^{(l)}} = \delta^{(l)}(\boldsymbol{h}^{(l-1)})^\top \tag{6.30}$$

$$\frac{\partial L}{\partial \boldsymbol{b}^{(l)}} = \delta^{(l)} \tag{6.31}$$

6.4.4 Computational Graph

Modern frameworks use automatic differentiation on computational graphs:

- **Forward mode:** efficient when outputs $\gg$ inputs
- **Reverse mode (backprop):** efficient when inputs $\gg$ outputs

6.5 Design Choices ☒

6.5.1 Intuition: Building the Right Network

Designing a neural network is like designing a building:

- **Depth (layers):** Like floors in a building - more floors can house more complex functions, but they're harder to build and maintain

- **Width (neurons per layer):** Like rooms per floor - more rooms give more space, but you need to fill them efficiently
- **Initialization:** Like the foundation - if it's wrong, the whole building might collapse
- **Training:** Like the construction process - you need the right tools and techniques

The key is finding the right balance for your specific task and data.

6.5.2 Network Depth and Width

Depth (number of layers):

- Deeper networks can learn more complex functions
- Can be harder to optimize (vanishing/exploding gradients)
- Modern techniques enable very deep networks (100+ layers)

Width (neurons per layer):

- Wider networks have more capacity
- Trade-off between width and depth
- Very wide shallow networks vs. narrow deep networks

6.5.3 Weight Initialization

Poor initialization can prevent learning. Common strategies:
Xavier/Glorot initialization:

$$W_{ij} \sim \mathcal{N}\left(0, \frac{2}{n_{\text{in}} + n_{\text{out}}}\right) \tag{6.32}$$

He initialization (for ReLU):

$$W_{ij} \sim \mathcal{N}\left(0, \frac{2}{n_{\text{in}}}\right) \tag{6.33}$$

6.5.4 Batch Training

Mini-batch gradient descent:

- Compute gradients on small batches (typically 32-256 examples)
- Provides regularization through noise
- Enables efficient parallel computation
- Balances between SGD and full-batch GD

6.6 Real World Applications

Deep feedforward networks serve as the foundation for many practical applications across industries. Here we explore how these networks solve real-world problems in accessible, less technical terms.

6.6.1 Medical Diagnosis Support

Feedforward networks help doctors make better decisions by analyzing patient data. For example:

- **Disease prediction:** Networks analyze patient symptoms, medical history, and test results to predict the likelihood of diseases like diabetes or heart disease. The network learns patterns from thousands of past patient records to help identify at-risk individuals early.
- **Treatment recommendations:** By learning from successful treatment outcomes, these networks can suggest personalized treatment plans based on a patient's unique characteristics, improving recovery rates and reducing side effects.
- **Drug dosage optimization:** Networks help determine optimal medication dosages by considering factors like patient weight, age, kidney function, and drug interactions, reducing risks of under or over-medication.

6.6.2 Financial Fraud Detection

Banks and financial institutions use feedforward networks to protect customers from fraud:

- **Credit card fraud detection:** Networks analyze transaction patterns in real-time, flagging unusual purchases (like expensive items bought far from home) within milliseconds. This happens seamlessly as you shop, protecting your account without interrupting legitimate purchases.

- **Loan default prediction:** Before approving loans, networks evaluate applicant information to predict repayment likelihood. This helps banks make fairer lending decisions while reducing financial risks.

- **Insurance claim verification:** Networks identify suspicious insurance claims by detecting patterns inconsistent with typical legitimate claims, saving companies billions while ensuring honest customers get quick payouts.

6.6.3 Product Recommendation Systems

Online platforms use feedforward networks to personalize your experience:

- **E-commerce recommendations:** When shopping online, networks analyze your browsing history, purchase patterns, and preferences to suggest products you're likely to enjoy. This makes shopping more efficient and helps you discover new items.

- **Content recommendations:** Streaming services use these networks to suggest movies, shows, or music based on what you've watched or listened to before. The network learns your taste profile and finds content matching your preferences.

- **Targeted advertising:** Networks help businesses show you relevant ads by understanding your interests and needs. This benefits both consumers (seeing useful products) and businesses (reaching interested customers).

6.6.4 Why These Applications Work

Feedforward networks excel at these tasks because they can:

- Learn complex patterns from historical data

- Make decisions quickly once trained
- Handle multiple input features simultaneously
- Generalize to new, unseen situations

These applications demonstrate how deep learning moves from theory to practice, improving everyday life in ways both visible and behind-the-scenes.

6.7 Problems ⊠

This section provides exercises to reinforce your understanding of feedforward networks. Problems are categorized by difficulty level, and hints are provided for each problem.

6.7.1 Easy Problems (6 problems)

Problem 6.2 (Forward Pass Calculation)**.** Consider a simple neural network with:

- Input layer: 2 neurons with values $x_1 = 1, x_2 = -0.5$
- Hidden layer: 2 neurons with weights $\boldsymbol{W} = \begin{bmatrix} 0.5 & -0.3 \\ 0.8 & 0.2 \end{bmatrix}$ and bias $\boldsymbol{b} = [0.1, -0.2]$
- Activation function: ReLU

Calculate the output of the hidden layer.
Hint: Apply the formula $\boldsymbol{h} = \text{ReLU}(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$ step by step.

Problem 6.3 (Activation Function Properties)**.** For each activation function, state whether it is:

1. Bounded or unbounded
2. Monotonic or non-monotonic
3. Zero-centered or not zero-centered

Functions: (a) Sigmoid, (b) Tanh, (c) ReLU, (d) Leaky ReLU
Hint: Consider the mathematical properties and plot each function to understand their behavior.

Problem 6.4 (Loss Function Selection)**.** Match each task with the appropriate loss function:

1. Predicting house prices
2. Classifying emails as spam/not spam
3. Recognizing digits 0-9
4. Predicting stock market direction (up/down)

Loss functions: MSE, Binary Cross-entropy, Categorical Cross-entropy
Hint: Consider the output format needed for each task and the mathematical properties of each loss function.

Problem 6.5 (Network Architecture)**.** Design a neural network architecture for each task:

1. Binary classification with 10 input features
2. Regression with 5 input features
3. 3-class classification with 8 input features

Specify the number of layers, neurons per layer, and activation functions.
Hint: Consider the complexity of each task and the output requirements.

Problem 6.6 (Gradient Calculation)**.** For the function $f(x, y) = x^2 + 2xy + y^2$, calculate:

1. $\frac{\partial f}{\partial x}$
2. $\frac{\partial f}{\partial y}$
3. The gradient vector ∇f

Hint: Use the power rule and product rule for differentiation.

Problem 6.7 (Chain Rule Application)**.** Given $f(x) = (x^2 + 1)^3$, find $\frac{df}{dx}$ using the chain rule.
Hint: Let $u = x^2 + 1$, then $f = u^3$. Apply the chain rule: $\frac{df}{dx} = \frac{df}{du} \cdot \frac{du}{dx}$.

6.7.2 Medium Problems (5 problems)

Problem 6.8 (Backpropagation Derivation)**.** Derive the backpropagation equations for a 2-layer network with:

- Input: $\boldsymbol{x} \in \mathbb{R}^d$
- Hidden layer: $\boldsymbol{h} = \sigma(\boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{b}_1)$
- Output: $\hat{y} = \boldsymbol{W}_2\boldsymbol{h} + \boldsymbol{b}_2$
- Loss: $L = \frac{1}{2}(\hat{y} - y)^2$

Find $\frac{\partial L}{\partial \boldsymbol{W}_1}$, $\frac{\partial L}{\partial \boldsymbol{W}_2}$, $\frac{\partial L}{\partial \boldsymbol{b}_1}$, and $\frac{\partial L}{\partial \boldsymbol{b}_2}$.
Hint: Use the chain rule systematically, starting from the loss and working backwards through each layer.

Problem 6.9 (Vanishing Gradient Analysis)**.** Consider a deep network with sigmoid activation functions. Show that the gradient can vanish exponentially with depth.
Hint: The derivative of sigmoid is $\sigma'(z) = \sigma(z)(1 - \sigma(z))$. Since $\sigma(z) \in (0, 1)$, the derivative is bounded by $\frac{1}{4}$. What happens when you multiply many such terms?

Problem 6.10 (Weight Initialization)**.** Derive the Xavier initialization for a layer with n_{in} inputs and n_{out} outputs, assuming the weights are drawn from a normal distribution.
Hint: Consider the variance of the output when inputs have unit variance. For the output to also have unit variance, what should be the variance of the weights?

Problem 6.11 (Universal Approximation)**.** Explain why a single hidden layer with sigmoid activation can approximate any continuous function, but in practice, deeper networks are often preferred.
Hint: Consider the difference between theoretical capacity and practical learnability. What are the computational and optimization challenges?

Problem 6.12 (Activation Function Comparison)**.** Compare ReLU and sigmoid activation functions in terms of:

1. Computational efficiency
2. Gradient flow
3. Saturation behavior

4. Zero-centering

When would you choose each one?
Hint: Consider the mathematical properties, training dynamics, and practical considerations for each activation function.

6.7.3 Hard Problems (5 problems)

Problem 6.13 (Deep Network Optimization)**.** Analyze why very deep networks are difficult to train and discuss modern solutions (residual connections, batch normalization, etc.).
Hint: Consider the relationship between network depth, gradient flow, and optimization landscape. How do modern techniques address these challenges?

Problem 6.14 (Loss Function Design)**.** Design a custom loss function for a multi-task learning problem where you need to:

1. Classify images into 10 categories
2. Predict bounding boxes for objects
3. Estimate object confidence scores

Hint: Consider how to combine different loss functions, handle different scales, and balance the importance of each task.

Problem 6.15 (Network Capacity Analysis)**.** Analyze the relationship between network capacity (number of parameters) and generalization. When does more capacity help, and when does it hurt?
Hint: Consider the bias-variance tradeoff, the role of regularization, and the relationship between model complexity and data size.

Problem 6.16 (Gradient Flow Analysis)**.** Derive the gradient flow equations for a network with skip connections (residual blocks) and analyze how they help with training.
Hint: Consider how skip connections affect the gradient computation and what this means for the optimization dynamics.

Problem 6.17 (Architecture Search)**.** Design a systematic approach to find the optimal network architecture for a given dataset and task. Consider:

1. Search space definition
2. Evaluation strategy
3. Computational constraints
4. Generalization considerations

Hint: Consider automated machine learning (AutoML) techniques, neural architecture search (NAS), and the tradeoffs between search efficiency and architecture quality.

Key Takeaways

Key Takeaways 6

- **Feedforward networks** compose layers of linear transformations and nonlinear activations to learn complex functions.
- **Activation functions** introduce nonlinearity; ReLU and variants balance expressiveness with gradient flow.
- **Backpropagation** efficiently computes gradients via the chain rule, enabling gradient-based optimisation.
- **Output layers and losses** must match the task: softmax/cross-entropy for classification, linear/MSE for regression.
- **Design choices** (depth, width, initialisation) profoundly affect training dynamics and generalisation.

Problems

Easy

Problem 6.18 (Activation Functions)**.** Compare ReLU and sigmoid activation functions. List two advantages of ReLU over sigmoid for hidden layers in deep networks.

Hint: Consider gradient flow, computational efficiency, and the vanishing gradient problem.

Problem 6.19 (Network Capacity)**.** A feedforward network has an input layer with 10 neurons, two hidden layers with 20 neurons each, and an output layer with 3 neurons. Calculate the total number of parameters (weights and biases).
Hint: For each layer transition, count weights and biases separately.

Problem 6.20 (Output Layer Design)**.** For a binary classification task, what activation function and loss function would you use for the output layer? Justify your choice.
Hint: Think about probability outputs and the relationship between binary cross-entropy and sigmoid activation.

Problem 6.21 (Backpropagation Basics)**.** Explain in simple terms why backpropagation is more efficient than computing gradients using finite differences for each parameter.
Hint: Consider the number of forward passes required and the chain rule of calculus.

Medium

Problem 6.22 (Gradient Computation)**.** For a simple network with one hidden layer: $\boldsymbol{h} = \sigma(\boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{b}_1)$ and $\boldsymbol{y} = \boldsymbol{W}_2\boldsymbol{h} + \boldsymbol{b}_2$, derive the gradient $\frac{\partial L}{\partial \boldsymbol{W}_1}$ for mean squared error loss.
Hint: Apply the chain rule: $\frac{\partial L}{\partial \boldsymbol{W}_1} = \frac{\partial L}{\partial \boldsymbol{y}}\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{h}}\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{W}_1}$.

Problem 6.23 (Universal Approximation)**.** The universal approximation theorem states that a feedforward network with a single hidden layer can approximate any continuous function. Discuss why we still use deep networks with multiple layers in practice.
Hint: Consider efficiency of representation, number of neurons needed, and hierarchical feature learning.

Hard

Problem 6.24 (Xavier Initialisation)**.** Derive the Xavier (Glorot) initialisation scheme for weights. Explain why it helps maintain variance of activations across layers.
Hint: Start with the variance of layer outputs and the assumption that inputs and weights are independent.

Problem 6.25 (Residual Connections)**.** Analyse how residual connections (skip connections) help with gradient flow in very deep networks. Derive the gradient through a residual block.

Hint: Consider $\boldsymbol{y} = \boldsymbol{x} + F(\boldsymbol{x})$ and compute $\frac{\partial L}{\partial \boldsymbol{x}}$ where F is a sub-network.

Chapter 7

Regularization for Deep Learning

This chapter explores techniques to improve generalization and prevent overfitting in deep neural networks. Regularization helps models perform well on unseen data.

Learning Objectives

After completing this chapter, you will be able to:

1. **Explain why regularization is needed** and how it improves generalization.
2. **Compare norm penalties** (L1, L2, Elastic Net) and identify when to use each.
3. **Apply data augmentation** strategies across vision, text, and audio tasks.
4. **Implement early stopping** and understand its interaction with optimization.
5. **Use dropout and normalization** layers and reason about their effects.
6. **Evaluate advanced techniques** such as label smoothing, mixup, adversarial training, and gradient clipping.

7.1 Parameter Norm Penalties ☒

Parameter norm penalties constrain model capacity by penalizing large weights.

7.1.1 Intuition: Shrinking the Model's "Complexity"

Think of a model as a musical band with many instruments (parameters). If every instrument plays loudly (large weights), the result can be noisy and overfit to the training song. Norm penalties are like asking the band to lower the volume uniformly (L2) or mute many instruments entirely (L1) so the melody (true signal) stands out. This discourages memorization and encourages simpler patterns that generalize.

7.1.2 L2 Regularization (Weight Decay)

Add squared L2 norm of weights to the loss:

$$\tilde{L}(\boldsymbol{\theta}) = L(\boldsymbol{\theta}) + \frac{\lambda}{2}\|\boldsymbol{w}\|^2 \tag{7.1}$$

Gradient update becomes:

$$\boldsymbol{w} \leftarrow (1 - \alpha\lambda)\boldsymbol{w} - \alpha\nabla_{\boldsymbol{w}} L \tag{7.2}$$

The factor $(1 - \alpha\lambda)$ causes "weight decay."

7.1.3 L1 Regularization

Add L1 norm:

$$\tilde{L}(\boldsymbol{\theta}) = L(\boldsymbol{\theta}) + \lambda\|\boldsymbol{w}\|_1 \tag{7.3}$$

L1 regularization:

- Promotes sparsity (many weights become exactly zero)
- Useful for feature selection
- Gradient: $\text{sign}(\boldsymbol{w})$

7.1.4 Elastic Net

Combines L1 and L2:

$$\tilde{L}(\boldsymbol{\theta}) = L(\boldsymbol{\theta}) + \lambda_1\|\boldsymbol{w}\|_1 + \lambda_2\|\boldsymbol{w}\|^2 \tag{7.4}$$

7.2 Dataset Augmentation ☒

Data augmentation artificially increases training set size by applying transformations that preserve labels.

7.2.1 Intuition: Seeing the Same Thing in Many Ways

Humans recognize an object despite different viewpoints, lighting, or small occlusions. Augmentation teaches models the same robustness by showing multiple, label-preserving variations of each example. This reduces overfitting by making spurious correlations less useful and forcing the model to focus on invariant structure.

7.2.2 Image Augmentation

Common transformations:

- **Geometric:** rotation, translation, scaling, flipping, cropping
- **Color:** brightness, contrast, saturation adjustments
- **Noise:** Gaussian noise, blur
- **Cutout/Erasing:** randomly mask regions
- **Mixup:** blend pairs of images and labels

Example: horizontal flip

$$\boldsymbol{x}_{\text{aug}} = \text{flip}(\boldsymbol{x}), \quad y_{\text{aug}} = y \tag{7.5}$$

7.2.3 Text Augmentation

For NLP:

- Synonym replacement
- Random insertion/deletion
- Back-translation
- Paraphrasing

7.2.4 Audio Augmentation

For speech/audio:

- Time stretching
- Pitch shifting
- Adding background noise
- SpecAugment (masking frequency/time regions)

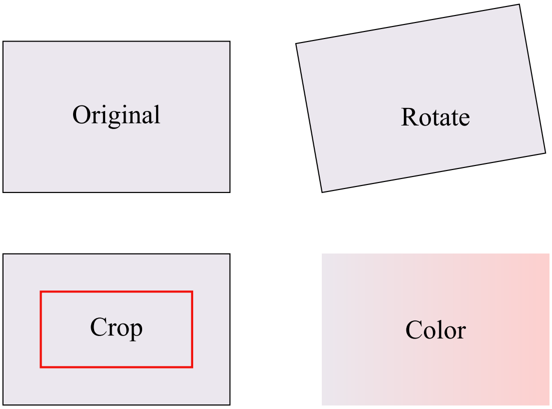

Figure 7.1: Illustration of common image augmentations. Variants preserve labels while encouraging invariances.

7.3 Early Stopping ☒

Early stopping monitors validation performance and stops training when it begins to degrade.

7.3.1 Intuition: Stop Before You Memorize

Imagine studying for an exam. Initially, practice improves your understanding (training and validation improve). If you keep cramming the exact same questions, you start memorizing answers that don't help with new questions (training improves, validation worsens). Early stopping is the principle of stopping at the point of best validation performance to avoid memorization.

7.3.2 Algorithm

1. Train model and evaluate on validation set periodically
2. Track best validation performance
3. If no improvement for p epochs (patience), stop
4. Return model with best validation performance

Algorithm 7.1 (Early stopping meta-algorithm). Let n be the number of steps between evaluations.
Let p be the patience (number of worsened validations before stopping).
Let $\boldsymbol{\theta}_0$ be the initial parameters.

```
1. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}_0, i \leftarrow 0, j \leftarrow 0, v \leftarrow \infty$
2. $\boldsymbol{\theta}^* \leftarrow \boldsymbol{\theta}, i^* \leftarrow i$
3. while $j < p$ do
    (a) Update $\boldsymbol{\theta}$ by running the training algorithm for $n$ steps
    (b) $i \leftarrow i + n$
    (c) $v' \leftarrow \text{ValidationSetError}(\boldsymbol{\theta})$
    (d) if $v' < v$ then
        i. $j \leftarrow 0$
        ii. $\boldsymbol{\theta}^* \leftarrow \boldsymbol{\theta}, i^* \leftarrow i, v \leftarrow v'$
    (e) else $j \leftarrow j + 1$
    (f) end if
4. end while
5. Return best parameters $\boldsymbol{\theta}^*$ and best step $i^*$
```

7.3.3 Benefits

- **Simple and effective:** Requires only tracking validation performance and a patience parameter; widely used in practice [GBC16a; Pri23].
- **Automatically determines training duration:** Finds a good stopping time without a pre-fixed epoch budget, often saving substantial compute.

- **Implicit regularization:** Halting before convergence limits effective capacity by keeping weights smaller and preventing memorization; in some regimes it mimics an L2 constraint under gradient descent [GBC16a].

- **Compatible with many settings:** Works with any loss, architecture (MLPs, CNNs, Transformers), and optimizer.

- **Reduces computational cost:** Training stops as soon as overfitting begins, reducing energy/time.

- **Improves generalization stability:** Curbs validation variance late in training when overfitting spikes.

7.3.4 Considerations

- **Validation protocol:** Requires a reliable validation set and evaluation cadence; noisy metrics may trigger premature stops. Use smoothing or require monotone improvements.

- **Patience and frequency:** Patience p and evaluation interval n interact with LR schedules; too small p can stop before a scheduled LR drop helps.

- **Checkpointing:** Always restore the best model (not the last); keep track of the weights at the best validation step.

- **Warmup and plateaus:** With warmup or long plateaus, consider larger patience or metric smoothing.

- **Multi-metric objectives:** For tasks with multiple metrics (e.g., accuracy and calibration), pick the primary metric or a composite.

- **Distributed training:** Ensure validation statistics are aggregated consistently across devices to avoid spurious decisions.

- **Historical context:** Early stopping predates modern deep learning and was popular in classical neural nets and boosting as a strong regularizer; it remains a standard baseline [GBC16a; Bis06].

Example 7.2. Example (vision): Train a ResNet on CIFAR-10 with validation accuracy checked each epoch; use patience $p = 20$. Accuracy peaks at epoch 142; training halts at 162 without improvement, and the checkpoint from 142 is used for testing.

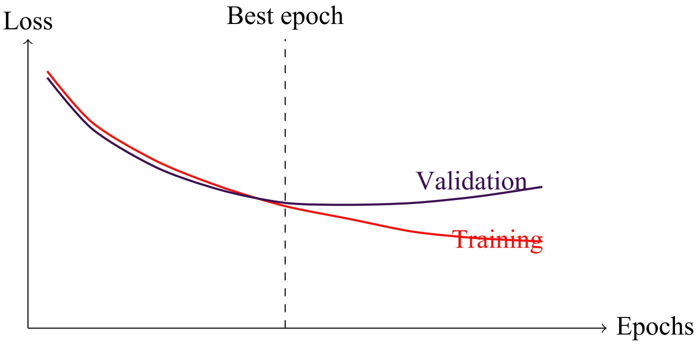

Figure 7.2: Early stopping: validation loss reaches a minimum before training loss; the best checkpoint is saved and restored.

7.4 Dropout ☒

Dropout randomly deactivates neurons during training, preventing co-adaptation.

7.4.1 Intuition: Training a Robust Ensemble

Dropout is like asking different subsets of a team to work on the same task on different days. No single member can rely on a particular colleague always being present, so each learns to be broadly useful. This results in a robust team (model) that performs well even when some members (neurons) are inactive.

7.4.2 Training with Dropout

At each training step, for each layer:

1. Sample binary mask $\boldsymbol{m}$ with $P(m_i = 1) = p$
2. Apply mask: $\boldsymbol{h} = \boldsymbol{m} \odot \boldsymbol{h}$

Mathematically:

$$\boldsymbol{h}_{\text{dropout}} = \boldsymbol{m} \odot f(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) \tag{7.6}$$

where $m_i \sim \text{Bernoulli}(p)$.

7.4.3 Inference

At test time, scale outputs by dropout probability:

$$\boldsymbol{h}_{\text{test}} = p \cdot f(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) \tag{7.7}$$

Or equivalently, scale weights during training by $\frac{1}{p}$ (inverted dropout).
In practice, modern frameworks implement *inverted dropout*: during training, activations are scaled by $\frac{1}{p}$ after masking so that the expected activation matches test-time activations, and no scaling is needed at inference [Sri+14; GBC16a]. For convolutional layers, use the same p per feature map to avoid distribution shift.

7.4.4 Interpretation

Dropout can be viewed as:

- **Implicit ensemble:** Sampling masks trains an ensemble of 2^n subnetworks whose shared weights yield a form of model averaging [Sri+14].

- **Noise injection:** Multiplicative Bernoulli noise on activations acts like data-dependent regularization analogous to adding Gaussian noise for linear models [GBC16a].

- **Approximate Bayesian inference:** With appropriate priors, dropout relates to variational inference; applying dropout at test time with multiple passes (MC dropout) estimates predictive uncertainty [GG16].

Example 7.3. Example (uncertainty): Run $T = 20$ stochastic forward passes with dropout enabled at test time and average predictions to obtain mean and variance; high variance flags low-confidence inputs.

7.4.5 Variants

DropConnect: Drop individual weights instead of activations, promoting sparsity at the parameter level.
Spatial Dropout: Drop entire feature maps in CNNs to preserve spatial coherence and regularize channel reliance.
Variational Dropout: Use the same dropout mask across time steps in RNNs to avoid injecting different noise per step that can harm temporal consistency.

MC Dropout: Keep dropout active at inference and average predictions to quantify epistemic uncertainty [GG16], useful in safety-critical applications.

Concrete/Alpha Dropout: Continuous relaxations or distributions tailored for specific activations (e.g., SELU) to maintain self-normalizing properties.

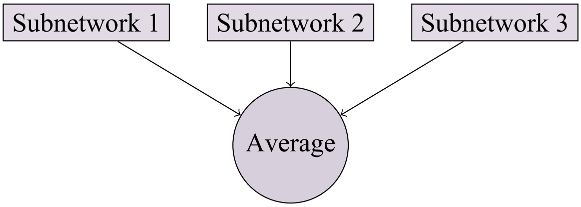

Figure 7.3: Dropout as implicit model averaging over many subnetworks.

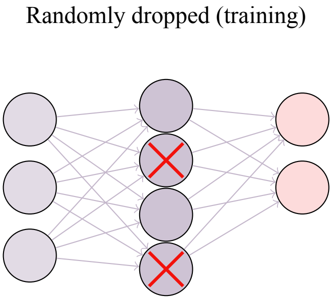

Figure 7.4: Dropout during training: randomly deactivating hidden units encourages redundancy and robustness.

7.5 Batch Normalization ☒

Batch normalization normalizes layer inputs across the batch dimension.

7.5.1 Intuition: Keeping Scales Stable

Training can become unstable if the distribution of activations shifts as earlier layers update (internal covariate shift). Batch normalization re-centers and re-scales activations, keeping them in a predictable range so downstream layers see a more stable input distribution. This allows larger learning rates and speeds up training.

7.5.2 Algorithm

For mini-batch $\mathcal{B}$ with activations $\boldsymbol{x}$:

$$\mu_{\mathcal{B}} = \frac{1}{|\mathcal{B}|}\sum_{i\in\mathcal{B}} x_i \tag{7.8}$$

$$\sigma_{\mathcal{B}}^2 = \frac{1}{|\mathcal{B}|}\sum_{i\in\mathcal{B}} (x_i - \mu_{\mathcal{B}})^2 \tag{7.9}$$

$$\hat{x}_i = \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \tag{7.10}$$

$$y_i = \gamma\hat{x}_i + \beta \tag{7.11}$$

where γ and β are learnable parameters.

Implementation details: maintain running averages $\mu_{\text{running}}, \sigma^2_{\text{running}}$ updated with momentum ρ per iteration; apply per-feature normalization for fully connected layers and per-channel per-spatial-location statistics for CNNs [IS15; GBC16a].

7.5.3 Benefits

- **Stabilizes distributions:** Mitigates internal covariate shift, keeping activations in a stable range [IS15].
- **Enables larger learning rates:** Better-conditioned optimization allows faster training.
- **Less sensitive initialization:** Wider set of workable initializations [GBC16a].
- **Regularization effect:** Mini-batch noise in statistics acts as stochastic regularization, improving generalization.
- **Supports deeper nets:** Facilitates training very deep architectures (e.g., ResNets) [He+16].
- **Improves gradient flow:** Normalized scales yield healthier signal-to-noise ratios in backprop.

7.5.4 Inference

At test time, use running averages computed during training:

$$y = \gamma \frac{x - \mu_{\text{running}}}{\sqrt{\sigma^2_{\text{running}} + \epsilon}} + \beta \tag{7.12}$$

Be careful when batch sizes are small at inference or differ from training: do not recompute batch statistics at test time; use the stored running averages. For distribution shift, consider recalibrating $\mu_{\text{running}}, \sigma^2_{\text{running}}$ with a small unlabeled buffer.

7.5.5 Variants

Layer Normalization: Normalize across features for each sample; effective in RNNs and Transformers where batch statistics are less stable.

Group Normalization: Normalize within groups of channels; robust to small batch sizes common in detection/segmentation.

Instance Normalization: Normalize each sample independently; prominent in style transfer where contrast/style per instance varies.

Batch Renormalization / Ghost BatchNorm: Adjust for mismatch between batch and population statistics or simulate small batches inside large ones for regularization.

Example 7.4. Example (vision): In a CNN with batch size 128, use per-channel BN after each convolution with momentum $\rho = 0.9$ and $\epsilon = 10^{-5}$. During inference, freeze γ, β and use stored running statistics.

7.6 Other Regularization Techniques ☒

7.6.1 Intuition: Many Small Guards Against Overfitting

Beyond penalties and normalization, there are practical techniques that act like small guards during training. Each introduces a mild constraint or noise that nudges the model away from brittle solutions and encourages smoother decision boundaries.

7.6.2 Label Smoothing

Replace hard targets with smoothed distributions:

$$y'_k = (1 - \epsilon) y_k + \frac{\epsilon}{K} \tag{7.13}$$

Prevents overconfident predictions and improves calibration by discouraging saturated logits; commonly used in large-scale classification (e.g., ImageNet) and sequence models. Choose ϵ in $[0.05, 0.2]$ depending on class count K and desired calibration. It can also mitigate overfitting to annotator noise.

Example 7.5. Example (ImageNet): With $K = 1000$ and $\epsilon = 0.1$, the target for the correct class becomes 0.9 while others receive 0.0001 each; top-1 accuracy and ECE often improve.

7.6.3 Gradient Clipping

Limit gradient magnitude to prevent exploding gradients:
Clipping by value:

$$g \leftarrow \max(\min(g, \theta), -\theta) \tag{7.14}$$

Clipping by norm: Clipping stabilizes training in RNNs and very deep nets by preventing exploding gradients, especially with large learning rates or noisy batches. Norm clipping with threshold θ is preferred as it preserves direction while scaling magnitude. Excessive clipping can bias updates and slow convergence; tune θ w.r.t. optimizer and batch size.

Example 7.6. Example (NLP): Train a GRU with global norm clip $\theta = 1.0$; without clipping, gradients occasionally explode causing loss spikes.

$$g \leftarrow \frac{g}{\max(1, \|g\|/\theta)} \tag{7.15}$$

7.6.4 Stochastic Depth

Randomly skip residual blocks during training with survival probability p_l per layer l, while using the full network depth at test time. This shortens expected depth during training, improving gradient flow and reducing overfitting in very deep networks [Hua+16].

Let p_l decrease with depth (e.g., linearly from 1.0 to p_{min}). During training, with probability $1 - p_l$ a residual block is bypassed; otherwise it is applied and its output is scaled to match test-time expectation.

Example 7.7. Example (ResNets): In a 110-layer ResNet, set p_l from 1.0 to 0.8 across depth; training converges faster and generalizes better on CIFAR-10.

7.6.5 Mixup

Train on convex combinations of examples:

$$\tilde{\boldsymbol{x}} = \lambda \boldsymbol{x}_i + (1 - \lambda)\boldsymbol{x}_j \tag{7.16}$$

$$\tilde{y} = \lambda y_i + (1 - \lambda) y_j \tag{7.17}$$

where $\lambda \sim \text{Beta}(\alpha, \alpha)$.

Mixup encourages linear behavior between classes, reduces memorization of spurious correlations, and improves robustness to label noise [Zha+18]. Typical α values are in $[0.2, 1.0]$. Variants include CutMix (patch-level mixing) [Yun+19] and Manifold Mixup (mix at hidden layers).

Example 7.8. Example (vision): With $\alpha = 0.4$, randomly pair images in a batch and form convex combinations; train using the mixed targets. Improves top-1 accuracy and calibration.

7.6.6 Adversarial Training

Add adversarially perturbed examples to training:

$$\boldsymbol{x}_{\text{adv}} = \boldsymbol{x} + \epsilon \cdot \text{sign}(\nabla_{\boldsymbol{x}} L(\boldsymbol{x}, y)) \tag{7.18}$$

This FGSM objective can be extended to multi-step PGD adversaries. Adversarial training improves worst-case robustness but often reduces clean accuracy and increases compute [Goo+14]. Robust features learned can transfer across tasks; careful tuning of ϵ, steps, and randomness is crucial.

Example 7.9. Example (robust CIFAR-10): Use $\epsilon = 8/255$, $k = 7$ PGD steps, step size $2/255$; train with a 1:1 mix of clean and adversarial samples.

Historical context and applications Label smoothing and dropout popularized regularization at scale; gradient clipping stabilized early RNNs; stochastic depth enabled training very deep residual networks; mixup and CutMix improved data-efficient generalization; adversarial training established the modern paradigm for robustness. Applications span medical imaging, autonomous driving, speech recognition, and large-scale language models where calibration and robustness are critical [GBC16a; He+16; IS15].

7.7 Real World Applications

Regularization techniques are essential for making deep learning models work reliably in real-world scenarios where data is messy and models need to perform well on new, unseen examples.

7.7.1 Autonomous Vehicle Safety

Self-driving cars rely heavily on regularization to ensure safe operation:

- **Robust object detection:** Regularization techniques like dropout and data augmentation help vehicles recognize pedestrians, cyclists, and other vehicles under diverse conditions (rain, fog, night driving, unusual angles). Without regularization, the system might fail when encountering weather or lighting conditions not heavily represented in training data.

- **Generalization to new environments:** A self-driving car trained in sunny California needs to work safely in snowy Boston. Regularization prevents the model from memorizing specific training locations and instead learns general driving principles that transfer across different cities and climates.

- **Preventing overfitting to rare events:** Regularization helps models maintain good performance on common scenarios (normal traffic) while still being prepared for rare but critical situations (emergency vehicles, unexpected obstacles).

7.7.2 Medical Imaging Analysis

Healthcare applications use regularization to make reliable diagnoses:

- **Cancer detection from limited data:** Medical datasets are often small because annotating medical images requires expert radiologists. Regularization techniques (especially data augmentation and early stopping) allow models to learn effectively from hundreds rather than millions of examples, making them practical for clinical use.

- **Consistent performance across hospitals:** Different hospitals use different imaging equipment. Regularization ensures models trained at one hospital generalize to work at others, despite variations in image quality, resolution, or equipment manufacturers.

- **Reducing false positives:** In medical diagnosis, false alarms cause unnecessary anxiety and costly follow-up tests. Regularization like label smoothing helps models be appropriately confident, reducing overconfident but incorrect predictions.

7.7.3 Natural Language Processing for Customer Service

Chatbots and virtual assistants benefit from regularization:

- **Handling diverse customer queries:** Customers phrase questions in countless ways. Regularization through data augmentation (paraphrasing, synonym replacement) helps chatbots understand intent even when people use unexpected wording.

- **Preventing memorization of training conversations:** Without regularization, chatbots might memorize training examples and give nonsensical responses to new queries. Dropout and other techniques force the model to learn general conversation patterns rather than specific exchanges.

- **Adapting to evolving language:** Language changes constantly with new slang and terminology. Regularization helps models stay flexible and adapt to linguistic shifts without extensive retraining.

7.7.4 Key Benefits in Practice

Regularization provides crucial advantages in real applications:

- **Works with limited data:** Not every problem has millions of training examples

- **Reduces maintenance costs:** Models generalize better, requiring less frequent retraining
- **Increases reliability:** Systems work consistently even when deployed conditions differ from training
- **Enables deployment:** Makes models trustworthy enough for safety-critical applications

These examples show that regularization isn't just a mathematical nicety—it's the difference between models that work only in labs and those that succeed in the real world.

7.8 Problems ☒

This section provides exercises to reinforce your understanding of regularization techniques. Problems are categorized by difficulty and include hints.

7.8.1 Easy Problems (6 problems)

Problem 7.10 (Identify Regularization)**.** List whether each technique primarily reduces variance, bias, or both: L2, L1, dropout, data augmentation, early stopping, batch normalization.
Hint: Consider effect on model capacity and training dynamics.

Problem 7.11 (Weight Decay Update)**.** Given learning rate α and L2 coefficient λ, write the update rule for weights with gradient g.
Hint: Combine gradient step with multiplicative shrinkage.

Problem 7.12 (L1 vs L2)**.** Explain when L1 is preferred over L2, and vice versa.
Hint: Sparsity, feature selection, stability, and correlated features.

Problem 7.13 (Early Stopping Curve)**.** Sketch training and validation loss over epochs showing overfitting and stopping point.
Hint: Validation loss reaches minimum before training loss.

Problem 7.14 (Augmentation Invariance)**.** For image classification, list three invariances augmentation can teach and one risk.
Hint: Rotation/translation invariance vs. label leakage or distribution shift.

Problem 7.15 (BatchNorm Inference)**.** Explain why running averages are used at test time for batch normalization.
Hint: Mini-batch statistics are noisy and unavailable at inference.

7.8.2 Medium Problems (5 problems)

Problem 7.16 (Elastic Net Penalty)**.** Derive the gradient of $\lambda_1 \|\boldsymbol{w}\|_1 + \frac{\lambda_2}{2}\|\boldsymbol{w}\|_2^2$ with respect to $\boldsymbol{w}$.
Hint: Subgradient for L1; standard gradient for L2.

Problem 7.17 (Dropout Scaling)**.** Show equivalence between scaling activations at test time by p and scaling at training by $1/p$ (inverted dropout).
Hint: Match expected activations across train/test.

Problem 7.18 (Early Stopping as Regularization)**.** Argue how early stopping can mimic an L2 constraint under gradient descent.
Hint: Consider that weights remain small when training halts early.

Problem 7.19 (Label Smoothing and Confidence)**.** For $\epsilon > 0$, show how label smoothing affects cross-entropy gradients.
Hint: Replace one-hot y by $(1 - \epsilon)\, y + \epsilon/K$.

Problem 7.20 (Mixup Geometry)**.** Explain how mixup encourages linear behavior between classes in representation space.
Hint: Consider convex combinations and linear decision boundaries.

7.8.3 Hard Problems (5 problems)

Problem 7.21 (Generalization Bound Intuition)**.** Discuss how norm constraints relate to capacity control (e.g., Rademacher complexity) and generalization.
Hint: Smaller norms can reduce hypothesis class complexity.

Problem 7.22 (Adversarial Training Trade-offs)**.** Analyze how adversarial training affects robustness, clean accuracy, and optimization.
Hint: Robust features vs. gradient masking and compute cost.

Problem 7.23 (BatchNorm and Optimization)**.** Provide a theoretical or empirical argument for why batch normalization enables higher learning rates.
Hint: Consider conditioning of the optimization problem.

Problem 7.24 (Stochastic Depth in Deep Nets)**.** Model the expected depth under stochastic depth and discuss gradient flow implications.
Hint: Consider per-layer survival probabilities.

Problem 7.25 (Design a Regularization Suite)**.** Given a 100-class image dataset with 50k images, design a regularization suite (penalties, augmentation, normalization, schedules). Justify choices.
Hint: Balance data, model size, compute budget, and desired robustness.

Key Takeaways

Key Takeaways 7

- **Regularisation** constrains model capacity to improve generalisation and prevent overfitting to training data.
- **Norm penalties** (L1, L2) encourage simpler models; L1 induces sparsity whilst L2 shrinks weights uniformly.
- **Data augmentation** artificially expands training sets by applying semantics-preserving transformations.
- **Dropout** randomly drops units during training, forcing redundant representations and reducing co-adaptation.
- **Early stopping and batch normalisation** are simple yet powerful techniques for better training dynamics and generalisation.

Problems

Easy

Problem 7.26 (L1 vs L2 Regularisation)**.** Explain the difference between L1 and L2 regularisation. Which one is more likely to produce sparse weights, and why?
Hint: Consider the shape of the L1 and L2 penalty terms and their gradients.

Problem 7.27 (Data Augmentation Strategies)**.** List three common data augmentation techniques for image classification tasks and explain how each helps improve

generalisation.
Hint: Think about geometric transformations, colour adjustments, and realistic variations.

Problem 7.28 (Early Stopping)**.** Describe how early stopping works as a regularisation technique. What metric should you monitor, and when should you stop training?
Hint: Consider validation set performance and the risk of overfitting to the training set.

Problem 7.29 (Dropout Interpretation)**.** During training, dropout randomly sets activations to zero with probability p. During inference, all neurons are active but their outputs are scaled. Explain why this scaling is necessary.
Hint: Think about the expected value of activations during training versus inference.

Medium

Problem 7.30 (Regularisation Trade-off)**.** Given a model with both L2 regularisation and dropout, discuss how you would tune the regularisation strength λ and dropout rate p. What signs would indicate too much or too little regularisation?
Hint: Monitor training and validation loss curves, and consider the bias-variance trade-off.

Problem 7.31 (Batch Normalisation Effect)**.** Explain how batch normalisation acts as a regulariser. Discuss its interaction with dropout.
Hint: Consider the noise introduced by computing statistics on mini-batches and why dropout is often not needed with batch normalisation.

Hard

Problem 7.32 (Mixup Derivation)**.** Mixup trains on convex combinations of examples: $\tilde{\boldsymbol{x}} = \lambda \boldsymbol{x}_i + (1 - \lambda)\boldsymbol{x}_j$ where $\lambda \sim \text{Beta}(\alpha, \alpha)$. Derive how this affects the decision boundary and explain why it improves generalisation.
Hint: Consider the effect on the loss surface and the implicit regularisation from interpolating between examples.

Problem 7.33 (Adversarial Training)**.** Design an adversarial training procedure for a classification model. Explain how to generate adversarial examples using FGSM (Fast Gradient Sign Method) and why this improves robustness.

Hint: Adversarial examples are $\boldsymbol{x}_{adv} = \boldsymbol{x} + \epsilon \cdot \text{sign}(\nabla_{\boldsymbol{x}} L)$. Discuss the trade-off between clean and adversarial accuracy.

Chapter 8

Optimization for Training Deep Models

This chapter covers optimization algorithms and strategies for training deep neural networks effectively. Modern optimizers go beyond basic gradient descent to accelerate convergence and improve performance.

Learning Objectives

After completing this chapter, you will be able to:

1. **Compare gradient descent variants** (batch, stochastic, mini-batch) and choose appropriate batch sizes.
2. **Explain and implement momentum-based methods** including classical momentum and Nesterov accelerated gradient.
3. **Apply adaptive optimizers** (AdaGrad, RMSProp, Adam) and tune key hyperparameters and schedules.
4. **Describe second-order approaches** (Newton, quasi-Newton/L-BFGS, natural gradient) and when they are practical.
5. **Diagnose optimization challenges** (vanishing/exploding gradients, saddle points, plateaus) and apply remedies.

6. **Design an optimization plan** combining initializer choice, optimizer, learning rate schedule, and gradient clipping.

8.1 Gradient Descent Variants ☒

8.1.1 Intuition: Following the Steepest Downhill Direction

Imagine standing on a foggy hillside trying to reach the valley. You feel the slope under your feet and take a step downhill. **Batch gradient descent** measures the average slope using the whole landscape (dataset) before each step: accurate but slow to gauge. **Stochastic gradient descent (SGD)** feels the slope at a single point: fast, but noisy. **Mini-batch** is a compromise: feel the slope at a handful of nearby points to get a reliable yet efficient direction. This trade-off underlies most practical training regimes.mini-batch

Historical note: Early neural network training widely used batch gradient descent, but the rise of large datasets and GPUs made mini-batch SGD the de facto standard [GBC16a; Pri23].

8.1.2 Batch Gradient Descent

Computes gradient using entire training set:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \nabla_{\boldsymbol{\theta}} \frac{1}{n} \sum_{i=1}^{n} L(\boldsymbol{\theta}, \boldsymbol{x}^{(i)}, y^{(i)}) \tag{8.1}$$

Characteristics:

- Deterministic and low-variance updates; well-suited for convex problems.
- Each step uses all n examples, incurring high computation and memory costs for large datasets.
- In deep, non-convex landscapes, stable but often too slow to respond to curvature changes.

Example (convex quadratic): Let $L(\theta) = \frac{1}{2}a\theta^2$ with gradient $a\theta$. Batch GD with step size α yields $\theta_{t+1} = (1 - \alpha a)\theta_t$. Convergence occurs if $0 < \alpha < \frac{2}{a}$, illustrating the learning-rate – curvature interaction.

When to use:

- Small datasets that fit comfortably in memory.
- Convex or nearly convex objectives (e.g., linear/logistic regression) when precise convergence is desired.

Historical context: Full-batch methods trace back to classical numerical optimization. For massive datasets, stochastic approximations dating to [RM51] became essential; modern deep learning typically favors mini-batches [GBC16a; GBC16b; Zha+24b].

8.1.3 Stochastic Gradient Descent (SGD)

Uses a single random example per update:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}, \boldsymbol{x}^{(i)}, y^{(i)}) \tag{8.2}$$

Characteristics:

- Very fast, streaming-friendly updates; one pass can begin learning immediately.
- Noisy gradients add exploration, helping traverse saddle points and plateaus.
- High variance can hamper precise convergence; diminishing α_t helps stabilize [RM51].

Practical tips:

- Use a decaying schedule (e.g., $\alpha_t = \alpha_0/(1 + \lambda t)$) or switch to momentum/Adam later for fine-tuning [GBC16b; Zha+24b].
- Shuffle examples every epoch to avoid periodic bias.

Applications: Early CNN training and many online/streaming scenarios employ SGD due to its simplicity and ability to handle large-scale data. In large vision models, SGD with momentum remains competitive [He+16].

8.1.4 Mini-Batch Gradient Descent

Balances batch and stochastic approaches:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \nabla_{\boldsymbol{\theta}} \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} L(\boldsymbol{\theta}, \boldsymbol{x}^{(i)}, y^{(i)}) \tag{8.3}$$

where $\mathcal{B}$ is a mini-batch (typically 32 – 1024 examples depending on model and hardware).
Benefits:

- Reduces gradient variance
- Efficient GPU utilization
- Good balance of speed and stability

Further guidance:

- Larger batches yield smoother estimates but may require proportionally larger learning rates; the “linear scaling rule” is a useful heuristic in some regimes.
- Very large batches can hurt generalization unless paired with warmup and appropriate regularization. See [GBC16b; Zha+24b].

Illustrative example (variance vs. batch size): For a fixed α, increasing $|\mathcal{B}|$ reduces the variance of the stochastic gradient approximately as $\mathcal{O}(1/|\mathcal{B}|)$, improving stability but diminishing returns once hardware is saturated.

8.2 Momentum-Based Methods ☒

8.2.1 Intuition: Rolling a Ball Down a Valley

Plain SGD can wobble like a marble on a bumpy path. **Momentum** acts like mass: it carries velocity so you keep moving in consistently good directions and smooth out small bumps. **Nesterov acceleration** adds anticipation by peeking where the momentum will take you before correcting, often yielding crisper steps in curved valleys.
Historical note: Momentum has deep roots in convex optimization and was popularized in early neural network training; Nesterov’s variant provided stronger theoretical guarantees in convex settings and inspired practical variants in deep learning [Pol64; Nes83; GBC16a; Bis06].

8.2.2 Momentum

Accumulates gradients over time:

$$\boldsymbol{v}_t = \beta \boldsymbol{v}_{t-1} - \alpha \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_t) \tag{8.4}$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \boldsymbol{v}_t \tag{8.5}$$

where $\beta \in [0, 1)$ is the momentum coefficient (typically 0.9).
Benefits:

- Accelerates convergence in relevant directions
- Dampens oscillations
- Helps escape local minima and saddle points

Interpretation: Momentum is equivalent to an exponentially weighted moving average of past gradients, implementing a low-pass filter that suppresses high-frequency noise. In anisotropic valleys, it allows larger effective step along the shallow curvature direction while reducing zig-zag across the sharp direction [Pol64; GBC16b; Zha+24b].
Choice of hyperparameters:

- $\beta \in [0.8, 0.99]$; larger values increase smoothing and memory.
- Couple with a tuned α; too-large α can still diverge.

Example (ravine function): For $L(\theta_1, \theta_2) = 100\theta_1^2 + \theta_2^2$, momentum reduces oscillations in the steep θ_1 direction and speeds travel along the gentle θ_2 direction.

8.2.3 Nesterov Accelerated Gradient (NAG)

”Look-ahead” version of momentum:

$$\boldsymbol{v}_t = \beta \boldsymbol{v}_{t-1} - \alpha \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_t + \beta \boldsymbol{v}_{t-1}) \tag{8.6}$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \boldsymbol{v}_t \tag{8.7}$$

Evaluates gradient at anticipated future position, often providing better updates.
Rationale: By computing the gradient at the look-ahead point $\theta_t + \beta v_{t-1}$, NAG corrects the course earlier, which reduces overshoot in curved valleys and can improve convergence rates in convex settings [Nes83; GBC16b; GBC16a].

Practice notes:

- Common defaults: $\beta = 0.9$, initial $\alpha \in [10^{-3}, 10^{-1}]$ depending on scale.
- Widely used with SGD in large-scale vision models [He+16].

8.3 Adaptive Learning Rate Methods ☒

8.3.1 Intuition: Per-Parameter Step Sizes

Different parameters learn at different speeds: some directions are steep, others are flat. **Adaptive methods** adjust the step size per parameter based on recent gradient information, allowing faster progress on rarely-updated or low-variance dimensions while stabilizing steps on highly-volatile ones.
Historical note: AdaGrad emerged for sparse problems; RMSProp stabilized AdaGrad's decay; Adam blended momentum with RMSProp-style adaptation and became a widely used default in deep learning [DHS11; TH12; KB14; GBC16a].

8.3.2 AdaGrad

Adapts learning rate per parameter based on historical gradients:

$$\boldsymbol{g}_t = \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_t) \tag{8.8}$$

$$\boldsymbol{r}_t = \boldsymbol{r}_{t-1} + \boldsymbol{g}_t \odot \boldsymbol{g}_t \tag{8.9}$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \frac{\alpha}{\sqrt{\boldsymbol{r}_t + \epsilon}} \odot \boldsymbol{g}_t \tag{8.10}$$

where ϵ (e.g., 10^{-8}) prevents division by zero. AdaGrad is well-suited to sparse features: infrequent parameters receive larger effective steps, accelerating learning in NLP and recommender settings [DHS11; GBC16b; Zha+24b]. A drawback is the ever-growing accumulator $\boldsymbol{r}_t$, which can shrink steps too aggressively over long runs.

8.3.3 RMSProp

Addresses AdaGrad's aggressive decay using exponential moving average:

$$\boldsymbol{r}_t = \rho \boldsymbol{r}_{t-1} + (1 - \rho) \boldsymbol{g}_t \odot \boldsymbol{g}_t \tag{8.11}$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \frac{\alpha}{\sqrt{\boldsymbol{r}_t + \epsilon}} \odot \boldsymbol{g}_t \tag{8.12}$$

Add a small ϵ for numerical stability and tune decay $\rho \in [0.9, 0.99]$. RMSProp prevents the learning rate from decaying to zero as in AdaGrad, making it effective for non-stationary objectives typical in deep networks [TH12; GBC16b; Zha+24b].

8.3.4 Adam (Adaptive Moment Estimation)

Combines momentum and adaptive learning rates:

$$\boldsymbol{m}_t = \beta_1 \boldsymbol{m}_{t-1} + (1 - \beta_1)\boldsymbol{g}_t \tag{8.13}$$

$$\boldsymbol{v}_t = \beta_2 \boldsymbol{v}_{t-1} + (1 - \beta_2)\boldsymbol{g}_t \odot \boldsymbol{g}_t \tag{8.14}$$

$$\hat{\boldsymbol{m}}_t = \frac{\boldsymbol{m}_t}{1 - \beta_1^t} \tag{8.15}$$

$$\hat{\boldsymbol{v}}_t = \frac{\boldsymbol{v}_t}{1 - \beta_2^t} \tag{8.16}$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \frac{\alpha \hat{\boldsymbol{m}}_t}{\sqrt{\hat{\boldsymbol{v}}_t} + \epsilon} \tag{8.17}$$

Default hyperparameters: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$, $\alpha = 0.001$ [KB14]. Adam often converges quickly and is robust to poorly scaled gradients. For best generalization in some vision tasks, SGD with momentum can still outperform Adam; consider switching optimizers during fine-tuning [GBC16a; Zha+24b; He+16].

8.3.5 Learning Rate Schedules

Step Decay:

$$\alpha_t = \alpha_0 \cdot \gamma^{\lfloor t/s \rfloor} \tag{8.18}$$

Exponential Decay:

$$\alpha_t = \alpha_0 e^{-\lambda t} \tag{8.19}$$

Cosine Annealing:

$$\alpha_t = \alpha_{\text{min}} + \frac{1}{2}(\alpha_{\text{max}} - \alpha_{\text{min}}) \left(1 + \cos\left(\frac{t}{T}\pi\right)\right) \tag{8.20}$$

Other useful schedules:

- Warmup: start from a small α and increase linearly over the first T_w steps to reduce early instabilities in large-batch training.

- One-cycle policy: increase then anneal α, often paired with momentum decay, to speed convergence and improve generalization.

Practical recipe: Combine cosine decay with warmup for transformer-like models, step decay for CNNs trained with SGD, and exponential decay for simple baselines [GBC16b; Zha+24b].

8.4 Second-Order Methods ☒

8.4.1 Intuition: Curvature-Aware Steps

First-order methods follow the slope; second-order methods also consider the *curvature* of the landscape to choose better-scaled steps. If the valley is sharply curved in one direction and flat in another, curvature-aware steps shorten strides along the sharp direction and lengthen them along the flat one.
Historical note: Classical optimization popularized Newton and quasi-Newton methods; in deep learning, memory and compute constraints motivated approximations like L-BFGS and natural gradient [LN89; Ama98; GBC16a; Bis06].

8.4.2 Newton's Method

Uses second-order Taylor expansion:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \boldsymbol{H}^{-1}\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_t) \tag{8.21}$$

where $\boldsymbol{H}$ is the Hessian matrix. Intuitively, the Hessian rescales the gradient by local curvature, yielding steps invariant to axis scaling in quadratic bowls (see Figure 8.1).

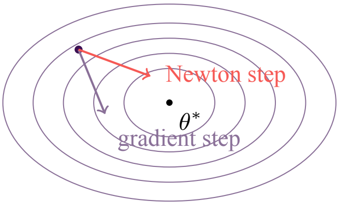

Figure 8.1: Newton's method rescales the gradient by curvature, taking longer steps along shallow directions and shorter steps along steep directions.

Challenges:

- Computing Hessian is $O(n^2)$ in parameters
- Inverting Hessian is $O(n^3)$
- Infeasible for large neural networks

8.4.3 Quasi-Newton Methods

Approximate the Hessian inverse:
L-BFGS: maintains low-rank approximation of Hessian inverse

- More efficient than full Newton's method
- Still expensive for very large models
- Used for smaller networks or specific applications

Historical note: Quasi-Newton methods, notably BFGS and its limited-memory variant L-BFGS [LN89], performed well on moderate-sized networks and remain valuable for fine-tuning smaller models or optimizing differentiable components inside larger systems [GBC16a; Bis06].

8.4.4 Natural Gradient

Uses Fisher information matrix instead of Hessian:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \boldsymbol{F}^{-1} \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_t) \tag{8.22}$$

Provides parameter updates invariant to reparameterization. In probabilistic models, $\boldsymbol{F}$ is the Fisher information, defining a Riemannian metric on the parameter manifold; stepping along $\boldsymbol{F}^{-1}\nabla L$ follows the steepest descent in information geometry [Ama98]. Approximations (e.g., K-FAC) make natural gradient practical in deep nets by exploiting layer structure (see Figure 8.2).

8.5 Optimization Challenges ☒

8.5.1 Intuition: Why Training Gets Stuck

Deep networks combine nonlinearity and depth, creating landscapes with flat plateaus, narrow valleys, and saddle points. Noise (SGD), momentum, and schedules act like navigational aids to keep moving and avoid getting stuck.

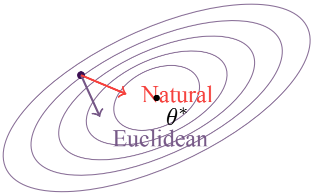

Figure 8.2: Natural gradient accounts for the local geometry (Fisher metric), often directing updates more orthogonally to level sets than Euclidean gradient.

8.5.2 Vanishing and Exploding Gradients

In deep networks, gradients can become exponentially small or large. See vanishing gradient problem and exploding gradient problem.

Vanishing gradients:

- Common with sigmoid/tanh activations
- Mitigated by ReLU, batch normalization, residual connections

Exploding gradients:

- Common in RNNs
- Mitigated by gradient clipping, careful initialization

Gradient clipping:

$$\boldsymbol{g} \leftarrow \frac{\boldsymbol{g}}{\max(1, \|\boldsymbol{g}\|/\theta)} \tag{8.23}$$

8.5.3 Local Minima and Saddle Points

In high dimensions, saddle points are more common than local minima.
Saddle points have:

- Zero gradient
- Mixed curvature (positive and negative eigenvalues)

Momentum and noise help escape saddle points.

8.5.4 Plateaus

Flat regions with small gradients slow convergence. Adaptive methods and learning rate schedules help navigate plateaus.

8.5.5 Practical Optimization Strategy

Recommended approach:

1. Start with Adam [KB14] for rapid progress; tune α in $\{10^{-3}, 3 \cdot 10^{-4}, 10^{-4}\}$.

2. Use cosine decay with warmup for transformer-like models; step decay for CNNs with SGD+momentum [GBC16b; Zha+24b; He+16].

3. If validation accuracy saturates, consider switching from Adam to SGD+Nesterov with tuned α and β to improve generalization.

4. Apply gradient clipping in recurrent models and when training becomes unstable.

5. Monitor training/validation loss, accuracy, and learning-rate schedule. Use early stopping when needed.

Applications and heuristics:

- Vision: SGD+momentum or Nesterov often yields state-of-the-art with careful schedules and augmentations [He+16].

- NLP/Transformers: Adam/AdamW with warmup+cosine is a strong default; clip global norm in seq2seq models.

- Reinforcement learning: Adam with small α stabilizes non-stationary objectives.

Common failure modes:

- Divergence at start: reduce α, add warmup, or increase ϵ for Adam.

- Plateau: try larger batch with warmup, use cosine schedule, or add momentum.

- Overfitting: increase regularization (weight decay, dropout), add data augmentation.

8.6 Key Takeaways ☒

Key Takeaways 8

- **Mini-batch SGD is the workhorse:** balances speed and stability; pair with momentum and schedules.
- **Momentum and Nesterov reduce oscillations:** accelerate along consistent directions and dampen noise.
- **Adaptive methods ease tuning:** Adam often works out-of-the-box; consider switching to SGD+momentum late for best generalisation.
- **Curvature matters:** second-order ideas inspire preconditioning; full Hessians are usually impractical in deep nets.
- **Schedules are critical:** cosine or step decay often yield large gains; warmup stabilises early training.
- **Mitigate pathologies:** use proper initialisation, normalisation, and gradient clipping to handle vanishing/exploding gradients and plateaus.

8.7 Real World Applications

Optimization techniques are the engine that makes deep learning work in practice. Without effective optimization, even the best model architectures would fail to learn. Here's how optimization impacts real-world systems.

8.7.1 Training Large Language Models

Modern AI assistants rely on advanced optimization for their capabilities:

- **Training ChatGPT and similar models:** These models contain billions of parameters and require months of training on thousands of GPUs. Advanced optimizers like Adam with learning rate schedules make this computationally feasible. Without proper optimization, training would take years or fail to converge entirely.

- **Managing computational costs:** Training large models costs millions of dollars in electricity and computing time. Efficient optimization techniques like gradient accumulation and mixed-precision training reduce these costs by 50% or more, making advanced AI accessible to more organizations.
- **Fine-tuning for specific tasks:** When adapting a pre-trained model for specific applications (like legal document analysis or medical question answering), careful optimization allows effective adaptation with just hours of training instead of starting from scratch.

8.7.2 Real-Time Computer Vision

Vision systems in phones, security cameras, and robots depend on optimization:

- **Smartphone facial recognition:** Your phone unlocks in a fraction of a second by running an optimized neural network. The model was trained using techniques like momentum and learning rate decay to achieve high accuracy with a small enough network to run on battery-powered devices.
- **Video surveillance systems:** Security systems process multiple camera feeds simultaneously, detecting unusual activities in real-time. Optimization techniques enable training models that are both accurate and fast enough to analyze video streams without delays.
- **Augmented reality applications:** AR effects on social media apps track faces and objects in real-time at 60 frames per second. This requires models optimized for both accuracy and speed, achieved through careful training with modern optimization methods.

8.7.3 Recommendation Systems at Scale

Online platforms serve personalized content to billions of users:

- **YouTube video recommendations:** YouTube's recommendation system processes viewing patterns from billions of users and millions of videos. Advanced optimization techniques like distributed training across data centers make it possible to update these models daily, ensuring recommendations stay fresh and relevant.

- **News feed personalization:** Social media platforms continuously optimize models to show you content you'll find engaging. The system needs to learn quickly from your recent interactions while balancing exploration (showing new content) and exploitation (showing what you've liked before).
- **E-commerce product matching:** Online retailers optimize models to match products with potential buyers. The optimization must balance multiple objectives: conversion rates, customer satisfaction, inventory levels, and profit margins, all while training on constantly changing data.

8.7.4 Why Optimization Matters

The impact of good optimization in real applications:

- **Speed:** Faster training means quicker iteration and deployment
- **Cost:** Efficient optimization reduces computational expenses dramatically
- **Quality:** Better optimization leads to more accurate models
- **Feasibility:** Some applications only become possible with advanced optimization

Optimization is where theory meets practice—the techniques in this chapter determine whether ambitious deep learning projects succeed or fail in the real world.

8.8 Problems ☒

This section provides exercises to reinforce your understanding of optimization. Problems are categorized by difficulty and include hints.

8.8.1 Easy Problems (6 problems)

Problem 8.1 (Batch vs. Mini-batch)**.** List two pros and two cons of batch GD vs. mini-batch SGD for large-scale training.
Hint: Consider gradient variance, compute efficiency, and convergence behavior.

Problem 8.2 (Learning Rate Intuition)**.** Describe qualitatively what happens if the learning rate is too small or too large.
Hint: Look for slow progress vs. divergence/oscillation.

Problem 8.3 (Momentum Update)**.** Write the momentum update for parameters and explain the role of β.

Hint: Velocity accumulates an EMA of past gradients.

Problem 8.4 (AdaGrad Scaling)**.** Explain why AdaGrad reduces step sizes over time and when this helps.

Hint: Accumulated squared gradients grow; sparse features benefit.

Problem 8.5 (Adam Defaults)**.** State Adam's common default hyperparameters and what each controls.

Hint: $\beta_1, \beta_2, \epsilon, \alpha$.

Problem 8.6 (Gradient Clipping)**.** Give a reason to clip gradients and one potential downside.

Hint: Stabilization vs. biasing the update direction.

8.8.2 Medium Problems (5 problems)

Problem 8.7 (Nesterov Look-ahead)**.** Derive the Nesterov update by evaluating the gradient at the look-ahead point and compare to classical momentum.

Hint: Replace $\nabla L(\theta_t)$ by $\nabla L(\theta_t + \beta v_{t-1})$.

Problem 8.8 (RMSProp vs. AdaGrad)**.** Show how RMSProp's EMA avoids AdaGrad's overly aggressive decay and discuss hyperparameter ρ.

Hint: Replace cumulative sum with exponential moving average.

Problem 8.9 (Adam Bias Correction)**.** Derive the bias-corrected moments $\hat{m}_t, \hat{v}_t$ and explain why correction matters early in training.

Hint: Compute $\mathbb{E}[m_t]$ under zero initialization.

Problem 8.10 (Plateaus and Schedules)**.** Explain why cosine annealing or step decay can help escape plateaus compared to a fixed learning rate.

Hint: Larger steps at the right time.

Problem 8.11 (L-BFGS Memory)**.** Describe how L-BFGS maintains a low-rank inverse Hessian approximation using recent curvature pairs.

Hint: Limited history of (s_t, y_t) updates.

8.8.3 Hard Problems (5 problems)

Problem 8.12 (Natural Gradient Rationale)**.** Argue why the Fisher information provides a geometry suited for probabilistic models and yields reparameterization-invariant updates.
Hint: Consider KL divergence as the local distance measure.

Problem 8.13 (Saddle Points in High Dimensions)**.** Provide an argument for why saddle points are more prevalent than local minima in high-dimensional non-convex problems.
Hint: Random matrix spectra and sign patterns of eigenvalues.

Problem 8.14 (Warmup Heuristics)**.** Justify learning rate warmup in the presence of normalization layers and large batch sizes.
Hint: Stabilize early updates before full-strength steps.

Problem 8.15 (SGD vs. Adam Generalization)**.** Discuss hypotheses for why SGD with momentum can outperform Adam on final generalization despite slower early progress.
Hint: Implicit regularization and noise structure.

Problem 8.16 (Design an Optimization Plan)**.** For training a ResNet-50 on a new 100-class dataset, propose an end-to-end optimization plan (initializer, optimizer, schedule, batch size, clipping, regularization) and justify choices.
Hint: Consider compute budget, stability, and expected generalization.

Problems

Easy

Problem 8.17 (Batch Size Selection)**.** Explain the trade-offs between using a large batch size versus a small batch size for training. Consider computation time, memory usage, and convergence properties.
Hint: Think about GPU utilization, gradient noise, and generalization gap.

Problem 8.18 (Momentum Intuition)**.** Describe how momentum helps accelerate optimization. Use the analogy of a ball rolling down a hill to explain the concept.
Hint: Consider how previous gradients influence the current update and help overcome small local variations.

Problem 8.19 (Learning Rate Scheduling)**.** List three common learning rate scheduling strategies and explain when each is most appropriate.
Hint: Consider step decay, exponential decay, cosine annealing, and cyclical learning rates.

Problem 8.20 (Adam Hyperparameters)**.** Adam optimizer has hyperparameters β_1 (typically 0.9) and β_2 (typically 0.999). Explain the role of each parameter.
Hint: β_1 controls momentum (first moment), β_2 controls adaptive learning rates (second moment).

Medium

Problem 8.21 (Optimizer Comparison)**.** Compare SGD with momentum, RMSProp, and Adam on a simple optimization problem. Discuss their convergence behavior and when to prefer one over another.
Hint: Consider sparse gradients, non-stationary objectives, and computational cost.

Problem 8.22 (Gradient Clipping)**.** Explain why gradient clipping is important for training recurrent neural networks. Derive the gradient clipping formula and discuss the choice of threshold.
Hint: Consider exploding gradients and the norm $||\nabla L||$. Clip by value or by norm.

Hard

Problem 8.23 (Natural Gradient Descent)**.** Derive the natural gradient update rule and explain why it is invariant to reparameterisations. Discuss the computational challenges of using natural gradient in deep learning.
Hint: Consider the Fisher information matrix $\boldsymbol{F}$ and the update $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \boldsymbol{F}^{-1} \nabla L$.

Problem 8.24 (Learning Rate Warmup)**.** Analyse why learning rate warmup is beneficial when training with large batch sizes. Provide theoretical justification based on the optimization landscape.
Hint: Consider the stability of gradients in early training and the sharpness of the loss landscape.

Chapter 9

Convolutional Networks

This chapter introduces convolutional neural networks (CNNs), which are particularly effective for processing grid-structured data like images. We build intuition first, then progressively add mathematical detail and modern algorithms, with historical context and key takeaways throughout [GBC16a; Pri23].

Learning Objectives

After completing this chapter, you will be able to:

1. **Explain the intuition of convolution, pooling, and receptive fields** and how they induce translation equivariance and parameter sharing.

2. **Derive output shapes and parameter counts** for common layer configurations (kernel size, stride, padding, channels).

3. **Implement core CNN algorithms**: convolution/cross-correlation, pooling, backpropagation through convolution, and residual connections.

4. **Compare classic architectures** (LeNet, AlexNet, VGG, Inception, ResNet, MobileNet/EfficientNet) and their design trade-offs.

5. **Apply CNNs to key tasks** in vision: classification, detection, and segmentation, and choose appropriate heads and losses.

6. **Cite historical milestones** that motivated CNN advances and understand why techniques were developed [LeC+89; KSH12; He+16; RFB15].

9.1 The Convolution Operation

Intuition

Convolution extracts local patterns by sliding small filters across the input, producing feature maps that respond strongly where patterns occur. Parameter sharing means the same filter detects the same pattern anywhere, yielding translation equivariance and dramatic parameter efficiency [GBC16a; Pri23].

9.1.1 Definition

The **convolution** operation applies a filter (kernel) across an input:
For discrete 2D convolution:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n) \tag{9.1}$$

where I is the input and K is the kernel. In deep learning libraries, the implemented operation is often cross-correlation (no kernel flip).
In practice, we often use **cross-correlation**:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n) \tag{9.2}$$

9.1.2 Properties

Parameter sharing: same kernel applied across spatial locations

- Dramatically reduces parameters compared to fully connected layers acting on flattened images.
- Yields *translation equivariance*: if the input shifts, the feature map shifts in the same way [GBC16a; Pri23]. Formally, letting T_δ denote a spatial shift, we have $(T_\delta I) * K = T_\delta(I * K)$ under appropriate boundary conditions.

Local connectivity: each output depends on a local input region (receptive field)

- Exploits spatial locality: nearby pixels are statistically dependent in natural images.
- *Compositionality*: stacking layers grows the effective receptive field, enabling detection of increasingly complex patterns (edges → corners → object parts) [GBC16a].

Linearity and nonlinearity:

- A single convolution is linear; CNNs interleave it with pointwise nonlinearities (e.g., ReLU) to model complex functions.
- With stride and padding fixed, convolution is a sparse matrix multiplication with a Toeplitz structure [GBC16a].

Padding and boundary effects: padding preserves spatial size and mitigates edge shrinkage. Without padding ("valid"), repeated convolutions rapidly reduce feature map size and bias features toward the center.

Stride and downsampling: stride $s > 1$ reduces spatial resolution. While efficient, aggressive striding early can remove fine detail; many designs delay downsampling to later stages (e.g., *conv stem* then stride) [KSH12; He+16].

Dilation (atrous convolutions): inserting *holes* between kernel elements increases receptive field without increasing parameters, useful for dense prediction (e.g., segmentation) [GBC16a].

Invariance via pooling: convolution is equivariant to translation; combining it with pooling or global aggregation introduces partial *translation invariance* in the representation [GBC16a].

Multi-channel mixing: kernels of shape $k \times k \times C_{\text{in}}$ learn both spatial and cross-channel interactions, while 1×1 convolutions mix channels without spatial coupling (used for bottlenecks and dimension reduction in Inception/ResNet).

9.1.3 Multi-Channel Convolution

For input with C_{in} channels and C_{out} output channels:

$$S_{c_{\text{out}}}(i,j) = \sum_{c_{\text{in}}=1}^{C_{\text{in}}} (I_{c_{\text{in}}} * K_{c_{\text{out}},c_{\text{in}}})(i,j) + b_{c_{\text{out}}} \tag{9.3}$$

9.1.4 Hyperparameters

Kernel size: Typically 3×3 or 5×5 (stacked 3×3 often preferred over a single 5×5 as in VGG [GBC16a])

Stride: Step size for sliding kernel (stride s):

$$\text{Output size} = \left\lfloor \frac{n-k}{s} \right\rfloor + 1 \tag{9.4}$$

Padding: Add zeros around input

- **Valid:** no padding
- **Same:** padding to preserve spatial size
- **Full:** maximum padding

For ”same” padding with stride 1:

$$p = \left\lfloor \frac{k-1}{2} \right\rfloor \tag{9.5}$$

9.2 Pooling

Pooling reduces spatial dimensions and provides translation invariance.

Intuition

Pooling summarizes nearby activations so that small translations of the input do not significantly change the summary. Max pooling keeps the strongest response, while average pooling smooths responses. It provides a degree of translation *invariance* complementary to convolution’s translation *equivariance* . Modern designs sometimes prefer strided convolutions to make downsampling learnable [GBC16a; He+16].

9.2.1 Max Pooling

Takes maximum value in each pooling region:

$$\text{MaxPool}(i, j) = \max_{m,n \in \mathcal{R}_{ij}} I(m, n) \tag{9.6}$$

Common: 2×2 max pooling with stride 2 (halves spatial dimensions).

Example. For input of size $H \times W = 32 \times 32$ and a 2×2 window with stride 2, the output is 16×16. Channels are pooled independently.

9.2.2 Average Pooling

Computes average:

$$\text{AvgPool}(i,j) = \frac{1}{|\mathcal{R}_{ij}|} \sum_{m,n \in \mathcal{R}_{ij}} I(m,n) \tag{9.7}$$

9.2.3 Global Pooling

Pools over entire spatial dimensions:

- **Global Average Pooling (GAP):** average over all spatial locations
- **Global Max Pooling:** maximum over all spatial locations

Useful for reducing parameters before fully connected layers and for connecting convolutional backbones to classification heads (e.g., GAP before softmax).

Note. Global average pooling (GAP) can replace large fully connected layers by averaging each feature map to a single scalar, reducing overfitting and parameter count [GBC16a].

9.2.4 Alternative: Strided Convolutions

Instead of a non-learned pooling operator, a convolution with stride $s > 1$ performs *learned downsampling*. For kernel size k, stride s, and padding p, the output spatial dimension per axis is

$$H' = \left\lfloor \frac{H - k + 2p}{s} \right\rfloor + 1, \quad W' = \left\lfloor \frac{W - k + 2p}{s} \right\rfloor + 1. \tag{9.8}$$

Pros and cons:

- **Pros:** learnable, can combine feature extraction and downsampling in one step; used in stage transitions of ResNet [He+16].
- **Cons:** may introduce aliasing if high-frequency content is not low-pass filtered prior to sub-sampling; anti-aliasing variants blur before stride.

Example. A 3×3 convolution with stride 2 and padding 1 keeps spatial size roughly halved (e.g., $32 \to 16$) while learning filters.

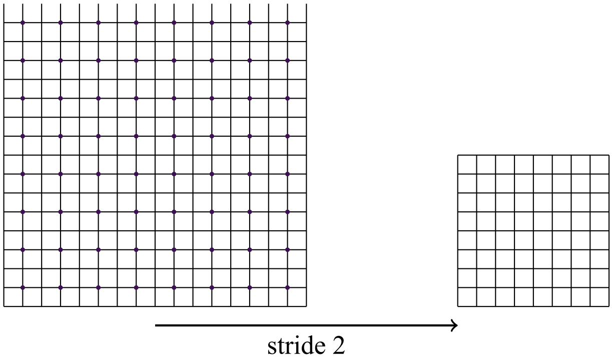

Figure 9.1: Downsampling via stride 2: fewer spatial samples after a strided convolution compared to pooling.

9.3 CNN Architectures ☒

Historical Context

CNNs evolved from early biologically inspired work to practical systems. **LeNet-5** established the template for digit recognition [LeC+89]. **AlexNet** showed large-scale training with ReLU, dropout, and data augmentation could dominate ImageNet [KSH12]. **VGG** emphasized simplicity via small filters, while **Inception** exploited multi-scale processing with 1×1 dimension reduction. **ResNet** enabled very deep networks via residual connections [He+16]. Efficiency-driven families like **MobileNet** and **EfficientNet** target edge devices and compound scaling.

9.3.1 LeNet-5 (1998)

LeNet-5 demonstrated the viability of CNNs for handwritten digit recognition (MNIST) [LeC+89]. It combined convolution, subsampling (pooling), and small fully connected layers.

- **Topology:** $\text{Conv}(6@5 \times 5) \to \text{Pool}(2 \times 2) \to \text{Conv}(16@5 \times 5) \to \text{Pool}(2 \times 2) \to \text{FC}(120) \to \text{FC}(84) \to \text{Softmax}(10)$.
- **Activations:** sigmoid/tanh; later works often retrofit ReLU for pedagogy.
- **Properties:** local receptive fields, parameter sharing, and early evidence of translation invariance via pooling.

- **Impact:** established the core conv-pool pattern and end-to-end learning for vision [GBC16a].

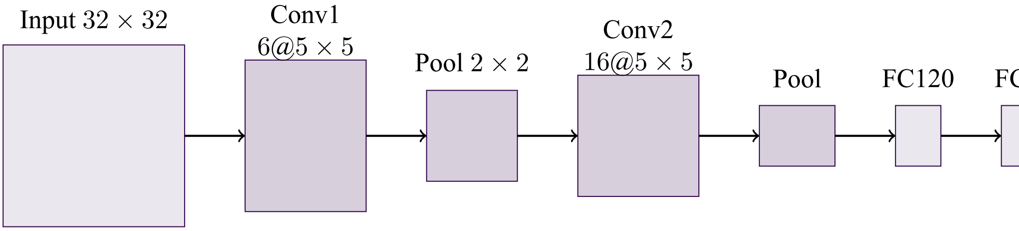

Figure 9.2: LeNet-5 architecture: alternating conv and pooling, followed by small fully connected layers.

9.3.2 AlexNet (2012)

AlexNet won ILSVRC 2012 by a large margin, catalyzing deep learning in vision [KSH12].

- **Design:** 5 conv + 3 FC layers; local response normalization (LRN) and overlapping pooling.
- **Optimization:** ReLU activations enabled faster training; heavy data augmentation; dropout in FC reduced overfitting.
- **Systems:** trained on 2 GPUs with model parallelism; used large kernels early and stride for rapid downsampling.
- **Impact:** established large-scale supervised pretraining on ImageNet as a standard.

9.3.3 VGG Networks (2014)

VGG emphasized depth with a simple recipe [GBC16a]: stacks of 3 × 3 convolutions with stride 1 and 2 × 2 max pooling for downsampling.

- **Uniform blocks:** replacing large kernels by multiple 3 × 3 layers increases nonlinearity and receptive field with fewer parameters.
- **Models:** VGG-16 and VGG-19; very large parameter counts in dense layers.

- **Trade-offs:** strong accuracy but memory/computation heavy; often used as feature extractors.

9.3.4 ResNet (2015)

ResNet introduced **identity skip connections** to learn residual functions [He+16]:

$$\boldsymbol{y} = \mathcal{F}(\boldsymbol{x}, \{\boldsymbol{W}_i\}) + \boldsymbol{x}, \tag{9.9}$$

where $\mathcal{F}$ is typically a small stack of convolutions and normalization/activation.

- **Depth:** enabled 50/101/152-layer models with stable optimization.
- **Gradient flow:** the Jacobian includes an identity term, mitigating vanishing gradients.
- **Blocks:** basic (two 3×3) and bottleneck (1×1-3×3-1×1) with projection shortcuts for dimension changes.

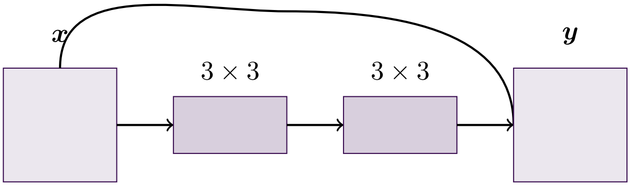

Figure 9.3: ResNet basic residual block with identity skip connection.

9.3.5 Inception/GoogLeNet (2014)

GoogLeNet popularized **Inception modules** with parallel multi-scale branches and 1×1 bottlenecks for efficiency.

- **Branches:** 1×1, 3×3, 5×5 convolutions and a pooled branch, then channel-wise concatenation.
- **Efficiency:** 1×1 convolutions reduce channel dimensions before larger kernels, cutting FLOPs while preserving capacity.
- **Impact:** competitive accuracy with fewer parameters than VGG; design influenced later hybrid networks.

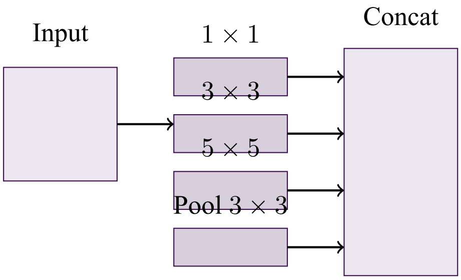

Figure 9.4: Inception module: parallel multi-scale branches concatenated along channels with 1×1 bottlenecks.

9.3.6 MobileNet and EfficientNet

MobileNet. Prioritizes efficiency for edge devices using depthwise separable convolutions (depthwise $k \times k$ followed by pointwise 1×1), drastically reducing FLOPs and parameters while maintaining accuracy.

EfficientNet. Introduces compound scaling to jointly scale depth, width, and resolution with a principled coefficient, yielding strong accuracy/efficiency trade-offs. Variants (B0–B7) demonstrate near-optimal Pareto fronts.

For introductory treatment of these families, see *D2L* (modern CNNs) and *Deep Learning* (convolutional networks).

9.4 Applications of CNNs ☒

Intuition

Backbones of stacked convolutions extract spatially local features that become increasingly abstract with depth. Task-specific heads (classification, detection, segmentation) transform backbone features into outputs appropriate to the problem [GBC16a; Pri23].

9.4.1 Image Classification

Task: assign a label to the entire image.

Backbone + head:

- Convolutional backbone extracts hierarchical features.

- Downsampling via pooling or strided convolutions.
- Global average pooling and a small fully connected (or 1×1 conv) layer with softmax.

Examples and datasets: CIFAR-10/100, ImageNet (ILSVRC). Transfer learning from ImageNet pretraining commonly improves downstream tasks.

9.4.2 Object Detection

Task: localize and classify objects with bounding boxes.
Region-based (two-stage):

- R-CNN: region proposals + CNN features (slow).
- Fast/Faster R-CNN: integrate feature extraction; Faster learns proposals (RPN).
- Mask R-CNN: extends with an instance segmentation branch.

Single-shot (one-stage):

- YOLO: dense predictions at multiple scales with real-time speed.
- SSD: default boxes across feature maps; efficient multi-scale detection.

Heads and losses: classification (cross-entropy/focal loss), box regression (smooth-ℓ_1 or IoU losses), non-maximum suppression (NMS) at inference.

9.4.3 Semantic Segmentation

Task: assign a class label to each pixel.
Architectures:

- FCN: replace dense layers with 1×1 convs and upsample (deconvolution) to input resolution.
- U-Net: encoder-decoder with skip connections for precise localization; widely used in medical imaging [RFB15].
- Atrous/dilated convolutions: enlarge receptive field without losing resolution.

Losses and metrics: pixel-wise cross-entropy, Dice/IoU; mIoU for evaluation.

9.5 Core CNN Algorithms ⊠

We introduce algorithms progressively, starting from basic cross-correlation to residual learning.

9.5.1 Cross-Correlation and Convolution

Given input $I \in \mathbb{R}^{H \times W \times C_{\text{in}}}$ and kernel $K \in \mathbb{R}^{k \times k \times C_{\text{in}} \times C_{\text{out}}}$, the output feature map $S \in \mathbb{R}^{H' \times W' \times C_{\text{out}}}$ under stride s and padding p is computed by cross-correlation as in Section 9.1. Libraries often refer to this as ”convolution” [GBC16a].

9.5.2 Backpropagation Through Convolution

For loss $\mathcal{L}$ and pre-activation output $S = I * K$, the gradients are:

$$\frac{\partial \mathcal{L}}{\partial K} = I \star \frac{\partial \mathcal{L}}{\partial S}, \tag{9.10}$$

$$\frac{\partial \mathcal{L}}{\partial I} = \frac{\partial \mathcal{L}}{\partial S} * K^{\text{rot}}, \tag{9.11}$$

where $\star$ denotes cross-correlation, $*$ denotes convolution, and K^{rot} is the kernel rotated by 180°. Implementations use efficient im2col/FFT variants [GBC16a].

Example (shape-aware): For $I \in \mathbb{R}^{32 \times 32 \times 64}$ and $K \in \mathbb{R}^{3 \times 3 \times 64 \times 128}$ with stride 1 and same padding, $S \in \mathbb{R}^{32 \times 32 \times 128}$. The gradient w.r.t. K accumulates over spatial locations and batch.

9.5.3 Pooling Backpropagation

For max pooling, the upstream gradient is routed to the maximal input in each region; for average pooling, it is evenly divided among elements.

9.5.4 Residual Connections

In a residual block with input $\boldsymbol{x}$ and residual mapping $\mathcal{F}$, the output is $\boldsymbol{y} = \mathcal{F}(\boldsymbol{x}) + \boldsymbol{x}$. Backpropagation yields $\frac{\partial \mathcal{L}}{\partial \boldsymbol{x}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{y}} \left(\frac{\partial \mathcal{F}}{\partial \boldsymbol{x}} + \boldsymbol{I} \right)$, stabilizing gradients and enabling very deep nets [He+16].

9.5.5 Progressive Complexity: Depthwise Separable Convolutions

Depthwise separable convolution factors standard convolution into depthwise (per-channel) and pointwise (1×1) operations, reducing FLOPs and parameters (used by MobileNet). This keeps representational power while improving efficiency.

Parameter comparison. For $C_{\text{in}} = C_{\text{out}} = c$ and kernel $k \times k$:

$$\text{standard} = k^2 c^2, \tag{9.12}$$

$$\text{depthwise separable} = k^2 c + c^2, \quad \text{saving} \approx 1 - \frac{k^2 c + c^2}{k^2 c^2}. \tag{9.13}$$

9.5.6 Normalization and Activation

Batch normalization [IS15] and ReLU-family activations improve optimization and generalization by smoothing the loss landscape and mitigating covariate shift.

9.5.7 Other Useful Variants

- **Group/Layer Normalization:** alternatives when batch sizes are small.
- **Dilated Convolutions:** expand receptive field without pooling; effective in segmentation.
- **Anti-aliased Downsampling:** blur pooling or low-pass before stride to reduce aliasing artifacts in features.

Key Takeaways

- Cross-correlation is the practical ”convolution” in deep learning; parameter sharing and locality are core.
- Residual connections preserve gradient flow, enabling very deep networks.
- Efficient convolutions (depthwise separable) trade compute for accuracy with strong Pareto gains.
- Normalization and careful downsampling are critical for stable optimization and preserving information.

9.6 Real World Applications

Convolutional neural networks have revolutionized how computers understand images and videos. Their applications touch nearly every aspect of modern visual technology.

9.6.1 Medical Image Analysis

CNNs help doctors diagnose diseases more accurately and quickly:

- **Cancer detection in radiology:** CNNs analyze X-rays, CT scans, and MRIs to detect tumors often invisible to the human eye. For example, mammography systems using CNNs can spot breast cancer earlier than traditional methods, potentially saving thousands of lives annually. The networks learn to recognize subtle patterns that indicate malignancy.

- **Diabetic retinopathy screening:** CNNs examine photos of patients' eyes to detect diabetes-related damage before vision loss occurs. This allows automated screening in remote areas without specialist ophthalmologists, making eye care accessible to millions more people worldwide.

- **Skin cancer classification:** Smartphone apps with CNNs let people photograph suspicious moles for instant preliminary assessment. While not replacing doctors, these tools encourage early medical consultation when something looks concerning.

9.6.2 Autonomous Driving

Self-driving cars rely on CNNs to understand their surroundings:

- **Object detection and tracking:** CNNs process camera feeds to identify pedestrians, other vehicles, traffic signs, and lane markings in real-time. The network must work perfectly under varied conditions—rain, snow, nighttime, construction zones—because lives depend on it.

- **Depth estimation:** CNNs analyze images to determine how far away objects are, helping vehicles make safe decisions about braking, turning, and merging. This works even with regular cameras, though it's enhanced when combined with other sensors.

- **Semantic segmentation:** CNNs label every pixel in the camera view (road, sidewalk, vehicle, sky, etc.), giving the vehicle complete understanding of its environment. This pixel-level understanding enables precise navigation.

9.6.3 Content Moderation and Safety

Social media platforms use CNNs to keep online spaces safe:

- **Inappropriate content detection:** CNNs scan billions of uploaded images and videos daily, automatically flagging harmful content (violence, explicit material, hate symbols) for human review. This happens before most users ever see problematic content.

- **Face blurring for privacy:** News organizations and mapping services use CNNs to automatically blur faces and license plates in photos and street view imagery, protecting people's privacy while sharing useful information.

- **Copyright protection:** CNNs help platforms identify copyrighted images and videos, preventing unauthorized sharing while allowing legitimate uses. This technology processes millions of uploads per hour.

9.6.4 Everyday Applications

CNNs power features you use daily:

- **Photo organization:** Your phone groups photos by people, places, and things

- **Visual search:** Find products by taking photos instead of typing descriptions

- **Document scanning:** Apps automatically detect document edges and enhance readability

- **Augmented reality:** Filters and effects in camera apps that track faces and scenes

These applications show how CNNs transform abstract computer vision research into practical tools that improve healthcare, safety, and daily convenience.

9.7 Problems ☒

Instructions

Attempt the following problems in order. Hints are provided to guide your reasoning.

Easy

E1. Output Size Basics: For input 32×32, kernel 5×5, stride 1, padding 2, what is the output size?
Hint: Use $\left\lfloor \frac{n-k+2p}{s} \right\rfloor + 1$.

E2. Parameter Count: For a conv layer with $C_{in} = 3$, $C_{out} = 16$, kernel 3×3, how many weights and biases?
Hint: Weights $= 3 \cdot 3 \cdot C_{in} \cdot C_{out}$; biases $= C_{out}$.

E3. Receptive Field Growth: Two stacked 3×3 convs (stride 1, same padding). What is the effective receptive field?
Hint: Add $k - 1$ per layer.

E4. Pooling Downsampling: How many 2×2 max-pooling layers (stride 2) reduce 128×128 to 8×8?
Hint: Halve per layer.

E5. Cross-Correlation vs Convolution: What is the difference?
Hint: Kernel flip vs no flip.

E6. GAP Head: Why does global average pooling reduce parameters compared to FC?
Hint: Eliminates dense connections over spatial maps.

Medium

M1. Strided Convolution Shape: Input $64 \times 64 \times 64$, 3×3 conv, $C_{out} = 128$, stride 2, padding 1. What is the output shape?
Hint: Apply formula per spatial dimension.

M2. FLOPs Estimate: Estimate multiply-accumulate ops for the above layer.
Hint: $H'W' \cdot k^2 \cdot C_{in} \cdot C_{out}$.

M3. Residual Block Gradient: Explain why adding identity improves gradient flow.
Hint: Jacobian adds $\boldsymbol{I}$.

M4. Depthwise Separable Savings: Compare parameters of standard vs depthwise+pointwise for $C_{in} = C_{out} = 256$, $k = 3$.
Hint: Standard $= k^2 C_{in} C_{out}$; separable $= k^2 C_{in} + C_{in} C_{out}$.

M5. VGG vs Inception: Contrast design philosophies.
Hint: Small uniform filters vs multi-branch multi-scale with 1×1 bottlenecks.

Hard

H1. Derive Backprop for Conv: Starting from $S = I \star K$, derive $\partial \mathcal{L} / \partial K$ and $\partial \mathcal{L} / \partial I$.
Hint: Use index notation and chain rule with shifts.

H2. Receptive Field in ResNet Stage: For a stage of N residual blocks each with 3×3 convs, stride 1, what is the receptive field increase relative to input?
Hint: Each 3×3 adds 2; consider two per block.

H3. Alias and Stride: Discuss when strided conv without pre-filtering can alias features.
Hint: Nyquist; low-pass before downsampling.

H4. Effective Stride and Dilation: Show how dilation increases receptive field without increasing parameters.
Hint: Holes in kernels; spacing factor.

H5. Designing a Lightweight Backbone: Propose a backbone achieving < 300 MFLOPs at 224×224 with competitive accuracy.
Hint: Depthwise separable convs, squeeze-expansion, or compound scaling.

Key Takeaways

- Start with simple conv-pool stacks; add residuals and normalization as depth grows.
- Prefer small kernels and learn downsampling where appropriate; consider efficiency via separable convs.

- Choose heads tailored to the task (classification/detection/segmentation) and ensure shapes match.

Key Takeaways

Key Takeaways 9

- **Convolution and pooling** exploit spatial structure through parameter sharing and local receptive fields, achieving translation equivariance.
- **Classic architectures** progressively deepened networks (LeNet → AlexNet → VGG → ResNet) via innovations like batch normalisation and residual connections.
- **ResNets solve vanishing gradients** by adding skip connections, enabling training of very deep networks.
- **Modern CNNs balance efficiency and accuracy** through depthwise separable convolutions (MobileNet) and compound scaling (EfficientNet).
- **CNNs excel in vision tasks**: classification, object detection, and semantic segmentation, with task-specific heads and losses.

Problems

Easy

Problem 9.1 (Receptive Field Calculation). A CNN has two convolutional layers with 3×3 kernels (no padding, stride 1). Calculate the receptive field of a neuron in the second layer.
Hint: Each layer expands the receptive field. For the second layer, consider how many input pixels affect it.

Problem 9.2 (Parameter Counting). Calculate the number of parameters in a convolutional layer with 64 input channels, 128 output channels, and 3×3 kernels (including bias).
Hint: Each output channel has a 3×3 kernel for each input channel, plus one bias term.

Problem 9.3 (Pooling Operations)**.** Explain the difference between max pooling and average pooling. When would you prefer one over the other?
Hint: Consider feature prominence, spatial information retention, and gradient flow.

Problem 9.4 (Translation Equivariance)**.** Explain what translation equivariance means in the context of CNNs and why it is a desirable property for image processing.
Hint: If the input is shifted, how does the output change? Consider the relationship $f(T(x)) = T(f(x))$.

Medium

Problem 9.5 (Output Shape Calculation)**.** Given an input image of size 224×224×3, apply the following operations and calculate the output shape at each step:

1. Conv2D: 64 filters, 7×7 kernel, stride 2, padding 3
2. MaxPool2D: 3×3, stride 2
3. Conv2D: 128 filters, 3×3 kernel, stride 1, padding 1

Hint: Use the formula: $\text{output\_size} = \lfloor \frac{\text{input\_size}+2\times\text{padding}-\text{kernel\_size}}{\text{stride}} \rfloor + 1$.

Problem 9.6 (ResNet Skip Connections)**.** Explain why residual connections (skip connections) help train very deep networks. Discuss the gradient flow through skip connections.
Hint: Consider the identity mapping $\boldsymbol{y} = \boldsymbol{x} + F(\boldsymbol{x})$ and compute $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$.

Hard

Problem 9.7 (Dilated Convolutions)**.** Derive the receptive field for a stack of dilated convolutions with dilation rates [1, 2, 4, 8]. Compare computational cost with standard convolutions achieving the same receptive field.
Hint: Dilated convolution with rate r introduces $(r - 1)$ gaps between kernel elements. Track receptive field growth layer by layer.

Problem 9.8 (Depthwise Separable Convolutions)**.** Analyse the computational savings of depthwise separable convolutions (as used in MobileNets) compared to standard convolutions. Derive the reduction factor for a layer with C_{in} input channels, C_{out} output channels, and K×K kernel size.
Hint: Depthwise separable splits into depthwise (C_{in} groups) and pointwise (1×1) convolutions. Compare FLOPs.

Chapter 10

Sequence Modeling: Recurrent and Recursive Nets

This chapter covers architectures designed for sequential and temporal data, including recurrent neural networks (RNNs) and their variants.

Learning Objectives

After completing this chapter, you will be able to:

1. **Explain why sequence models are needed** and identify data modalities that require temporal context.
2. **Describe and compare** vanilla RNNs, LSTMs, and GRUs, including their gating mechanisms and trade-offs.
3. **Implement and reason about** backpropagation through time (BPTT) and truncated BPTT, including gradient clipping.
4. **Build sequence-to-sequence models with attention** and explain the intuition behind alignment and context vectors.
5. **Apply advanced decoding and architecture variants** such as bidirectional RNNs, teacher forcing, and beam search.
6. **Evaluate common failure modes** (vanishing/exploding gradients, exposure bias) and mitigation strategies.

10.1 Recurrent Neural Networks ☒

Intuition

An RNN carries a running summary of the past—like a notepad you update after reading each word. This hidden state lets the model use prior context to influence current predictions. However, keeping reliable notes over long spans is hard: small errors can compound, and gradients may shrink or grow too much [GBC16a].

Historical Context

Early sequence models struggled with long-term dependencies due to vanishing gradients, motivating gated designs such as LSTM [HS97]. Practical training stabilized with techniques like gradient clipping and better initialization [GBC16a].

10.1.1 Motivation

Sequential data exhibits *temporal dependencies* and *order-sensitive* structure that cannot be modeled well by i.i.d. assumptions or fixed-size context windows alone [GBC16a; Pri23; Bis06]. Examples include:

- Time series: forecasting energy load, financial returns, or physiological signals (ECG).
- Natural language: the meaning of a word depends on its context; sentences conform to syntax and discourse structure.
- Speech/audio: phonemes combine to form words; coarticulation effects span multiple frames.
- Video: actions unfold over time; temporal cues disambiguate similar frames.
- Control and reinforcement learning: actions influence future observations, requiring memory.

Classic feedforward networks assume fixed-size inputs and lack a persistent state, making them ill-suited for long-range dependencies. Recurrent architectures introduce a hidden state that acts as a compact, learned memory and enables conditioning on arbitrary-length histories. Historically, recurrent ideas trace back to early neural sequence models and dynamical systems; practical training matured with BPTT [RHW86] and later with gated units to mitigate vanishing/exploding gradients [HS97]. For further background see the RNN overview in [GBC16a] and educational treatments in [Zha+24c; Wik25b; GBC16c].

10.1.2 Basic RNN Architecture

An RNN maintains a hidden state $\boldsymbol{h}_t$ that evolves over time:

$$\boldsymbol{h}_t = \sigma(\boldsymbol{W}_{hh}\boldsymbol{h}_{t-1} + \boldsymbol{W}_{xh}\boldsymbol{x}_t + \boldsymbol{b}_h) \tag{10.1}$$

$$\boldsymbol{y}_t = \boldsymbol{W}_{hy}\boldsymbol{h}_t + \boldsymbol{b}_y \tag{10.2}$$

where $\boldsymbol{x}_t$ is input at time t, and σ is typically tanh.

Visual aid. The following unrolled diagram shows shared parameters across time:

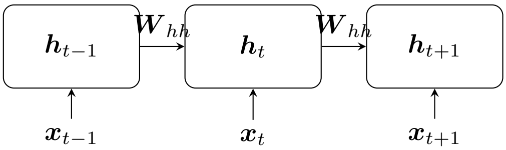

Figure 10.1: Unrolled RNN with shared parameters across time steps.

10.1.3 Unfolding in Time

RNNs can be ”unrolled” into a deep feedforward computation graph over time with *shared parameters*. This perspective clarifies how gradients flow backward through temporal connections and why depth-in-time can cause vanishing/exploding gradients [GBC16a].

$$\boldsymbol{h}_t = f(\boldsymbol{h}_{t-1}, \boldsymbol{x}_t; \boldsymbol{\theta}) \tag{10.3}$$

Visual aid. Unfolding reveals repeated applications of the same transition function across steps. We annotate inputs, hidden states, and outputs to emphasize sharing and the temporal chain rule during BPTT.

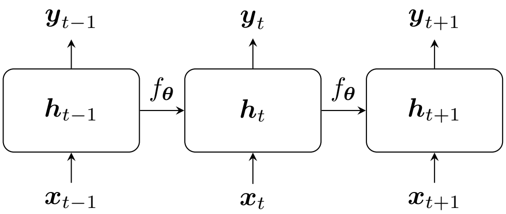

Figure 10.2: Unrolling an RNN across time: the same parameters $\boldsymbol{\theta}$ are reused at each step.

This view connects RNNs to dynamic Bayesian networks and emphasizes that training complexity scales with the unroll length. See [Wik25b; GBC16a; Zha+24c].

10.1.4 Types of Sequences

One-to-one: Fixed-size input to fixed-size output with temporal structure ignored or not present (e.g., static image classification with no time component).

One-to-many: Single input to sequence output (e.g., image captioning, where an image produces a sequence of words).

Many-to-one: Sequence input to single output (e.g., sentiment classification from a review; keyword spotting from audio).

Many-to-many (synchronous): Sequence labeling with aligned input/output lengths (e.g., part-of-speech tagging, CTC-free frame labeling).

Many-to-many (asynchronous): Sequence transduction with potentially different lengths (e.g., machine translation, speech recognition with CTC/attention). Attention helps bridge length mismatch by learning soft alignments [BCB14].

Design choices (teacher forcing, bidirectionality, attention, beam search) depend on whether future context is available and whether output timing must be causal. See [Zha+24c; Wik25b] for further taxonomy.

See [GBC16a; Pri23; Bis06; Wik25b; GBC16c; Zha+24c] for introductions to sequence modeling and RNNs.

10.2 Backpropagation Through Time ☒

Intuition

BPTT treats the unrolled RNN as a deep network across time and applies backpropagation through each time slice. Gradients flow backward along temporal edges, accumulating effects from future steps. Truncation limits how far signals propagate to balance cost and dependency length [GBC16a].

Historical Context

The backpropagation algorithm [RHW86] enabled efficient training of deep networks; applying it to unrolled RNNs became known as BPTT. Awareness of vanishing/exploding gradients led to clipping and gated architectures [GBC16a; HS97].

10.2.1 BPTT Algorithm

Gradients are computed by unrolling the network and applying backpropagation through the temporal graph. Let $L = \sum_{t=1}^{T} L_t$ and $\boldsymbol{h}_t = f(\boldsymbol{h}_{t-1}, \boldsymbol{x}_t; \boldsymbol{\theta})$, $\boldsymbol{y}_t = g(\boldsymbol{h}_t; \boldsymbol{\theta}_y)$. The total derivative w.r.t. hidden states satisfies the recurrence:

$$\frac{\partial L}{\partial \boldsymbol{h}_t} = \frac{\partial L}{\partial \boldsymbol{y}_t} \frac{\partial \boldsymbol{y}_t}{\partial \boldsymbol{h}_t} + \frac{\partial L}{\partial \boldsymbol{h}_{t+1}} \frac{\partial \boldsymbol{h}_{t+1}}{\partial \boldsymbol{h}_t} \tag{10.4}$$

For any parameter block $\boldsymbol{W} \in \boldsymbol{\theta}$ appearing at each time step:

$$\frac{\partial L}{\partial \boldsymbol{W}} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial \boldsymbol{W}} \tag{10.5}$$

Algorithm (BPTT).

1. Forward pass: for $t = 1, \ldots, T$, compute $\boldsymbol{h}_t$, $\boldsymbol{y}_t$, and L_t.
2. Initialize temporal gradients: $\frac{\partial L}{\partial \boldsymbol{h}_{T+1}} = \boldsymbol{0}$.
3. Backward pass: for $t = T, \ldots, 1$:

$$\begin{aligned} \delta_t &\leftarrow \frac{\partial L}{\partial \boldsymbol{y}_t} \frac{\partial \boldsymbol{y}_t}{\partial \boldsymbol{h}_t} + \left(\frac{\partial L}{\partial \boldsymbol{h}_{t+1}} \right) \frac{\partial \boldsymbol{h}_{t+1}}{\partial \boldsymbol{h}_t} \\ \frac{\partial L}{\partial \boldsymbol{W}} &\mathrel{+}= \delta_t \frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{W}} \quad \text{(or add contributions for all parameters at time } t\text{)} \end{aligned}$$

4. Apply gradient clipping if needed and update parameters.

This view aligns with the computational-graph treatment in [GBC16a] and standard expositions [GBC16c; Zha+24c].

10.2.2 Vanishing and Exploding Gradients

Gradients can vanish or explode exponentially:

$$\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_k} = \prod_{i=k+1}^{t} \frac{\partial \boldsymbol{h}_i}{\partial \boldsymbol{h}_{i-1}} = \prod_{i=k+1}^{t} \boldsymbol{W}^\top \text{diag}(\sigma'(\boldsymbol{z}_i)) \tag{10.6}$$

If eigenvalues of $\boldsymbol{W}$ are:

- < 1: gradients vanish
- > 1: gradients explode

Solutions:

- Gradient clipping (for explosion)
- Careful initialization
- ReLU activation
- LSTM/GRU architectures

10.2.3 Truncated BPTT

For very long sequences, truncate gradient computation by limiting backpropagation to a sliding window of k steps [GBC16a]:

- Process inputs in segments of length k (possibly overlapping with stride s).
- Backpropagate gradients only within each segment to reduce memory and time.
- Hidden state is carried forward between segments but treated as a constant during the truncated backward step.

Algorithm (Truncated BPTT).

1. For $t = 1, 1+s, 1+2s, \ldots$, take the chunk $[t,\ t+k-1]$.
2. Run forward pass over the chunk using the current hidden state as initialization.
3. Backpropagate losses only within the chunk; accumulate gradients.
4. Optionally detach the final hidden state from the graph before the next chunk to bound gradient length.

Trade-offs: Lower memory and latency versus potentially missing very long-range dependencies. Increasing k improves dependency length coverage but increases cost. Hybrids with dilated RNNs or attention can mitigate the trade-off.

10.3 Long Short-Term Memory (LSTM) ☒

Intuition

The LSTM adds a highway for information (the cell state) that can pass signals forward with minimal modification. Gates act like valves to forget unhelpful information, write new content, and reveal outputs, which preserves gradients over long spans [HS97; GBC16a].

Historical Context

Introduced in the 1990s to address vanishing gradients [HS97], LSTMs unlocked practical sequence learning across speech, language, and time-series tasks before attention-based Transformers became dominant [Vas+17].

10.3.1 Architecture

LSTM uses **gating mechanisms** to control information flow and maintain a persistent cell state that supports long-range credit assignment [HS97; GBC16a]:

$$\boldsymbol{f}_t = \sigma(\boldsymbol{W}_f[\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_f) \quad \text{(forget gate)} \tag{10.7}$$

$$\boldsymbol{i}_t = \sigma(\boldsymbol{W}_i[\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_i) \quad \text{(input gate)} \tag{10.8}$$

$$\tilde{\boldsymbol{c}}_t = \tanh(\boldsymbol{W}_c[\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_c) \quad \text{(candidate)} \tag{10.9}$$

$$\boldsymbol{c}_t = \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \odot \tilde{\boldsymbol{c}}_t \quad \text{(cell state)} \tag{10.10}$$

$$\boldsymbol{o}_t = \sigma(\boldsymbol{W}_o[\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_o) \quad \text{(output gate)} \tag{10.11}$$

$$\boldsymbol{h}_t = \boldsymbol{o}_t \odot \tanh(\boldsymbol{c}_t) \quad \text{(hidden state)} \tag{10.12}$$

10.3.2 Key Ideas

Cell state $\boldsymbol{c}_t$: Long-term memory

- Information flows with minimal transformation
- Gates control what to remember/forget

Forget gate $\boldsymbol{f}_t$: Decides what to discard from cell state
Input gate $\boldsymbol{i}_t$: Decides what new information to store
Output gate $\boldsymbol{o}_t$: Decides what to output

10.3.3 Advantages

- Addresses vanishing gradient problem
- Can learn long-term dependencies
- Gradients flow more easily through cell state
- Widely used for sequential tasks

Visual aid. A compact LSTM cell diagram:

10.4 Gated Recurrent Units (GRU) ☒

Intuition

GRU simplifies LSTM by merging cell and hidden state and combining gates, often matching performance with fewer parameters—useful when data or compute is limited [Cho+14; GBC16a].

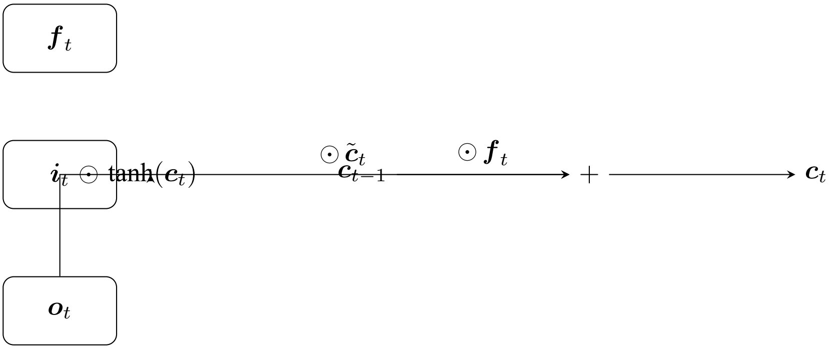
$\boldsymbol{f}_t$
$\boldsymbol{i}_t \odot \tanh(\boldsymbol{c}_t)$
$\odot\, \tilde{\boldsymbol{c}}_t$
$\boldsymbol{c}_{t-1}$
$\odot\, \boldsymbol{f}_t$
$+$
$\boldsymbol{c}_t$
$\boldsymbol{o}_t$

Figure 10.3: Illustrative LSTM flow with forget, input, and output gates.

Historical Context

Proposed in the mid-2010s, GRU offered a practical alternative to LSTM with competitive empirical results and simpler implementation [Cho+14].

10.4.1 Architecture

GRU simplifies LSTM with fewer parameters and merges the cell and hidden state into a single vector, often yielding comparable performance with less computation [Cho+14; GBC16a]:

$$\boldsymbol{z}_t = \sigma(\boldsymbol{W}_z[\boldsymbol{h}_{t-1}, \boldsymbol{x}_t]) \quad \text{(update gate)} \tag{10.13}$$

$$\boldsymbol{r}_t = \sigma(\boldsymbol{W}_r[\boldsymbol{h}_{t-1}, \boldsymbol{x}_t]) \quad \text{(reset gate)} \tag{10.14}$$

$$\tilde{\boldsymbol{h}}_t = \tanh(\boldsymbol{W}[\boldsymbol{r}_t \odot \boldsymbol{h}_{t-1}, \boldsymbol{x}_t]) \quad \text{(candidate)} \tag{10.15}$$

$$\boldsymbol{h}_t = (1 - \boldsymbol{z}_t) \odot \boldsymbol{h}_{t-1} + \boldsymbol{z}_t \odot \tilde{\boldsymbol{h}}_t \tag{10.16}$$

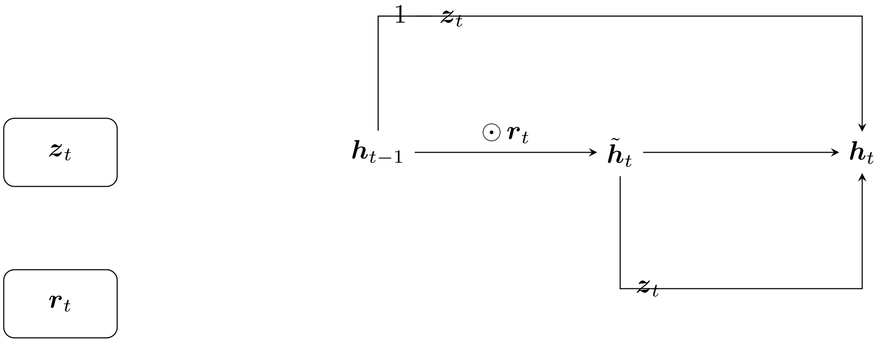

Figure 10.4: Illustrative GRU flow with update and reset gates.

	GRU	**LSTM**
State	Single hidden state $\boldsymbol{h}_t$	Hidden $\boldsymbol{h}_t$ and cell $\boldsymbol{c}_t$
Gates	Update, reset	Input, forget, output
Parameters	Fewer (often faster)	More (more expressive)
Long-range deps.	Good in practice	Often stronger on very long spa
Simplicity	Simpler to implement	Slightly more complex
Typical use	Smaller data/compute budgets	Longer sequences or when cap helps

Table 10.1: GRU vs. LSTM at a glance [Cho+14; HS97; GBC16a].

10.4.2 Architecture (visual)

10.4.3 Comparison with LSTM

10.5 Sequence-to-Sequence Models ⊠

Intuition

Encoder – decoder models compress a source sequence into a representation and then generate a target sequence step-by-step. Attention augments this by letting the decoder look back at encoder states as needed, creating a dynamic context per output token [Cho+14; BCB14].

Historical Context

Early seq2seq relied on fixed context vectors, which degraded on long inputs. Content-based attention [BCB14] lifted this bottleneck and paved the way toward Transformer architectures [Vas+17].

10.5.1 Encoder-Decoder Architecture

For sequence transduction tasks like machine translation [Cho+14; BCB14]:

Encoder: Processes input sequence into representation (fixed or per-step states)

$$\boldsymbol{c} = f(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_T) \tag{10.17}$$

Decoder: Generates output sequence from representation

Visual aid. A minimal encoder – decoder with context vector.

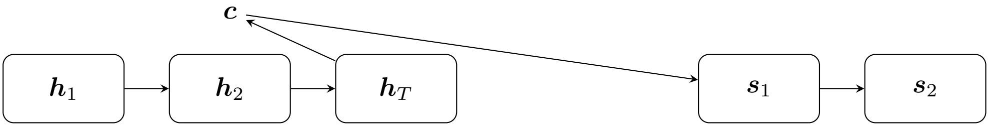

Figure 10.5: Encoder – decoder with a fixed context vector $\boldsymbol{c}$. Attention replaces $\boldsymbol{c}$ with step-dependent $\boldsymbol{c}_t$.

$$\boldsymbol{y}_t = g(\boldsymbol{y}_{t-1}, \boldsymbol{c}, \boldsymbol{s}_{t-1}) \tag{10.18}$$

10.5.2 Attention Mechanism

Standard seq2seq compresses entire input into fixed vector $\boldsymbol{c}$, causing information bottleneck.

Attention allows the decoder to focus on relevant input parts by computing a content-based weighted average of encoder states [BCB14]. At each decoding step t:

$$e_{ti} = a(\boldsymbol{s}_{t-1}, \boldsymbol{h}_i) \quad \text{(alignment scores)} \tag{10.19}$$

$$\alpha_{ti} = \frac{\exp(e_{ti})}{\sum_j \exp(e_{tj})} \quad \text{(attention weights)} \tag{10.20}$$

$$\boldsymbol{c}_t = \sum_i \alpha_{ti} \boldsymbol{h}_i \quad \text{(context vector)} \tag{10.21}$$

Common scoring functions $a(\cdot)$ include additive (Bahdanau) attention using a small MLP, and multiplicative/dot-product attention which is parameter-efficient and forms the basis for scaled dot-product attention in Transformers [Vas+17]. Attention weights α_{ti} are interpretable as soft alignments between target position t and source position i (see [Wik25a]). For a broader overview, consult standard references [GBC16c; Zha+24a].

Training and inference. Attention is trained end-to-end with the seq2seq objective. During inference, attention enables the model to retrieve the most relevant encoder features for each generated token, improving long-input performance and handling reordering. Variants include multi-head attention, local/monotonic attention for streaming, and coverage terms to reduce repetition.
Benefits:

- Dynamic context for each output
- Better for long sequences
- Interpretable (visualize attention weights)

10.5.3 Applications

- Machine translation (NMT)
- Text summarization (extractive and abstractive)
- Question answering and dialogue systems
- Image captioning (CNN/ViT encoder, RNN/Transformer decoder)
- Speech recognition and speech translation
- OCR and handwriting recognition

- Code generation and program repair

10.6 Advanced Topics ☒

Intuition

Variants extend context (bidirectional), depth (stacked layers), supervision signals (teacher forcing), and search quality (beam search). These choices trade training stability, inference latency, and quality depending on the task.

Historical Context

Bidirectional RNNs improved context use for labeling tasks; teacher forcing stabilized decoder training but highlighted exposure bias; beam search became standard for autoregressive decoding in translation and speech.

10.6.1 Bidirectional RNNs

Process sequence in both directions:

$$\overrightarrow{\boldsymbol{h}}_t = f(\boldsymbol{x}_t, \overrightarrow{\boldsymbol{h}}_{t-1}) \tag{10.22}$$

$$\overleftarrow{\boldsymbol{h}}_t = f(\boldsymbol{x}_t, \overleftarrow{\boldsymbol{h}}_{t+1}) \tag{10.23}$$

$$\boldsymbol{h}_t = [\overrightarrow{\boldsymbol{h}}_t; \overleftarrow{\boldsymbol{h}}_t] \tag{10.24}$$

Useful when future context is available.

Use cases and caveats. Effective for tagging, chunking, and ASR with full utterances, but not suitable for strictly causal, low-latency streaming since backward states require future tokens. Alternatives include limited-lookahead or online approximations.

10.6.2 Deep RNNs

Stack multiple RNN layers:

$$\boldsymbol{h}_t^{(l)} = f(\boldsymbol{h}_t^{(l-1)}, \boldsymbol{h}_{t-1}^{(l)}) \tag{10.25}$$

Each layer captures different levels of abstraction.

Visual aid. Stacked recurrent layers over time.

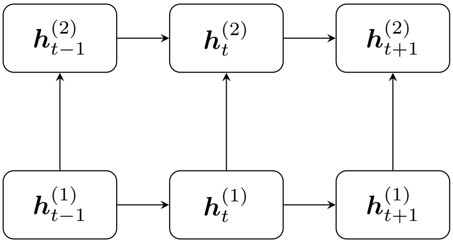
$\boldsymbol{h}_{t-1}^{(2)}$
$\boldsymbol{h}_{t}^{(2)}$
$\boldsymbol{h}_{t+1}^{(2)}$
$\boldsymbol{h}_{t-1}^{(1)}$
$\boldsymbol{h}_{t}^{(1)}$
$\boldsymbol{h}_{t+1}^{(1)}$

Figure 10.6: Deep RNN: multiple recurrent layers stacked over time.

Practical tips: residual connections, layer normalization, and dropout between layers help optimization and generalization.

10.6.3 Teacher Forcing

During training, use ground truth as decoder input (not model's prediction):

- Faster convergence
- Stable training
- May cause exposure bias at test time —a train – test mismatch where the model never learns to recover from its own errors. Mitigations include scheduled sampling and sequence-level training.

10.6.4 Beam Search

For inference, maintain top-k hypotheses:

- Better than greedy decoding
- Trade-off between quality and speed
- Common beam size: 5 – 10; length normalization and coverage penalties are often used in NMT.

Key Takeaways

- **RNNs model temporal dependencies** by maintaining a hidden state that propagates through time; vanilla RNNs struggle with long-term dependencies due to vanishing/exploding gradients [GBC16a].

- **BPTT and truncated BPTT** enable gradient-based training of sequence models; gradient clipping mitigates exploding gradients [GBC16a; RHW86].

- **LSTM and GRU** use gating to preserve and control information flow, greatly improving learning of long-term dependencies [HS97; Cho+14].

- **Seq2seq with attention** overcomes fixed-bottleneck limitations by learning soft alignments and per-step context vectors, enabling effective long-input transduction and reordering [BCB14; Vas+17].

- **Attention scoring** can be additive (Bahdanau) or multiplicative/dot-product; both are trained end-to-end and yield interpretable weights [BCB14; Vas+17].

- **Decoding strategies** matter: greedy is fast but myopic; beam search (with length normalization/coverage) improves quality at added cost.

- **Model variants** expand capacity or context: bidirectional RNNs leverage future context (when available); deep/stacked RNNs add hierarchical abstraction.

- **Training techniques** like teacher forcing speed convergence but induce exposure bias; mitigations include scheduled sampling and sequence-level objectives.

- **Truncation trade-offs** in BPTT balance compute/memory with the ability to learn very long-range dependencies; choose window size to match task needs.

- **Applications** span translation, summarization, QA, captioning, ASR/OCR, and code generation; encoder choices (CNN/RNN/Transformer) and decoding policies should reflect task constraints.

- **Practical considerations** include initialization, clipping, architecture choice (GRU vs. LSTM), attention design, and decoding policy appropriate to data and latency requirements.

10.7 Real World Applications

Recurrent and recursive networks excel at understanding sequences—whether words in sentences, notes in music, or events over time. These capabilities enable numerous practical applications.

10.7.1 Machine Translation

Breaking down language barriers worldwide:

- **Google Translate and similar services:** Sequence models translate between over 100 languages, helping billions of people access information and communicate across language barriers. The models understand context—for example, translating "bank" correctly as a financial institution or riverbank depending on the surrounding words.
- **Real-time conversation translation:** Apps now translate spoken conversations in real-time, enabling tourists to have conversations with locals, business meetings across languages, and international collaboration. The sequence models process speech patterns and convert them to another language while preserving meaning and tone.
- **Document translation:** Businesses use sequence models to translate contracts, user manuals, and websites automatically. While human review remains important, these tools make multilingual business operations feasible and affordable.

10.7.2 Voice Assistants and Speech Recognition

Making human-computer interaction natural:

- **Smartphone assistants:** Siri, Google Assistant, and Alexa use sequence models to understand your voice commands despite accents, background noise, and casual phrasing. These models process sound waves sequentially, recognizing words even when spoken quickly or unclearly.
- **Automated transcription:** Services transcribe meetings, podcasts, and lectures automatically, making content searchable and accessible. Sequence models handle multiple speakers, technical terminology, and varying audio quality—tasks that once required hours of human effort.

- **Accessibility tools:** Voice-to-text applications help people with mobility or vision impairments interact with devices, write documents, and access information independently. These tools become more accurate and responsive through better sequence modeling.

10.7.3 Predictive Text and Content Generation

Enhancing writing and communication:

- **Smart compose in emails:** Email clients predict what you'll type next, suggesting complete sentences based on your writing patterns and the context of your message. This saves time and reduces typing, especially on mobile devices where typing is slower.
- **Code completion in programming:** Development tools like GitHub Copilot suggest code as you type, understanding programming context and patterns. Sequence models trained on billions of lines of code help developers write software faster with fewer bugs.
- **Content moderation:** Social media platforms use sequence models to detect toxic comments, spam, and harmful content in text. The models understand context, slang, and subtle linguistic patterns that indicate problematic content.

10.7.4 Financial Forecasting and Analysis

Understanding temporal patterns in markets:

- **Stock price prediction:** While markets are notoriously difficult to predict, sequence models analyze historical price patterns, trading volumes, and news sentiment to identify trends and inform trading decisions.
- **Fraud detection in transactions:** Banks use sequence models to analyze transaction sequences, identifying unusual patterns that might indicate stolen cards or fraudulent activity. The temporal aspect is crucial—legitimate behavior follows certain patterns over time.
- **Credit risk assessment:** Lenders analyze sequences of financial behaviors (payment histories, spending patterns, income changes) to assess creditworthiness more accurately than snapshot-based approaches.

10.7.5 Why Sequences Matter

The unique value of sequence modeling:

- **Context awareness:** Understanding how earlier elements affect later ones
- **Variable-length handling:** Working with inputs of any length
- **Temporal patterns:** Capturing how things change over time
- **Natural interaction:** Enabling human-like communication with machines

These applications demonstrate how sequence models transform our ability to process language, speech, and time-series data at scale.

10.8 Problems ☒

This section provides exercises to reinforce your understanding of sequence models. Problems are categorized by difficulty and include hints.

10.8.1 Easy Problems (6 problems)

Problem 10.1 (Sequence Types)**.** Classify the following tasks as one-to-one, one-to-many, many-to-one, or many-to-many: sentiment classification, speech recognition, image captioning, machine translation, next-word prediction, time-series forecasting.
Hint: Consider input/output sequence lengths.

Problem 10.2 (Hidden State Intuition)**.** Explain in your own words what the hidden state in an RNN represents and why it is necessary.
Hint: Think of it as a compressed summary of the past.

Problem 10.3 (Exploding vs. Vanishing)**.** Describe the difference between exploding and vanishing gradients in RNNs and one practical mitigation for each.
Hint: Clipping vs. gating/initialization.

Problem 10.4 (Gate Roles)**.** List the roles of the LSTM's forget, input, and output gates.
Hint: Control what to remember, write, and reveal.

Problem 10.5 (Attention Benefit)**.** Why does attention often improve seq2seq performance compared to a fixed context vector?
Hint: Per-step, content-based context.

Problem 10.6 (Bidirectionality)**.** When is a bidirectional RNN appropriate, and when is it not?
Hint: Availability of future context at inference time.

10.8.2 Medium Problems (5 problems)

Problem 10.7 (BPTT Derivative)**.** Derive the recursive relation for $\frac{\partial L}{\partial \boldsymbol{h}_t}$ in a vanilla RNN with $\boldsymbol{h}_t = \sigma(\boldsymbol{W}_{hh}\boldsymbol{h}_{t-1} + \boldsymbol{W}_{xh}\boldsymbol{x}_t + \boldsymbol{b}_h)$.
Hint: Apply the chain rule through time and sum contributions.

Problem 10.8 (Truncated BPTT Trade-off)**.** Discuss the trade-offs in choosing the truncation window k for truncated BPTT.
Hint: Memory/compute vs. long-range dependencies.

Problem 10.9 (GRU vs. LSTM)**.** Compare GRU and LSTM in terms of parameter count, training speed, and ability to model long-term dependencies. When might you prefer each?
Hint: Simplicity vs. expressiveness; dataset size and sequence length.

Problem 10.10 (Teacher Forcing)**.** Explain teacher forcing and describe exposure bias. Propose a mitigation strategy.
Hint: Scheduled sampling; sequence-level training.

Problem 10.11 (Beam Search)**.** Explain how beam size affects quality and speed in sequence decoding.
Hint: Larger beams approximate global search but increase computation.

10.8.3 Hard Problems (5 problems)

Problem 10.12 (Gradient Dynamics)**.** Analyze conditions on $\boldsymbol{W}_{hh}$ and σ' under which gradients vanish or explode in a vanilla RNN. Relate to spectral radius.
Hint: Consider products of Jacobians and eigenvalues.

Problem 10.13 (Attention Mechanisms)**.** Derive additive (Bahdanau) attention scores and compare with multiplicative (dot-product) attention. Discuss computational trade-offs.
Hint: Parameterized MLP vs. dot products; complexity in sequence length.

Problem 10.14 (Alignment Visualization)**.** Propose a method to visualize attention alignments for a translation model and how to interpret them.

Hint: Heatmaps over source – target positions.

Problem 10.15 (Exposure Bias Remedies)**.** Formulate scheduled sampling and discuss its pros/cons. Compare with sequence-level training (e.g., REINFORCE, minimum risk training).

Hint: Train – test mismatch vs. optimization variance.

Problem 10.16 (Design a Seq2Seq System)**.** For speech recognition on 1k-hour dataset, design a system: feature extraction, encoder – decoder choice (GRU/LSTM), attention type, regularization, decoding strategy. Justify choices.

Hint: Compute budget, sequence length, latency constraints, language model integration.

Key Takeaways

Key Takeaways 10

- **RNNs process sequential data** by maintaining hidden states that capture temporal dependencies.
- **LSTMs and GRUs** mitigate vanishing gradients via gating mechanisms that control information flow.
- **Backpropagation through time** (BPTT) computes gradients by unrolling the recurrent computation graph.
- **Attention mechanisms** allow models to focus on relevant parts of the input sequence, improving alignment in seq2seq tasks.
- **Practical challenges** include gradient clipping, teacher forcing, and exposure bias in autoregressive generation.

Problems

Easy

Problem 10.17 (RNN vs Feedforward)**.** Explain why standard feedforward networks are not suitable for sequence modeling tasks. What key capability do RNNs provide?
Hint: Consider variable-length inputs and the need to maintain temporal context.

Problem 10.18 (LSTM Gates)**.** Name the three gates in an LSTM cell and briefly describe the role of each.
Hint: Think about what information needs to be forgotten, what new information to store, and what to output.

Problem 10.19 (Vanishing Gradients)**.** Explain why vanilla RNNs suffer from the vanishing gradient problem when processing long sequences.
Hint: Consider repeated matrix multiplication during backpropagation through time.

Problem 10.20 (Sequence-to-Sequence Tasks)**.** Give three examples of sequence-to-sequence tasks and explain what makes them challenging.
Hint: Consider machine translation, speech recognition, and video captioning.

Medium

Problem 10.21 (BPTT Implementation)**.** Describe how truncated backpropagation through time (BPTT) works. What are the trade-offs compared to full BPTT?
Hint: Consider memory requirements, gradient approximation quality, and the effective temporal window.

Problem 10.22 (Attention Mechanism)**.** Explain the intuition behind attention mechanisms in sequence-to-sequence models. How does attention address the bottleneck of fixed-size context vectors?
Hint: Consider how different parts of the input sequence should influence different parts of the output.

Hard

Problem 10.23 (GRU vs LSTM)**.** Compare GRU (Gated Recurrent Unit) and LSTM architectures mathematically. Derive their update equations and analyse computational complexity.
Hint: Count the number of parameters and operations per cell. GRU has fewer gates.

Problem 10.24 (Bidirectional RNN Gradient)**.** Derive the gradient flow in a bidirectional RNN. Explain why bidirectional RNNs cannot be used for online prediction tasks.
Hint: Consider that backward pass requires seeing the entire future sequence.

Chapter 11

Practical Methodology

This chapter provides practical guidelines for successfully applying deep learning to real-world problems.

Learning Objectives

After studying this chapter, you will be able to:

1. **Scope a deep learning project**: define objectives, constraints, and success metrics.

2. **Design data pipelines**: split datasets, manage leakage, and ensure reproducibility.

3. **Select architectures and baselines**: choose strong baselines and iterate systematically.

4. **Tune hyperparameters**: apply principled search and learning-rate schedules.

5. **Diagnose failures**: use loss curves, ablations, and sanity checks to debug.

6. **Deploy responsibly**: monitor drift, handle distribution shift, and document models.

Intuition

Practical deep learning succeeds when we reduce uncertainty early and iterate quickly. Start with *simple, auditable baselines* to validate data and objectives, then progressively add complexity only when it measurably helps. Prefer experiments that answer the biggest unknowns first (e.g., data quality vs. model capacity). Treat metrics, validation splits, and ablations as your instrumentation layer; they convert intuition into evidence. See also Goodfellow, Bengio, and Courville [GBC16a] for methodology patterns.

11.1 Performance Metrics ☒

11.1.1 Classification Metrics

Robust model evaluation depends on selecting metrics aligned with task requirements and operational costs . Accuracy alone can be misleading under class imbalance ; prefer precision/recall, AUC, PR-AUC, calibration, and cost-sensitive metrics when appropriate Goodfellow, Bengio, and Courville [GBC16a] and Prince [Pri23].

Confusion matrix For binary classification with positive/negative classes, define true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN). The confusion matrix summarizes counts:

	Predicted +	**Predicted –**
Actual +	TP	FN
Actual –	FP	TN

Accuracy accuracy measures overall correctness but can obscure minority-class performance:

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{TN} + \text{FN}}. \tag{11.1}$$

Precision and recall precision and recall quantify quality on the positive class:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}, \tag{11.2}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}. \tag{11.3}$$

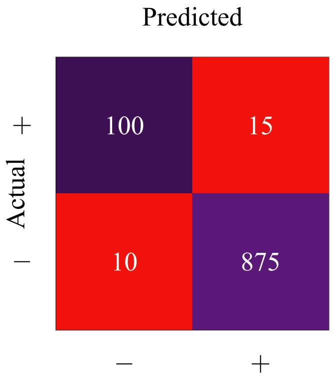

Figure 11.1: Confusion matrix heatmap (example counts). High diagonal values indicate good performance.

F1 score The harmonic mean balances precision and recall:

$$F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}. \tag{11.4}$$

ROC and AUC The ROC curve plots TPR vs. FPR as the decision threshold varies; AUC summarizes ranking quality and is threshold-independent.

$$\text{TPR} = \frac{\text{TP}}{\text{TP} + \text{FN}}, \qquad \text{FPR} = \frac{\text{FP}}{\text{FP} + \text{TN}}. \tag{11.5}$$

Precision–Recall (PR) curve Under heavy class imbalance, the PR curve and average precision (AP) are often more informative than ROC Prince [Pri23].

Calibration A calibrated classifier's predicted probabilities match observed frequencies. Use reliability diagrams and expected calibration error (ECE) . Calibration matters in risk-sensitive applications Goodfellow, Bengio, and Courville [GBC16a].

Visual aids

11.1.2 Regression Metrics

Choose metrics that reflect business loss and robustness to outliers . Mean squared error (MSE) penalizes large errors more heavily than mean absolute error (MAE).

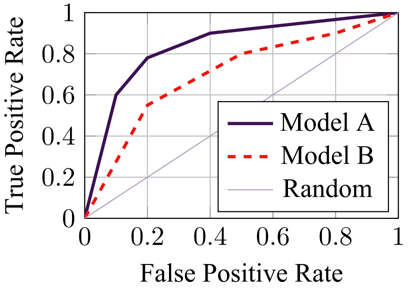

Figure 11.2: ROC curves for two models. Higher AUC indicates better ranking quality.

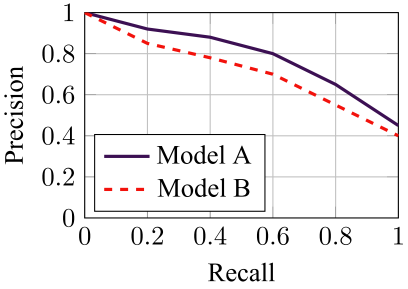

Figure 11.3: Precision–recall curves emphasize performance on the positive class under imbalance.

Root mean squared error (RMSE) is in the original units. Coefficient of determination R^2 measures variance explained.

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2, \quad \mathrm{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|, \quad \mathrm{RMSE} = \sqrt{\mathrm{MSE}}. \tag{11.6}$$

For heavy-tailed noise, consider Huber loss and quantile losses for pinball objectives Prince [Pri23].

11.1.3 NLP and Sequence Metrics

Sequence generation quality is commonly measured by BLEU and ROUGE (n-gram overlap), while language models use *perplexity* (negative log-likelihood in exponential

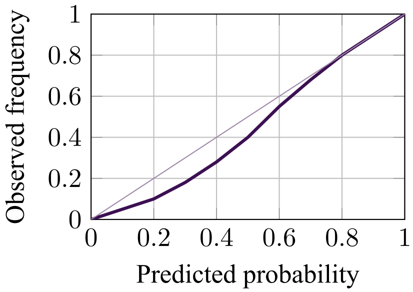

Figure 11.4: Reliability diagram illustrating calibration. The diagonal is perfect calibration.

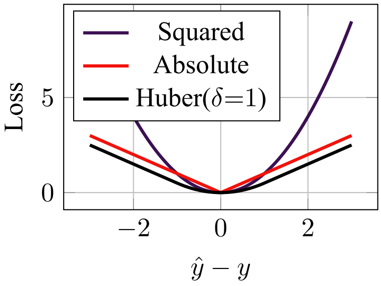

Figure 11.5: Comparison of squared, absolute, and Huber losses.

form) Goodfellow, Bengio, and Courville [GBC16a] and Zhang et al. [Zha+24c]:

$$\text{PPL} = \exp\left(-\frac{1}{N}\sum_{i=1}^{N} \log P(x_i)\right). \tag{11.7}$$

For retrieval and ranking, report mean average precision (mAP), normalized discounted cumulative gain (nDCG), and recall@k.

11.1.4 Worked examples

Imbalanced disease detection In a 1% prevalence setting, a classifier with 99% accuracy can be worthless. Reporting PR-AUC and calibration surfaces early detection quality and absolute risk estimates valued by clinicians Ronneberger, Fischer, and Brox [RFB15].

Threshold selection Optimize thresholds against a cost matrix or utility function (e.g., false negative cost $\gg$ false positive). Plot utility vs. threshold to choose operating points.

Macro vs. micro averaging For multi-class, macro-averaged F1 treats classes equally; micro-averaged F1 weights by support. Choose based on fairness vs. prevalence alignment Prince [Pri23].

11.2 Baseline Models and Debugging ☒

11.2.1 Establishing Baselines

Strong baselines de-risk projects by validating data quality, metrics, and feasibility Goodfellow, Bengio, and Courville [GBC16a] and Prince [Pri23]. Establish multiple baselines and track them as immutable references.
Start with simple baselines:

1. **Random baseline:** Random predictions
2. **Simple heuristics:** Rule-based systems
3. **Classical ML:** Logistic regression, random forests
4. **Simple neural networks:** Small architectures

Compare deep learning improvements against these baselines. Use *data leakage* checks (e.g., time-based splits, patient-level splits) and ensure identical preprocessing across baselines.

11.2.2 Debugging Strategy

Step 1: Overfit a small dataset

- Take 10-100 examples
- Turn off regularization
- If can't overfit, suspect implementation, data, or optimization bugs

Step 2: Check intermediate outputs

- Visualize activations
- Check gradient magnitudes
- Verify loss decreases on training set
- Plot learning-rate vs. loss; test different seeds

Step 3: Diagnose underfitting vs. overfitting

- **Underfitting:** Poor train performance → increase capacity
- **Overfitting:** Good train, poor validation → add regularization

11.2.3 Common Issues

Vanishing/exploding gradients:

- Use batch normalization
- Gradient clipping
- Better initialization
- Consider residual connections

Dead ReLUs:

- Lower learning rate
- Use Leaky ReLU or ELU

Loss not decreasing:

- Check learning rate (too high or too low)
- Verify gradient computation
- Check data preprocessing
- Confirm label alignment and class indexing

11.2.4 Ablation and sanity checks

Perform *ablation studies* to quantify the contribution of each component (augmentation, architecture blocks, regularizers). Use *label shuffling* to verify the pipeline cannot learn when labels are randomized. Train with *frozen features* to isolate head capacity.

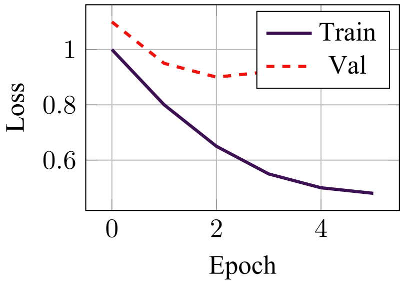

Figure 11.6: Typical overfitting: training loss decreases while validation loss bottoms out and rises.

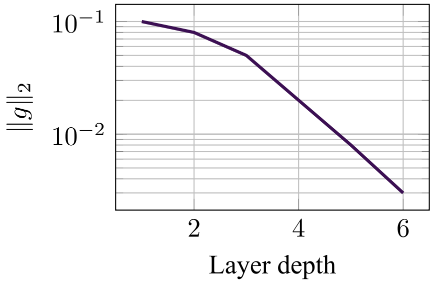

Figure 11.7: Gradient norms vanishing with depth; motivates normalization and residual connections.

11.2.5 Historical notes and references

Debugging by overfitting a tiny subset and systematic ablations has roots in classical ML practice and was emphasized in early deep learning methodology Goodfellow, Bengio, and Courville [GBC16a]. Modern best practices are also surveyed in open textbooks Prince [Pri23] and Zhang et al. [Zha+24b].

11.3 Hyperparameter Tuning ☒

11.3.1 Key Hyperparameters (Priority Order)

Effective tuning prioritizes learning rate, regularization, and capacity before fine details . Treat the validation set as your instrumentation layer and control randomness via fixed seeds Goodfellow, Bengio, and Courville [GBC16a], Prince [Pri23], and Zhang et al. [Zha+24b].

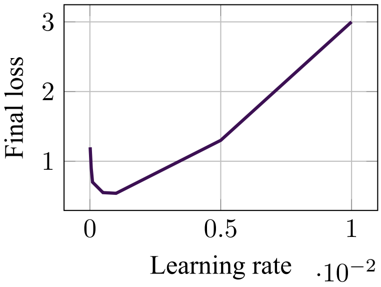

Figure 11.8: Learning-rate sweep to identify a stable training regime.

1. **Learning rate:** Most critical; consider warmup and cosine decay
2. **Network architecture:** Depth/width, normalization, residuals
3. **Batch size:** Affects noise scale and generalization
4. **Regularization:** Weight decay, dropout, label smoothing
5. **Optimizer parameters:** Momentum, β values in Adam

11.3.2 Search Strategies

Manual Search:

- Start with educated guesses
- Adjust based on results
- Time-consuming but insightful

Grid Search:

- Try all combinations from predefined values
- Exhaustive but expensive
- Better for 2-3 hyperparameters

Random Search:

- Sample hyperparameters randomly
- More efficient than grid search

- Better for high-dimensional spaces

Bayesian Optimization:

- Model hyperparameter performance
- Choose next trials intelligently
- More sample-efficient

11.3.3 Best Practices

- Use logarithmic scale for learning rate; sweep $[10^{-5}, 10^{-1}]$
- Vary batch size and adjust learning rate proportionally
- Track results with a consistent random seed and multiple repeats
- Early-stop poor runs; allocate budget adaptively
- Use a fixed validation protocol to avoid leakage
- Retrain with best setting on train+val and report on held-out test

11.3.4 Historical notes

Random search and Bayesian optimization rose to prominence as models and search spaces grew large; they offer better coverage than grid for high-dimensional spaces. Practical DL texts emphasize learning-rate schedules (step, cosine) and warmup for stability Goodfellow, Bengio, and Courville [GBC16a], Prince [Pri23], and Zhang et al. [Zha+24b].

11.4 Data Preparation and Preprocessing ☒

11.4.1 Data Splitting

Train/Validation/Test split:

- Training: 60-80%
- Validation: 10-20%
- Test: 10-20%

Cross-validation: For small datasets

- k-fold cross-validation
- Stratified splits for imbalanced data

11.4.2 Normalization

Min-Max Scaling:

$$x' = \frac{x - x_{\text{min}}}{x_{\text{max}} - x_{\text{min}}} \tag{11.8}$$

Standardization (Z-score):

$$x' = \frac{x - \mu}{\sigma} \tag{11.9}$$

Always compute statistics on training set only!

11.4.3 Handling Imbalanced Data

- **Oversampling:** Duplicate minority class examples
- **Undersampling:** Remove majority class examples
- **SMOTE:** Synthetic minority oversampling
- **Class weights:** Penalize errors on minority class more
- **Focal loss:** Focus on hard examples

11.4.4 Data Augmentation

Generate additional training examples through transformations (see Chapter 7). For images: flips, crops, color jitter; for text: back-translation; for audio: time stretch, noise. Calibrate augmentation strength to avoid distribution shift Prince [Pri23].

11.4.5 Visual aids

11.4.6 Historical notes

Careful dataset design (train/val/test segregation, leakage prevention) has long underpinned reliable evaluation in ML and remains essential at scale in deep learning Bishop [Bis06] and Goodfellow, Bengio, and Courville [GBC16a].

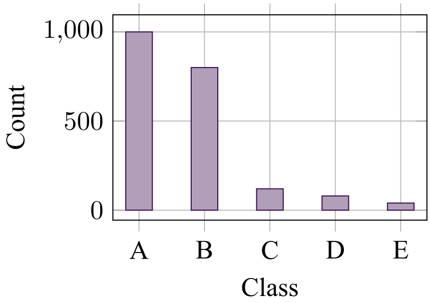

Figure 11.9: Imbalanced dataset example motivating class weights or resampling.

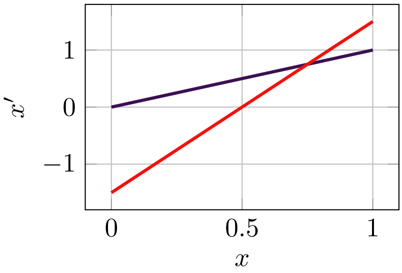

Figure 11.10: Min-max scaling (purple) vs. standardization (red) schematic.

11.5 Production Considerations ☒

11.5.1 Model Deployment

- Model compression (pruning, quantization) to meet latency/size budgets
- Model serving infrastructure (A/B testing, canary deploys)
- Latency budgets and tail performance (p95/p99)
- Batch vs. online inference and feature freshness

11.5.2 Monitoring

Track in production:

- Prediction distribution shifts (covariate/label shift)
- Model performance metrics and calibration drift

- System latency and throughput; autoscaling behavior
- Error analysis with human-in-the-loop review

11.5.3 Iterative Improvement

1. Deploy initial model
2. Monitor performance
3. Collect more data
4. Retrain and improve
5. A/B test new models

11.5.4 Visual aids

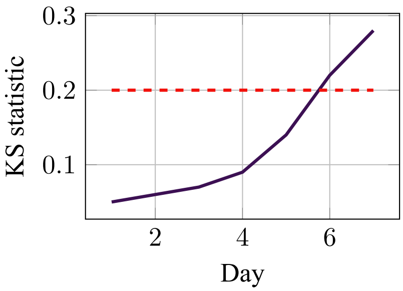

Figure 11.11: Simple drift monitor: KS statistic over time with an alert threshold.

11.5.5 Applications and context

Compression and low-latency serving are crucial in mobile vision, speech on device, and recommender systems; calibration and post-deployment monitoring are critical in healthcare and finance Ronneberger, Fischer, and Brox [RFB15] and Prince [Pri23].

11.6 Real World Applications

Practical methodology—the systematic approach to designing, training, and deploying deep learning systems—is what separates successful real-world projects from academic experiments.

11.6.1 Healthcare Diagnostic System Deployment

Bringing AI from lab to clinic:

- **FDA-approved medical imaging systems:** Companies developing AI diagnostic tools must follow rigorous methodologies: careful dataset collection from diverse hospitals, systematic validation on held-out test sets, extensive clinical trials, and continuous monitoring post-deployment. A stroke detection system, for example, must work reliably across different scanners, patient populations, and hospital settings before doctors trust it with patient care.

- **Handling data quality issues:** Real medical data is messy—images have artifacts, labels contain errors, and rare diseases are underrepresented. Practical methodology includes data cleaning procedures, handling class imbalance, and establishing confidence thresholds for when the system should defer to human experts.

- **Continuous learning and monitoring:** Once deployed, medical AI systems need ongoing validation. Methodology includes establishing monitoring dashboards, detecting distribution shift (when patient populations change), and protocols for updating models without disrupting clinical workflows.

11.6.2 Recommendation System Development

Building and maintaining large-scale personalization:

- **A/B testing and evaluation:** When Netflix develops new recommendation algorithms, they don't just optimize offline metrics. Practical methodology involves carefully designed A/B tests with real users, balancing multiple objectives (user engagement, diversity, content discovery, business goals), and understanding long-term effects beyond immediate clicks.

- **Cold start problem:** New users have no history, and new items have no ratings. Practical methodology addresses this through strategic initialization, hybrid approaches combining content features with collaborative filtering, and active learning to quickly gather useful information.

- **Production infrastructure:** Serving recommendations to millions of users simultaneously requires careful system design. Methodology includes choosing appropriate model architectures that balance accuracy with inference speed, caching strategies, and gradual rollouts to detect problems early.

11.6.3 Autonomous Vehicle Development

The most safety-critical deep learning application:

- **Simulation and testing methodology:** Self-driving cars must handle rare but critical scenarios (a child running into the street). Companies use systematic methodologies combining real-world data collection, photorealistic simulation of dangerous scenarios, and extensive closed-track testing before public road trials.
- **Failure analysis and iteration:** When test vehicles make mistakes, teams follow rigorous procedures to understand root causes, reproduce issues in simulation, develop fixes, and validate improvements. This includes systematic logging of all sensor data and decisions for later analysis.
- **Multi-stage validation:** Models progress through increasingly realistic testing: simulation, closed tracks, controlled public roads, then broader deployment. Each stage has specific success criteria and methodologies for objective evaluation.

11.6.4 Key Methodological Principles

What makes real-world projects succeed:

- **Start simple:** Baseline models first, then increase complexity as needed
- **Measure what matters:** Align metrics with actual business or user goals
- **Understand your data:** Invest time in data exploration and cleaning
- **Iterate systematically:** Change one thing at a time to understand impact
- **Plan for production:** Consider deployment constraints from the beginning
- **Monitor continuously:** Real-world conditions change; models must adapt

These examples show that methodology—the ”how” of deep learning—is just as important as the ”what” when building systems that work reliably in practice.

Key Takeaways

Key Takeaways 11

- **Start simple**: Establish a reliable baseline to validate data, metrics, and training code.
- **Measure relentlessly**: Use clear validation splits, confidence intervals, and learning curves.
- **Ablate to learn**: Prefer small, controlled changes to isolate causal effects.
- **Prioritise data**: Label quality, coverage, and augmentation often beat model complexity.
- **Tune methodically**: Track hyperparameters, seeds, and environments for reproducibility.
- **Deploy with monitoring**: Watch for drift, performance decay, and fairness regressions; plan periodic re-training.

Problems

Easy

Problem 11.1 (Define Success Metrics)**.** You are building a classifier for defect detection. Propose suitable metrics beyond accuracy and justify validation splits.
Hint: Consider precision/recall, AUROC vs. AUPRC under class imbalance, and stratified splits.

Problem 11.2 (Sanity Checks)**.** List three quick sanity checks to run before large-scale training and explain expected outcomes.
Hint: Overfit a tiny subset; randomise labels; train with shuffled pixels.

Problem 11.3 (Baseline First)**.** Explain why a strong non-deep baseline can accelerate iteration on a deep model.
Hint: Separates data/metric issues from model capacity; provides performance floor.

Problem 11.4 (Data Leakage)**.** Define data leakage and give two concrete examples.
Hint: Temporal leakage; using normalised stats computed on the full dataset.

Medium

Problem 11.5 (Hyperparameter Search Budget)**.** Given budget for 30 runs, propose an allocation between exploration (random search) and exploitation (local search). Defend your choice.
Hint: Start broad (e.g., 20 random), then refine top configurations (e.g., 10 local).

Problem 11.6 (Early Stopping vs. Schedules)**.** Compare early stopping with cosine decay schedules under limited training budget.
Hint: Consider variance, bias, and checkpoint selection.

Hard

Problem 11.7 (Confidence Intervals)**.** Derive a 95% Wilson interval for a classifier with n samples and accuracy $\hat{p}$.
Hint: Use $\frac{\hat{p}+z^2/(2n)\pm z\sqrt{\frac{\hat{p}(1-\hat{p})}{n}+\frac{z^2}{4n^2}}}{1+z^2/n}$ with $z \approx 1.96$.

Problem 11.8 (Causal Confounding)**.** Your model uses a spurious feature. Propose an experimental protocol to detect and mitigate it.
Hint: Counterfactual augmentation, environment splitting, invariant risk minimisation.

Chapter 12

Applications

This chapter showcases deep learning applications across various domains, demonstrating the breadth and impact of the field.

Learning Objectives

After studying this chapter, you will be able to:

1. Identify opportunities where deep learning provides value in vision, language, speech, and tabular domains.
2. Select appropriate model families and data representations for each application area.
3. Design evaluation protocols and metrics suited to each task type.
4. Recognize common failure modes and deployment considerations across applications.

Intuition

Applications differ not only in architectures but in *data assumptions and metrics*. A robust recipe is: clarify the task and loss, understand the data distribution and constraints, then choose the simplest model that can plausibly meet the requirements before scaling up.

12.1 Computer Vision Applications 💫

12.1.1 Image Classification

Assign labels to images; modern benchmarks like ImageNet catalyzed deep CNN adoption Krizhevsky, Sutskever, and Hinton [KSH12], He et al. [He+16], Goodfellow, Bengio, and Courville [GBC16a], and Prince [Pri23].

- **ImageNet:** 1000-class object recognition
- **Fine-grained classification:** Bird species, car models
- **Medical imaging:** Disease classification from X-rays, CT scans

Architecture: CNN backbone (e.g., ResNet) + classification head; transfer learning from pretrained weights is standard practice.

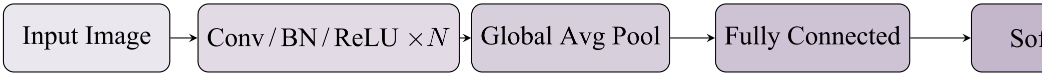

Figure 12.1: Typical CNN classification pipeline.

12.1.2 Object Detection

Locate and classify objects in images; real-time variants (YOLO) emphasize speed, while two-stage models (Faster R-CNN) emphasize accuracy.

- **Autonomous driving:** Pedestrians, vehicles, traffic signs
- **Surveillance:** Person detection and tracking
- **Retail:** Product recognition

Methods: YOLO, Faster R-CNN, RetinaNet; focal loss handles class imbalance in dense detectors.

12.1.3 Semantic Segmentation

Classify every pixel:

- **Autonomous driving:** Road, sidewalk, vehicle segmentation
- **Medical imaging:** Tumor segmentation, organ delineation

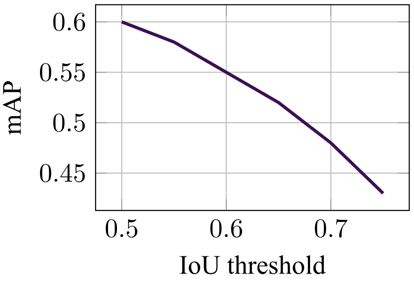

Figure 12.2: Detector performance (mAP) vs. IoU threshold schematic.

- **Satellite imagery:** Land use classification

Architectures: U-Net, DeepLab, Mask R-CNN

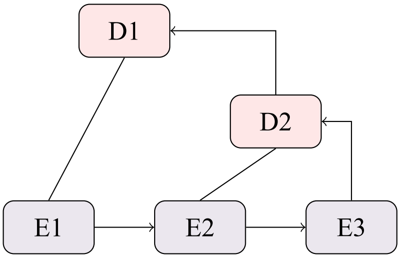

Figure 12.3: U-Net style encoder-decoder with skip connections.

12.1.4 Face Recognition

Identify or verify individuals:

- Security and access control
- Photo organization
- Payment authentication

Approach: Face detection + embedding (FaceNet, ArcFace) + similarity matching; report ROC/PR, false accept/reject at target FAR/FRR.

12.1.5 Image Generation and Manipulation

- **Style transfer:** Apply artistic styles

- **Super-resolution:** Enhance image quality
- **Inpainting:** Fill missing regions
- **Deepfakes:** Face swapping (ethical concerns)

12.1.6 Historical context and references

Modern CV breakthroughs stem from CNNs popularized by ImageNet-scale training Krizhevsky, Sutskever, and Hinton [KSH12], deeper residual networks He et al. [He+16], and specialized architectures for segmentation Ronneberger, Fischer, and Brox [RFB15]. See Goodfellow, Bengio, and Courville [GBC16a] and Prince [Pri23] for broader context.

12.2 Natural Language Processing 💫

12.2.1 Text Classification

Categorize text documents using pretrained transformers and task heads; fine-tuning is data-efficient and standard Devlin et al. [Dev+18], Prince [Pri23], and Zhang et al. [Zha+24a].

- **Sentiment analysis:** Positive/negative reviews
- **Spam detection:** Email filtering
- **Topic classification:** News categorization

Models: BERT, RoBERTa, DistilBERT; report accuracy, F1, calibration for risk-sensitive domains.

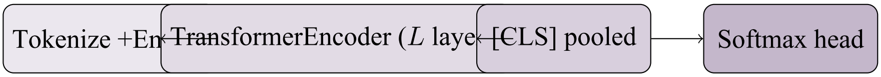

Figure 12.4: Transformer fine-tuning for text classification.

12.2.2 Machine Translation

Translate between languages; attention and Transformers supplanted earlier seq2seq RNNs Bahdanau, Cho, and Bengio [BCB14], Vaswani et al. [Vas+17], Goodfellow, Bengio, and Courville [GBC16a], and Zhang et al. [Zha+24a].

- Google Translate, DeepL
- Sequence-to-sequence with attention
- Transformer models

Architecture: Encoder-decoder transformers with subword tokenization; report BLEU/chrF and human eval.

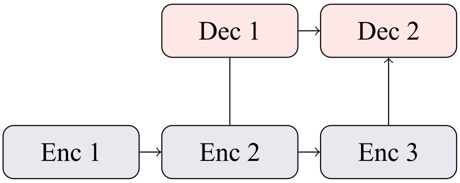

Figure 12.5: Encoder – decoder Transformer schematic with cross-attention.

12.2.3 Question Answering

Answer questions based on context:

- **Extractive QA:** Find answer span in text (SQuAD)
- **Open-domain QA:** Answer from large corpora
- **Visual QA:** Answer questions about images

12.2.4 Language Models and Text Generation

Generate coherent text with large language models (LLMs) Radford et al. [Rad+19] and Prince [Pri23]. Evaluate with task-specific metrics and human preferences; ensure safety and grounding.

- GPT models for general text generation
- Code generation (GitHub Copilot)
- Chatbots and conversational AI
- Content creation

12.2.5 Named Entity Recognition

Extract entities from text; use BIO tagging, CRF heads on top of encoders; report token/entity F1.

- Person, organization, location names
- Dates, quantities, technical terms
- Applications in information extraction

12.2.6 Historical context and references

Transformers Vaswani et al. [Vas+17] and pretrained models Devlin et al. [Dev+18] and Radford et al. [Rad+19] dramatically improved NLP performance and data efficiency. See Goodfellow, Bengio, and Courville [GBC16a], Prince [Pri23], and Zhang et al. [Zha+24a] for broader context and tutorials.

12.3 Speech Recognition and Synthesis 💫

12.3.1 Automatic Speech Recognition (ASR)

Convert speech to text using acoustic front-ends and sequence models; self-supervised pretraining (e.g., wav2vec) reduces labeled data needs Prince [Pri23].

- Virtual assistants (Siri, Alexa, Google Assistant)
- Transcription services
- Voice commands

Architectures: CTC-based models, Listen-Attend-Spell, Transducer models, wav2vec.

12.3.2 Text-to-Speech (TTS)

Generate natural-sounding speech with neural vocoders; evaluate with MOS and intelligibility measures.

- WaveNet, Tacotron
- Voice cloning
- Accessibility tools

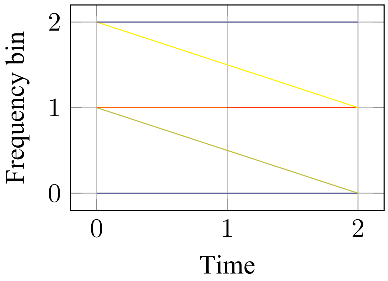

Figure 12.6: Schematic spectrogram input for ASR.

12.3.3 Speaker Recognition

Identify speakers:

- Voice biometrics
- Speaker diarization (who spoke when)

12.3.4 Historical context and references

End-to-end ASR and neural TTS displaced classical pipelines by learning representations directly from data. Self-supervised audio pretraining further improved robustness Prince [Pri23].

12.4 Healthcare and Medical Imaging 💫

12.4.1 Medical Image Analysis

Disease detection:

- Cancer detection in mammograms, CT scans
- Diabetic retinopathy from retinal images
- Pneumonia detection from chest X-rays

Segmentation:

- Tumor boundary delineation
- Organ segmentation for surgical planning

12.4.2 Drug Discovery

- Predicting molecular properties
- Protein structure prediction (AlphaFold)
- Drug-target interaction prediction

12.4.3 Clinical Decision Support

- Diagnosis assistance
- Treatment recommendation
- Risk prediction (readmission, mortality)

12.4.4 Genomics

12.4.5 Historical context and references

CNNs enabled breakthroughs in segmentation and classification, notably U-Net in biomedical imaging Ronneberger, Fischer, and Brox [RFB15]. Clinical deployment emphasizes calibration and monitoring Prince [Pri23].

- DNA sequence analysis
- Variant calling
- Gene expression prediction

12.5 Reinforcement Learning Applications 💫

12.5.1 Game Playing

Superhuman performance using deep reinforcement learning and self-play Silver et al. [Sil+16] and Prince [Pri23]:

- **AlphaGo:** Defeated world champion in Go
- **AlphaZero:** Mastered chess, shogi, and Go
- **OpenAI Five:** Dota 2
- **AlphaStar:** StarCraft II

12.5.2 Robotics

- **Manipulation:** Grasping, assembly
- **Navigation:** Autonomous movement
- **Locomotion:** Walking, running

12.5.3 Autonomous Vehicles

- Path planning and decision making
- Combined with perception (CV)
- Safety-critical systems

12.5.4 Recommendation Systems

- Netflix, YouTube content recommendations
- E-commerce product suggestions
- Personalized news feeds

12.5.5 Resource Management

12.5.6 Historical context and references

Self-play with deep networks and tree search led to landmark results in games Silver et al. [Sil+16]. RL also powers practical optimization in data centers and networks Prince [Pri23].

- Data center cooling optimization (DeepMind)
- Traffic light control
- Energy grid optimization

12.6 Other Applications 💫

12.6.1 Finance

- Algorithmic trading
- Fraud detection
- Credit risk assessment
- Market prediction

12.6.2 Scientific Research

- **Physics:** Particle classification, gravitational wave detection
- **Climate science:** Weather prediction, climate modeling
- **Astronomy:** Galaxy classification, exoplanet detection

12.6.3 Agriculture

- Crop disease detection
- Yield prediction
- Precision agriculture

12.6.4 Manufacturing

12.6.5 References

For domain-specific overviews, see Prince [Pri23] (applications survey).

- Quality control and defect detection
- Predictive maintenance
- Supply chain optimization

12.7 Real World Applications

While this chapter already discusses many applications, this section provides additional concrete examples of how deep learning technologies integrate into everyday products and services.

12.7.1 Smart Home Devices

Deep learning makes homes more intelligent and responsive:

- **Smart speakers with voice control:** Devices like Amazon Echo and Google Home use multiple deep learning models working together: speech recognition converts your voice to text, natural language understanding interprets your intent, and text-to-speech responds naturally. These run locally for privacy while cloud models handle complex queries.
- **Energy optimization:** Smart thermostats learn your schedule and preferences, using sequence models to predict when you'll be home and adjusting temperature efficiently. This saves energy while maintaining comfort, learning patterns like ”weekday mornings” versus ”weekend sleep-ins.”
- **Security cameras with intelligent alerts:** Instead of alerting you to every motion, smart cameras use computer vision to distinguish between family members, delivery people, and potential intruders. They can even recognize familiar faces and alert you only to unexpected visitors.

12.7.2 Entertainment and Media

Personalized content discovery and creation:

- **Music streaming personalization:** Spotify's Discover Weekly and similar features use deep learning to analyze your listening habits, finding new music matching your taste. Models consider factors like song characteristics, listening context (workout versus relaxation), and discovery patterns of similar users.
- **Video game AI:** Modern games use deep learning for realistic non-player character behavior, procedural content generation, and adaptive difficulty. Racing games learn from your driving style to provide challenging but fair opponents; strategy games develop tactics matching your skill level.

- **Content creation tools:** Apps that remove backgrounds from photos, enhance low-light images, or create artistic filters all use deep learning. Tools like DALL-E and Midjourney enable anyone to create visual content from text descriptions, democratizing digital art creation.

12.7.3 Agriculture and Food Production

Feeding the world more efficiently:

- **Crop disease detection:** Farmers photograph plants with smartphones; deep learning identifies diseases early when treatment is most effective. This reduces crop losses and chemical use, as farmers apply pesticides only where needed rather than entire fields.

- **Autonomous farming equipment:** Tractors and harvesters using computer vision navigate fields autonomously, optimize planting patterns, and identify weeds for precision herbicide application. This increases yields while reducing environmental impact and labor costs.

- **Food quality inspection:** Processing facilities use vision systems to inspect products at superhuman speeds—checking eggs for cracks, sorting fruits by ripeness, detecting foreign objects in packaged foods. This improves food safety while reducing waste from over-cautious human inspection.

12.7.4 Integration and Impact

What makes these applications successful:

- **Seamless integration:** AI works invisibly, enhancing rather than complicating user experience

- **Continuous improvement:** Systems learn from usage, becoming more personalized over time

- **Practical constraints:** Solutions balance accuracy, speed, cost, and privacy appropriately

- **Accessibility:** Advanced AI capabilities available through consumer devices

These examples illustrate how deep learning pervades modern life, often working behind the scenes to make products smarter, more efficient, and more personalized.

12.8 Ethical Considerations 💫

Deep learning applications raise important concerns:

- **Bias and fairness:** Models may perpetuate societal biases
- **Privacy:** Data collection and usage concerns
- **Transparency:** ”Black box” nature of deep models
- **Security:** Adversarial attacks and model manipulation
- **Job displacement:** Automation impact on employment
- **Environmental impact:** Energy consumption of large models

Responsible AI development requires addressing these challenges. Incorporate documentation (model cards, datasheets), human oversight, and continuous monitoring; consider environmental costs during model selection.

Key Takeaways

Key Takeaways 12

- **Match model to modality**: Use architectures aligned with data structure and constraints.
- **Task-first thinking**: Metrics and deployment constraints shape design more than model fashion.
- **Data pipelines matter**: Label quality, coverage, and drift monitoring are critical.
- **Simplicity scales**: Strong baselines and incremental complexity speed up real projects.

Problems

Easy

Problem 12.1 (Vision Metrics). Choose between accuracy, AUROC, and mAP for object detection. Justify.
Hint: Class imbalance and localisation vs. classification.

Problem 12.2 (NLP Tokenisation). Explain implications of BPE vs. WordPiece for rare words.
Hint: Subword frequency, OOV handling, sequence length.

Problem 12.3 (Tabular Baselines). Why can tree ensembles outperform deep nets on small tabular datasets?
Hint: Inductive bias, feature interactions, sample efficiency.

Problem 12.4 (Data Privacy). List two privacy risks when deploying medical models and mitigations.
Hint: Re-identification, membership inference; anonymisation, DP.

Medium

Problem 12.5 (Dataset Shift). Design a test to detect covariate shift between training and production.
Hint: Train a domain classifier; compare feature distributions.

Problem 12.6 (Active Learning). Propose an acquisition strategy for labelling a limited budget.
Hint: Uncertainty sampling, diversity, class balance.

Hard

Problem 12.7 (Cost-aware Serving). Formalise an objective trading accuracy vs. compute cost and latency.
Hint: Multi-objective optimisation; constrained maximisation.

Problem 12.8 (Ethical Deployment). Specify post-deployment monitoring and escalation for high-stakes tasks.
Hint: Thresholds, audits, rollback, human oversight.

Part III

Deep Learning Research

Chapter 13

Linear Factor Models

This chapter introduces probabilistic models with linear structure, which form the foundation for many unsupervised learning methods.

Learning Objectives

After studying this chapter, you will be able to:

1. Explain probabilistic PCA and factor analysis and their relationships to PCA.
2. Derive EM updates for latent variable models with linear-Gaussian structure.
3. Compare identifiability and rotation issues in factor models.
4. Connect linear latent models to modern representation learning.

Intuition

Linear factor models posit a small set of hidden sources generating observed data through linear mixing plus noise. The learning problem is to recover those hidden coordinates that best explain variance while respecting uncertainty.

13.1 Probabilistic PCA ☒

13.1.1 Principal Component Analysis Review

PCA finds orthogonal directions of maximum variance:

$$\boldsymbol{z} = \boldsymbol{W}^\top(\boldsymbol{x} - \boldsymbol{\mu}) \tag{13.1}$$

where $\boldsymbol{W}$ contains principal components (eigenvectors of covariance matrix).

13.1.2 Probabilistic Formulation

Model observations as:

$$\boldsymbol{z} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I}) \tag{13.2}$$

$$\boldsymbol{x}|\boldsymbol{z} \sim \mathcal{N}(\boldsymbol{W}\boldsymbol{z} + \boldsymbol{\mu}, \sigma^2\boldsymbol{I}) \tag{13.3}$$

Marginalizing over $\boldsymbol{z}$:

$$\boldsymbol{x} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{W}\boldsymbol{W}^\top + \sigma^2\boldsymbol{I}) \tag{13.4}$$

13.1.3 Learning

Maximize likelihood using EM algorithm:

- **E-step:** Compute $p(\boldsymbol{z}|\boldsymbol{x})$

- **M-step:** Update $\boldsymbol{W}$, $\boldsymbol{\mu}$, σ^2

As $\sigma^2 \to 0$, recovers standard PCA.

13.1.4 Historical context and references

The probabilistic formulation connects PCA with latent variable models and enables principled handling of noise and missing data Bishop [Bis06] and Goodfellow, Bengio, and Courville [GBC16a].

13.2 Factor Analysis ☒

Similar to probabilistic PCA but with diagonal noise covariance:

$$\boldsymbol{x}|\boldsymbol{z} \sim \mathcal{N}(\boldsymbol{W}\boldsymbol{z} + \boldsymbol{\mu}, \boldsymbol{\Psi}) \tag{13.5}$$

where $\boldsymbol{\Psi}$ is diagonal. Each observed dimension has its own noise variance.

Applications: Psychology, social sciences, finance

13.2.1 Learning via EM

EM alternates between inferring posteriors over factors and updating loadings $\boldsymbol{W}$ and noise $\boldsymbol{\Psi}$. Diagonal noise permits modeling idiosyncratic measurement error per dimension Bishop [Bis06].

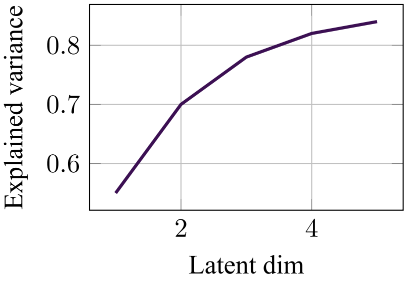

Figure 13.1: Explained variance as a function of latent dimensionality (illustrative).

13.3 Independent Component Analysis ☒

13.3.1 Objective

Find independent sources from linear mixtures:

$$\boldsymbol{x} = \boldsymbol{A}\boldsymbol{s} \tag{13.6}$$

where $\boldsymbol{s}$ contains independent sources.

13.3.2 Non-Gaussianity

ICA exploits that independent signals are typically non-Gaussian.
Applications:

- Blind source separation (cocktail party problem)
- Signal processing
- Feature extraction

13.4 Sparse Coding ☒

Learn overcomplete dictionary where data has sparse representation:

$$\min_{\boldsymbol{D},\boldsymbol{z}} \|\boldsymbol{x} - \boldsymbol{D}\boldsymbol{z}\|^2 + \lambda\|\boldsymbol{z}\|_1 \tag{13.7}$$

Applications:

- Image denoising
- Feature learning
- Compression

13.4.1 Optimization and interpretation

The ℓ_1 penalty promotes sparsity, yielding parts-based representations and robust denoising. Alternating minimization over dictionary $\boldsymbol{D}$ and codes $\boldsymbol{z}$ is common; convolutional variants are used in images Goodfellow, Bengio, and Courville [GBC16a].

13.5 Real World Applications

Linear factor models, including PCA, ICA, and sparse coding, provide interpretable representations of complex data. These techniques underpin many practical systems for data compression, signal processing, and feature extraction.

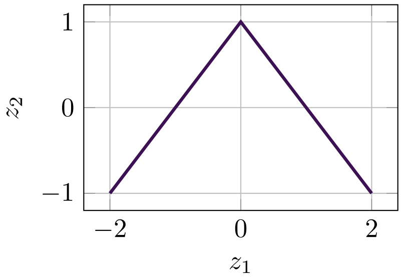

Figure 13.2: Schematic illustrating ℓ_1-induced sparsity geometry.

13.5.1 Facial Recognition Systems

Efficient and robust face identification:

- **Eigenfaces for face recognition:** One of the earliest successful face recognition systems used PCA to represent faces efficiently. Each face is expressed as a weighted combination of ”eigenfaces” (principal components). This reduces storage requirements dramatically—instead of storing full images, systems store just a few dozen numbers per person while maintaining recognition accuracy.

- **Robust to lighting and expression:** Factor models capture the most important variations in face appearance (identity) while being less sensitive to less important variations (lighting, expression). This makes recognition work under different conditions without requiring massive datasets of each person.

- **Privacy-preserving representations:** The compressed representations from factor models can enable face recognition without storing actual face images, providing better privacy protection. The low-dimensional codes contain enough information for matching but can’t easily be reversed to reconstruct recognizable faces.

13.5.2 Audio Signal Processing

Extracting meaning from sound:

- **Music analysis and recommendation:** Spotify and similar services use factor models to decompose songs into latent features (mood, genre, tempo, instrumentation). These compact representations enable efficient similarity search across millions of songs. When you like a song, the system finds others with similar factor patterns.

- **Noise reduction in hearing aids:** Modern hearing aids use sparse coding to separate speech from background noise. The factor model learns to represent speech efficiently with few active components while requiring many more components for noise. This distinction enables selective amplification of speech while suppressing noise.
- **Source separation:** Isolating individual instruments in music recordings or separating overlapping speakers in recordings uses independent component analysis (ICA). This enables remixing old recordings, improving audio quality, and creating karaoke tracks from normal songs.

13.5.3 Anomaly Detection in Systems

Finding unusual patterns in complex data:

- **Network intrusion detection:** Cybersecurity systems use factor models to represent normal network traffic patterns compactly. Unusual activities (potential attacks) don't fit well into this low-dimensional representation, triggering alerts. This approach detects novel attacks without explicitly programming rules for every possible threat.
- **Manufacturing quality control:** Production lines use factor models to analyze sensor data from equipment. Normal operations cluster in low-dimensional space; deviations indicate problems like tool wear, calibration drift, or defects. Early detection prevents defective products and costly equipment damage.
- **Healthcare monitoring:** Wearable devices compress continuous health metrics (heart rate, activity, sleep patterns) into factor representations. Doctors can spot concerning trends without examining raw data streams, and anomaly detection alerts patients to unusual patterns warranting attention.

13.5.4 Practical Advantages

Why factor models remain valuable:

- **Interpretability:** Components often correspond to meaningful concepts

- **Efficiency:** Dramatically reduce data storage and transmission costs
- **Generalization:** Capture essential patterns while ignoring noise
- **Foundation:** Serve as building blocks for more complex deep learning systems

These applications demonstrate that relatively simple factor models continue to provide practical value, either standalone or as components within larger deep learning systems.

Key Takeaways

Key Takeaways 13

- **Linear-Gaussian latent models** provide probabilistic PCA and factor analysis with uncertainty estimates.
- **EM algorithm** alternates inference of latents and parameter updates, exploiting conjugacy.
- **Identifiability** requires fixing rotations/scales; solutions are not unique without conventions.
- **Bridge to deep models**: Linear factors motivate nonlinear representation learning and VAEs.

Problems

Easy

Problem 13.1 (PPCA vs PCA)**.** Contrast PCA and probabilistic PCA in assumptions and outputs.
Hint: Deterministic vs. probabilistic view; noise model; likelihood.

Problem 13.2 (Dimensionality Choice)**.** List two heuristics to select latent dimensionality.
Hint: Explained variance, information criteria.

Problem 13.3 (Gaussian Conditionals)**.** Recall the conditional of a joint Gaussian and its role in E-steps.
Hint: Use block-partitioned mean and covariance formulas.

Problem 13.4 (Rotation Ambiguity)**.** Explain why factor loadings are identifiable only up to rotation.
Hint: Orthogonal transforms preserve latent covariance.

Medium

Problem 13.5 (PPCA Likelihood)**.** Derive the log-likelihood of PPCA and the M-step for σ^2.
Hint: Marginalise latents; differentiate w.r.t. variance.

Problem 13.6 (EM for FA)**.** Write the E and M steps for Factor Analysis with diagonal noise.
Hint: Use expected sufficient statistics of latents.

Hard

Problem 13.7 (Equivalence of PPCA Solution)**.** Show that the MLE loading matrix spans the top-k eigenvectors of the sample covariance.
Hint: Use spectral decomposition of covariance.

Problem 13.8 (Nonlinear Extension)**.** Sketch how to generalise linear factor models to VAEs.
Hint: Replace linear-Gaussian with neural encoder/decoder.

Chapter 14

Autoencoders

This chapter explores autoencoders, neural networks designed for unsupervised learning through data reconstruction.

Learning Objectives

After studying this chapter, you will be able to:

1. Describe the autoencoder framework and common variants (denoising, sparse, contractive).
2. Explain the role of bottlenecks and regularization in learning useful representations.
3. Implement training objectives and evaluate reconstruction vs. downstream utility.
4. Understand the connection between autoencoders and generative models.

Intuition

Autoencoders compress input data into a compact code that retains salient information for reconstruction. Constraining capacity (via architecture or penalties) encourages the model to discard noise and redundancies, surfacing structure that transfers to other tasks.

14.1 Undercomplete Autoencoders ⊠

14.1.1 Architecture

An autoencoder consists of:

- **Encoder:** $\boldsymbol{h} = f(\boldsymbol{x})$ maps input to latent representation
- **Decoder:** $\hat{\boldsymbol{x}} = g(\boldsymbol{h})$ reconstructs from latent code

14.1.2 Training Objective

Minimize reconstruction error:

$$L = \|\boldsymbol{x} - g(f(\boldsymbol{x}))\|^2 \tag{14.1}$$

or more generally:

$$L = -\log p(\boldsymbol{x}|g(f(\boldsymbol{x}))) \tag{14.2}$$

14.1.3 Undercomplete Constraint

If $\dim(\boldsymbol{h}) < \dim(\boldsymbol{x})$, the autoencoder learns compressed representation. Acts as dimensionality reduction (similar to PCA but non-linear).

14.2 Regularized Autoencoders ⊠

14.2.1 Sparse Autoencoders

Add sparsity penalty on hidden activations:

$$L = \|\boldsymbol{x} - \hat{\boldsymbol{x}}\|^2 + \lambda \sum_j |h_j| \tag{14.3}$$

Encourages learning of sparse, interpretable features.

14.2.2 Denoising Autoencoders (DAE)

Train to reconstruct clean input from corrupted version:

1. Corrupt input: $\tilde{\boldsymbol{x}} \sim q(\tilde{\boldsymbol{x}}|\boldsymbol{x})$

2. Encode corrupted input: $\boldsymbol{h} = f(\tilde{\boldsymbol{x}})$
3. Decode and reconstruct: $\hat{\boldsymbol{x}} = g(\boldsymbol{h})$
4. Minimize: $L = \|\boldsymbol{x} - \hat{\boldsymbol{x}}\|^2$

Corruption types:

- Additive Gaussian noise
- Masking (randomly set inputs to zero)
- Salt-and-pepper noise

Learns robust representations.

14.2.3 Contractive Autoencoders (CAE)

Add penalty on Jacobian of encoder:

$$L = \|\boldsymbol{x} - \hat{\boldsymbol{x}}\|^2 + \lambda \left\| \frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}} \right\|_F^2 \tag{14.4}$$

Encourages locally contractive mappings (robust to small perturbations).

14.3 Variational Autoencoders ⊠

14.3.1 Probabilistic Framework

VAE is a generative model:

$$p(\boldsymbol{x}) = \int p(\boldsymbol{x}|\boldsymbol{z})p(\boldsymbol{z})d\boldsymbol{z} \tag{14.5}$$

$$p(\boldsymbol{z}) = \mathcal{N}(\boldsymbol{0}, \boldsymbol{I}) \tag{14.6}$$

$$p(\boldsymbol{x}|\boldsymbol{z}) = \mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}_\theta(\boldsymbol{z}), \boldsymbol{\sigma}_\theta^2(\boldsymbol{z})\boldsymbol{I}) \tag{14.7}$$

14.3.2 Evidence Lower Bound (ELBO)

Cannot directly maximize $\log p(\boldsymbol{x})$. Instead maximize ELBO:

$$\mathcal{L} = \mathbb{E}_{q(\boldsymbol{z}|\boldsymbol{x})}[\log p(\boldsymbol{x}|\boldsymbol{z})] - D_{KL}(q(\boldsymbol{z}|\boldsymbol{x})\|p(\boldsymbol{z})) \tag{14.8}$$

where $q(\boldsymbol{z}|\boldsymbol{x}) = \mathcal{N}(\boldsymbol{z}; \boldsymbol{\mu}_\phi(\boldsymbol{x}), \boldsymbol{\sigma}_\phi^2(\boldsymbol{x})\boldsymbol{I})$ is the encoder.

14.3.3 Reparameterization Trick

To backpropagate through sampling:

$$\boldsymbol{z} = \boldsymbol{\mu}_\phi(\boldsymbol{x}) + \boldsymbol{\sigma}_\phi(\boldsymbol{x}) \odot \boldsymbol{\epsilon}, \quad \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I}) \tag{14.9}$$

Enables end-to-end gradient-based training.

14.3.4 Generation

Sample from prior $\boldsymbol{z} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$ and decode to generate new data.

14.3.5 Notes and references

VAEs provide a principled probabilistic framework for representation learning and generation Kingma and Welling [KW13], Goodfellow, Bengio, and Courville [GBC16a], and Prince [Pri23].

14.4 Applications of Autoencoders ☒

14.4.1 Dimensionality Reduction

Learn compact representations for:

- Visualization (like t-SNE, UMAP)
- Preprocessing for downstream tasks
- Feature extraction

14.4.2 Anomaly Detection

High reconstruction error indicates anomalies:

- Fraud detection
- Manufacturing defects
- Network intrusion detection

14.4.3 Denoising

DAEs remove noise from:

- Images
- Audio signals
- Sensor data

14.4.4 References

For autoencoder variants and use-cases, see Goodfellow, Bengio, and Courville [GBC16a] and Prince [Pri23].

14.5 Real World Applications

Autoencoders learn to compress and reconstruct data, finding compact representations that capture essential information. This capability enables numerous practical applications in compression, denoising, and anomaly detection.

14.5.1 Image and Video Compression

Efficient storage and transmission of visual data:

- **Next-generation image compression:** Traditional formats like JPEG use hand-crafted compression algorithms. Learned autoencoder-based compression achieves better quality at the same file size or smaller files at the same quality. This matters for websites, cloud storage, and mobile apps where bandwidth and storage costs are significant.

- **Video streaming optimization:** Netflix and YouTube experiment with autoencoder-based video compression to stream higher quality video at lower bitrates. This reduces buffering, saves bandwidth costs, and enables HD streaming in areas with limited internet connectivity. The autoencoders learn to preserve perceptually important details humans notice while discarding subtle information we don't.

- **Satellite imagery compression:** Earth observation satellites generate terabytes of imagery daily. Autoencoder compression reduces transmission bandwidth from space to ground stations, allowing more frequent imagery updates or higher resolution within bandwidth constraints. This improves applications from weather forecasting to agriculture monitoring.

14.5.2 Denoising and Enhancement

Improving signal quality in degraded data:

- **Medical image enhancement:** Denoising autoencoders improve quality of MRI and CT scans, reducing radiation exposure needed for diagnostic-quality images or enabling faster scanning. The autoencoder learns the manifold of healthy tissue appearance, removing noise while preserving medically relevant details like tumor boundaries.
- **Old photo restoration:** Consumer apps use autoencoders to remove scratches, stains, and aging artifacts from old photographs. The models learn the structure of clean images and infer what damaged regions likely looked like originally. This helps preserve family histories and restore historical photographs.
- **Audio enhancement:** Autoencoders clean up audio recordings, removing background noise, hum, or compression artifacts. This improves voice clarity in phone calls, enhances podcast quality, and helps restore old audio recordings. Unlike simple filtering, autoencoders understand speech structure and preserve natural sound.

14.5.3 Anomaly Detection

Identifying unusual patterns in complex systems:

- **Credit card fraud detection:** Autoencoders learn to represent normal spending patterns compactly. Fraudulent transactions often don’t fit these patterns well, resulting in poor reconstruction. High reconstruction error flags potential fraud for investigation. This catches novel fraud schemes without requiring examples of every possible type of fraud.
- **Industrial equipment monitoring:** Manufacturing plants use autoencoders to monitor vibration patterns, temperatures, and other sensor data from machinery. Normal operation reconstructs well; unusual patterns indicating bearing wear, misalignment, or impending failure show high reconstruction error, triggering maintenance before catastrophic breakdowns.

- **Cybersecurity threat detection:** Network security systems use autoencoders trained on normal traffic patterns. Malware, intrusions, and data exfiltration create unusual patterns that reconstruct poorly, alerting security teams. This detects zero-day attacks and insider threats that evade signature-based detection.

14.5.4 Why Autoencoders Excel

Key advantages in practical applications:

- **Unsupervised learning:** Don't require labeled examples, just normal data
- **Dimensionality reduction:** Capture essential information compactly
- **Noise robustness:** Learn underlying structure despite corrupted inputs
- **Reconstruction ability:** Can generate clean versions of corrupted data

These applications show how autoencoders bridge classical compression and modern deep learning, providing practical solutions for data efficiency, quality enhancement, and anomaly detection.

Key Takeaways

Key Takeaways 14

- **Bottlenecks and noise** force representations to capture structure, not memorisation.
- **Regularised variants** (denoising, sparse, contractive) improve robustness and usefulness.
- **Utility beyond reconstruction**: learned codes transfer to downstream tasks.

Problems

Easy

Problem 14.1 (Undercomplete AE)**.** Explain why undercomplete AEs can avoid trivial identity mapping.

Hint: Bottleneck limits capacity.

Problem 14.2 (Denoising Noise)**.** How does noise type affect learned features?
Hint: Gaussian vs. masking vs. salt-and-pepper.

Problem 14.3 (Sparse Codes)**.** List benefits of sparsity in latent codes.
Hint: Interpretability, robustness, compression.

Problem 14.4 (Contractive Penalty)**.** What does a Jacobian penalty encourage?
Hint: Local invariance.

Medium

Problem 14.5 (Loss Choices)**.** Compare MSE vs. cross-entropy for images.
Hint: Data scale/likelihood assumptions.

Problem 14.6 (Regularisation Trade-offs)**.** Contrast denoising vs. contractive penalties.
Hint: Noise robustness vs. local smoothing.

Hard

Problem 14.7 (Jacobian Penalty)**.** Derive gradient of contractive loss w.r.t. encoder parameters.
Hint: Chain rule through Jacobian norm.

Problem 14.8 (Generative Link)**.** Explain links between AEs and VAEs/flows.
Hint: Likelihood vs. reconstruction objectives.

Chapter 15

Representation Learning

This chapter discusses the central challenge of deep learning: learning meaningful representations from data.

Learning Objectives

After studying this chapter, you will be able to:

1. Define representations and desirable properties (invariance, disentanglement, sparsity).

2. Compare supervised, self-supervised, and contrastive objectives.

3. Evaluate representations via linear probes and transfer learning.

4. Relate information-theoretic perspectives to empirical practice.

Intuition

Good representations separate task-relevant factors from nuisance variability. Learning signals that compare positive and negative pairs, or predict masked content, shape geometry in embedding spaces to reflect semantics.

15.1 What Makes a Good Representation? ☒

15.1.1 Desirable Properties

Disentanglement: Different factors of variation are separated

- Changes in one dimension affect one factor
- Easier interpretation and manipulation

Invariance: Representation unchanged under irrelevant transformations

- Translation, rotation invariance for objects
- Speaker invariance for speech content

Smoothness: Similar inputs have similar representations

- Enables generalization
- Supports interpolation

Sparsity: Few features active for each input

- Computational efficiency
- Interpretability

15.1.2 Manifold Hypothesis

Natural data lies on low-dimensional manifolds embedded in high-dimensional space. Deep learning learns to:

- Discover the manifold structure
- Map data to meaningful coordinates on manifold

15.1.3 Notes and references

Desirable properties are discussed in modern DL texts; disentanglement and invariance connect to inductive biases and data augmentation Goodfellow, Bengio, and Courville [GBC16a] and Prince [Pri23].

15.2 Transfer Learning and Domain Adaptation ☒

15.2.1 Transfer Learning

Leverage knowledge from source task to improve target task:

Feature extraction:

1. Pre-train on large dataset (e.g., ImageNet)
2. Freeze convolutional layers
3. Train only final classification layers on target task

Fine-tuning:

1. Start with pre-trained model
2. Continue training on target task with lower learning rate
3. Optionally freeze early layers

15.2.2 Domain Adaptation

Adapt model when training (source) and test (target) distributions differ.

Approaches:

- **Domain-adversarial training:** Learn domain-invariant features
- **Self-training:** Use confident predictions on target domain
- **Multi-task learning:** Joint training on both domains

15.2.3 Few-Shot Learning

Learn from few examples per class:

- **Meta-learning:** Learn to learn quickly (MAML)
- **Prototypical networks:** Learn metric space
- **Matching networks:** Attention-based comparison

15.3 Self-Supervised Learning ☒

Learn representations without manual labels by solving pretext tasks.

15.3.1 Pretext Tasks

For images:

- **Rotation prediction:** Predict rotation angle
- **Jigsaw puzzle:** Arrange shuffled patches
- **Colorization:** Predict colors from grayscale
- **Inpainting:** Fill masked regions

For text:

- **Masked language modeling:** Predict masked words (BERT)
- **Next sentence prediction:** Predict if sentences are consecutive
- **Autoregressive generation:** Predict next token (GPT)

15.3.2 Benefits

- Leverage unlabeled data
- Learn general-purpose representations
- Often outperforms supervised pre-training

15.4 Contrastive Learning ☒

Learn representations by contrasting positive and negative pairs.

15.4.1 Core Idea

Maximize agreement between different views of same data (positive pairs), minimize agreement with other data (negative pairs).

15.4.2 SimCLR Framework

1. Apply two random augmentations to each image
2. Encode both views: $\boldsymbol{z}_i = f(\boldsymbol{x}_i)$, $\boldsymbol{z}_j = f(\boldsymbol{x}_j)$
3. Minimize contrastive loss (NT-Xent):

$$\ell_{i,j} = -\log \frac{\exp(\text{sim}(\boldsymbol{z}_i, \boldsymbol{z}_j)/\tau)}{\sum_{k=1}^{2N} \mathbb{I}_{[k \neq i]} \exp(\text{sim}(\boldsymbol{z}_i, \boldsymbol{z}_k)/\tau)} \tag{15.1}$$

where $\text{sim}(\cdot, \cdot)$ is cosine similarity and τ is temperature.

15.4.3 MoCo (Momentum Contrast)

Uses momentum encoder and queue of negative samples for efficiency.

15.4.4 BYOL (Bootstrap Your Own Latent)

Surprisingly, can work without negative samples using:

- Online network (updated by gradients)
- Target network (momentum update)
- Prediction head on online network

15.4.5 Applications

State-of-the-art results in:

- Image classification
- Object detection
- Segmentation
- Medical imaging with limited labels

15.5 Real World Applications

Representation learning—automatically discovering useful features from raw data—is fundamental to modern deep learning success. Good representations make downstream tasks easier and enable transfer learning across domains.

15.5.1 Transfer Learning in Computer Vision

Reusing learned representations saves time and data:

- **Medical imaging with limited data:** Hospitals often have only hundreds of labeled examples for rare diseases—far too few to train deep networks from scratch. Transfer learning solves this by starting with representations learned from millions of general images (ImageNet), then fine-tuning on medical data. A network that learned to recognize textures and shapes in everyday photos can adapt to recognize pathologies in X-rays with just a small medical dataset.

- **Custom object detection for businesses:** Retailers want to detect their specific products on shelves; manufacturers need to identify particular defects. Instead of collecting millions of labeled images, they use pre-trained vision models and fine-tune with just hundreds of examples. The learned representations of edges, textures, and objects transfer effectively, making custom vision systems practical for small businesses.

- **Wildlife monitoring:** Conservation projects use camera traps to monitor endangered species, generating millions of images. Transfer learning enables creating species classifiers with limited labeled examples, accelerating research without requiring biologists to manually label vast datasets.

15.5.2 Natural Language Processing

Learned language representations revolutionize text applications:

- **Multilingual models:** Modern language models learn representations capturing meaning across languages. A model trained on English, Spanish, and Chinese text learns that ”cat,” ”gato,” and ”mao” (Chinese for ”cat”) represent similar concepts. This enables zero-shot translation and allows improvements in high-resource languages to benefit low-resource languages automatically.

- **Domain adaptation:** Customer service chatbots use language models pre-trained on general text, then fine-tuned on company-specific conversations. The general language understanding (grammar, reasoning, world knowledge) transfers, while fine-tuning adds domain expertise. This makes sophisticated chatbots feasible without training from scratch.

- **Sentiment analysis for brands:** Companies monitor social media sentiment about their products. Instead of training separate models for each product, they use general text representations learned from billions of documents, then adapt to specific brand vocabulary. This provides accurate sentiment analysis even for newly launched products.

15.5.3 Cross-Modal Representations

Learning representations spanning multiple modalities:

- **Image-text search:** Systems like Google Images let you search photos using text descriptions. This requires representations where images and text descriptions of the same concept are similar. Models learn joint representations by training on millions of image-caption pairs, enabling finding relevant images even for queries with no exact text matches.
- **Video understanding:** YouTube's recommendation and search systems learn representations combining visual content, audio, speech transcripts, and metadata. These multi-modal representations understand videos better than any single modality alone, improving search relevance and recommendations.
- **Accessibility tools:** Screen readers for visually impaired users generate descriptions of images on web pages. Cross-modal representations trained on image-caption pairs enable generating relevant, helpful descriptions automatically, making the web more accessible.

15.5.4 Impact of Good Representations

Why representation learning matters:

- **Data efficiency:** Solve new tasks with less labeled data
- **Generalization:** Better performance on diverse, real-world examples
- **Knowledge transfer:** Expertise learned on one task helps others
- **Semantic understanding:** Captures meaningful structure in data

These applications demonstrate that representation learning is not just a theoretical concept—it's the foundation enabling practical deep learning with limited data and computational resources.

Key Takeaways

Key Takeaways 15

- **Representation quality** is judged by transfer and linear separability.
- **Self-supervision** shapes embeddings via predictive or contrastive signals.
- **Evaluation matters**: consistent probes and protocols enable fair comparison.

Problems

Easy

Problem 15.1 (Encoder-Decoder Symmetry)**.** Why share architecture between $q(z|x)$ and $p(x|z)$?

Hint: Computation; conceptual symmetry.

Problem 15.2 (KL in ELBO)**.** State the role of $D_{KL}(q||p)$ in VAE training.

Hint: Regularisation; posterior matching.

Problem 15.3 (Reparameterisation Trick)**.** Explain why reparameterisation enables gradient flow.

Hint: Sampling vs. deterministic path.

Problem 15.4 (Prior Choice)**.** Justify Gaussian prior for VAE latents.

Hint: Tractability; simplicity; universality.

Medium

Problem 15.5 (ELBO Derivation)**.** Derive the ELBO from Jensen's inequality.

Hint: $\log \mathbb{E}[X] \geq \mathbb{E}[\log X]$.

Problem 15.6 (Beta-VAE)**.** Explain how β-VAE encourages disentanglement.

Hint: Weighted KL penalty; independence.

Hard

Problem 15.7 (Posterior Collapse)**.** Analyse conditions causing posterior collapse and propose mitigation.
Hint: Strong decoder; KL annealing; free bits.

Problem 15.8 (Importance-Weighted ELBO)**.** Derive the importance-weighted ELBO and show it tightens the bound.
Hint: Multiple samples; log-mean-exp.

Chapter 16

Structured Probabilistic Models for Deep Learning

This chapter covers graphical models and their integration with deep learning.

Learning Objectives

After studying this chapter, you will be able to:

1. Explain directed and undirected graphical models and conditional independencies.
2. Combine neural networks with graphical structures for hybrid models.
3. Perform basic inference and understand when approximations are required.
4. Design learning objectives for structured prediction tasks.

Intuition

Graphical structure encodes assumptions about which variables interact. Neural components capture complex local relationships; the graph constrains global behavior, aiding data efficiency and interpretability.

16.1 Graphical Models 🌟

16.1.1 Motivation

Graphical models represent complex probability distributions using graphs:

- Nodes: Random variables
- Edges: Probabilistic dependencies

16.1.2 Bayesian Networks

Directed acyclic graphs (DAGs) represent conditional dependencies:

$$p(\boldsymbol{x}) = \prod_{i=1}^{n} p(x_i|\text{Pa}(x_i)) \tag{16.1}$$

where Pa(x_i) are parents of x_i.
Example: Naive Bayes classifier

$$p(y, \boldsymbol{x}) = p(y) \prod_{i=1}^{d} p(x_i|y) \tag{16.2}$$

16.1.3 Markov Random Fields

Undirected graphs with potential functions:

$$p(\boldsymbol{x}) = \frac{1}{Z} \prod_{c \in \mathcal{C}} \psi_c(\boldsymbol{x}_c) \tag{16.3}$$

where $\mathcal{C}$ are cliques and Z is the partition function.
Example: Ising model, Conditional Random Fields (CRFs)

16.2 Inference in Graphical Models 🌟

16.2.1 Exact Inference

Variable elimination: Marginalize variables sequentially
Belief propagation: Message passing on tree-structured graphs
Complexity exponential in tree-width, often intractable.

16.2.2 Approximate Inference

Variational inference: Optimize tractable approximation (Chapter 19)

Sampling methods: Monte Carlo approaches (Chapter 17)

Loopy belief propagation: Approximate inference on graphs with cycles

16.3 Deep Learning and Structured Models 🌟

16.3.1 Structured Output Prediction

Use graphical models to model output structure:

Conditional Random Fields (CRFs):

$$p(\boldsymbol{y}|\boldsymbol{x}) = \frac{1}{Z(\boldsymbol{x})} \exp\left(\sum_c \boldsymbol{w}^\top \boldsymbol{\phi}_c(\boldsymbol{x}, \boldsymbol{y}_c)\right) \tag{16.4}$$

Applications:

- Sequence labeling (NER, POS tagging)
- Image segmentation
- Parsing

16.3.2 Structured Prediction with Neural Networks

Combine neural networks with graphical models:

- **Feature extraction:** CNN/RNN extracts features
- **Structured inference:** CRF layer for structured output
- End-to-end training with backpropagation

Example: CNN-CRF for semantic segmentation

16.3.3 Neural Module Networks

Compose neural modules based on program structure for visual reasoning.

16.3.4 Graph Neural Networks

Graph neural networks (GNNs) operate on graphs via message passing and permutation-invariant aggregations; useful for molecules, social networks, and scene graphs Prince [Pri23].

16.4 Real World Applications

Structured probabilistic models capture dependencies and uncertainties in complex systems. These models enable reasoning under uncertainty and provide principled frameworks for decision-making in real-world applications.

16.4.1 Autonomous Vehicle Decision Making

Safe navigation requires reasoning about uncertainties:

- **Predicting pedestrian behavior:** Will that person step into the street or wait? Probabilistic models capture uncertainty about pedestrian intentions based on their position, posture, and gaze direction. The vehicle uses these probability distributions to make conservative decisions—slowing down when a pedestrian might cross, rather than assuming they won't.

- **Sensor fusion with uncertainty:** Self-driving cars combine cameras, radar, and lidar, each with different strengths and noise characteristics. Probabilistic graphical models integrate these sensors, weighing each according to reliability in current conditions (cameras work poorly in fog; radar penetrates fog better). This provides robust perception despite individual sensor limitations.

- **Planning under uncertainty:** Routes to destinations involve uncertain travel times due to traffic, weather, and road conditions. Structured probabilistic models help vehicles plan routes considering these uncertainties, balancing expected arrival time with reliability and providing realistic time estimates.

16.4.2 Medical Diagnosis and Treatment

Healthcare requires careful uncertainty quantification:

- **Bayesian diagnosis systems:** Medical diagnosis involves uncertainty—symptoms might indicate multiple diseases with different probabilities.

Structured probabilistic models encode relationships between symptoms, diseases, and test results, computing probability distributions over possible diagnoses. This helps doctors order appropriate tests and consider differential diagnoses systematically.

- **Personalized treatment planning:** Cancer treatment response varies by patient. Probabilistic models integrate genetic markers, tumor characteristics, and treatment histories to estimate probability distributions over treatment outcomes. Doctors use these to discuss risks and benefits with patients, making informed shared decisions about treatment options.

- **Drug interaction modeling:** Patients taking multiple medications face interaction risks. Structured models capture dependencies between drugs, considering individual patient factors (age, kidney function, genetics) to estimate risk probabilities. This enables safer prescribing, especially for elderly patients on many medications.

16.4.3 Natural Language Understanding

Language has inherent ambiguity and structure:

- **Machine translation quality:** Translation systems use probabilistic models to capture ambiguity—words have multiple possible translations depending on context. Structured models representing sentence structure help select appropriate translations, providing confidence estimates for different interpretations. This enables highlighting uncertain translations for human review.

- **Information extraction:** Extracting structured information (who did what to whom, when, where) from text requires understanding relationships between entities. Probabilistic graphical models capture these dependencies, providing uncertainty estimates about extracted information. News aggregators use these to reconcile potentially conflicting reports from multiple sources.

- **Voice assistant intent recognition:** ”Book a table for two” could mean restaurant reservations or furniture arrangements. Structured models use conversation context and user history to estimate intent probabilities, asking clarifying questions when uncertainty is high rather than guessing incorrectly.

16.4.4 Value of Structured Models

Why explicit structure matters:

- **Interpretability:** Model structure reflects domain knowledge and causal relationships
- **Uncertainty quantification:** Provides principled probability estimates
- **Data efficiency:** Structure reduces parameters and sample complexity
- **Reasoning:** Enables inference, prediction, and decision-making under uncertainty

These applications show how structured probabilistic models provide principled frameworks for dealing with uncertainty in safety-critical and high-stakes applications.

Key Takeaways

Key Takeaways 16

- **Structure** encodes independence, enabling efficient inference.
- **Hybrid models** leverage neural expressivity with graphical constraints.
- **Inference choices** depend on graph type and potential functions.

Problems

Easy

Problem 16.1 (Generator Role)**.** Describe the generator's objective in GANs.
Hint: Fool the discriminator.

Problem 16.2 (Discriminator Role)**.** Describe the discriminator's objective in GANs.
Hint: Distinguish real from fake.

Problem 16.3 (Mode Collapse)**.** Define mode collapse and its symptoms.
Hint: Generator produces limited variety.

Problem 16.4 (Training Instability)**.** Name two causes of GAN training instability.
Hint: Oscillation; gradient vanishing.

Medium

Problem 16.5 (Nash Equilibrium)**.** Explain why GAN training seeks a Nash equilibrium.
Hint: Minimax game; no unilateral improvement.

Problem 16.6 (Wasserstein Distance)**.** State the advantage of Wasserstein distance over JS divergence.
Hint: Gradient behaviour with non-overlapping distributions.

Hard

Problem 16.7 (Optimal Discriminator)**.** Derive the optimal discriminator for fixed generator.
Hint: Maximise expected log-likelihood.

Problem 16.8 (Spectral Normalisation)**.** Analyse how spectral normalisation stabilises GAN training.
Hint: Lipschitz constraint; gradient norms.

Chapter 17

Monte Carlo Methods

This chapter introduces sampling-based approaches for probabilistic inference and learning.

Learning Objectives

After studying this chapter, you will be able to:

1. Describe Monte Carlo estimation and variance reduction techniques.

2. Explain MCMC algorithms (Metropolis – Hastings, Gibbs) and their diagnostics.

3. Apply sampling to approximate expectations and gradients.

4. Identify pitfalls such as poor mixing and autocorrelation.

Intuition

When exact integrals are intractable, we approximate them with random samples. The art is to sample efficiently from complicated posteriors and to estimate uncertainty from finite chains.

17.1 Sampling and Monte Carlo Estimators 🌟

17.1.1 Monte Carlo Estimation

Approximate expectations using samples:

$$\mathbb{E}_{p(x)}[f(x)] \approx \frac{1}{N}\sum_{i=1}^{N} f(x^{(i)}), \quad x^{(i)} \sim p(x) \tag{17.1}$$

Law of large numbers: Estimate converges to true expectation as $N \to \infty$.

17.1.2 Variance Reduction

Reduce variance of estimators:

Rao-Blackwellization: Use conditional expectations

Control variates: Subtract correlated zero-mean terms

Antithetic sampling: Use negatively correlated samples

17.2 Markov Chain Monte Carlo 🌟

17.2.1 Markov Chains

Sequence where $p(x_t|x_{t-1},\ldots,x_1) = p(x_t|x_{t-1})$.

Stationary distribution: $\pi(x)$ such that if $x_t \sim \pi$, then $x_{t+1} \sim \pi$.

17.2.2 Metropolis-Hastings Algorithm

Sample from target distribution $p(x)$:

1. Propose: $x' \sim q(x'|x_t)$

2. Accept with probability:

$$A(x', x_t) = \min\left(1, \frac{p(x')q(x_t|x')}{p(x_t)q(x'|x_t)}\right) \tag{17.2}$$

3. If accepted, $x_{t+1} = x'$; otherwise $x_{t+1} = x_t$

17.2.3 Gibbs Sampling

Special case where each variable updated conditionally:

$$x_i^{(t+1)} \sim p(x_i | x_{-i}^{(t)}) \tag{17.3}$$

Simple when conditional distributions are tractable.

17.2.4 Hamiltonian Monte Carlo

Uses gradient information for efficient exploration:

- Treats parameters as position in physics simulation
- Uses momentum for faster mixing
- More efficient than random walk methods

17.3 Importance Sampling 🌟

Sample from proposal $q(x)$ instead of target $p(x)$:

$$\mathbb{E}_p[f(x)] = \mathbb{E}_q\left[\frac{p(x)}{q(x)} f(x)\right] \approx \frac{1}{N}\sum_{i=1}^{N} \frac{p(x^{(i)})}{q(x^{(i)})} f(x^{(i)}) \tag{17.4}$$

Effective when:

- q is easy to sample from
- q has heavier tails than p

17.4 Applications in Deep Learning 🌟

Bayesian deep learning:

- Sample network weights
- Uncertainty quantification

Reinforcement learning:

- Policy gradient estimation

- Monte Carlo tree search

Generative models:

- Training energy-based models
- Sampling from learned distributions

17.5 Real World Applications

Monte Carlo methods use random sampling to solve complex problems that would be intractable through direct computation. These techniques enable approximating difficult integrals, exploring high-dimensional spaces, and quantifying uncertainty.

17.5.1 Financial Risk Management

Understanding and managing financial uncertainty:

- **Value at Risk (VaR) estimation:** Banks must estimate potential losses to maintain adequate capital reserves. Monte Carlo simulation generates thousands of possible market scenarios, computing portfolio values in each. This provides distributions of potential losses rather than single-point estimates, helping banks understand risks in different market conditions.
- **Option pricing:** Financial derivatives have values depending on uncertain future asset prices. Monte Carlo methods simulate possible price paths, computing option values as averages over many scenarios. This handles complex derivatives (like exotic options with path-dependent payoffs) where analytical solutions don't exist.
- **Retirement planning:** Financial advisors use Monte Carlo simulation to project retirement savings over decades, considering uncertainties in investment returns, inflation, and life expectancy. Rather than promising a single outcome, simulations show probability distributions—like ”85% chance your savings last through age 95”—helping people make informed decisions.

17.5.2 Climate and Weather Modeling

Predicting complex physical systems:

- **Ensemble weather forecasting:** Weather services run multiple simulations with slightly different initial conditions, representing measurement uncertainty. Monte Carlo-style ensembles provide probability distributions for forecasts—like ”70% chance of rain”—more useful than deterministic predictions. This helps with decisions from umbrella-carrying to disaster preparedness.

- **Climate change projections:** Long-term climate models involve enormous uncertainty in cloud physics, ocean circulation, and human emissions. Monte Carlo sampling over parameter uncertainties generates probability distributions for future climate scenarios, informing policy decisions about emissions reductions and adaptation strategies.

- **Hurricane path prediction:** The ”cone of uncertainty” in hurricane forecasts comes from Monte Carlo simulations exploring possible paths given current conditions and atmospheric uncertainties. This helps emergency managers make evacuation decisions balancing safety against unnecessary disruption.

17.5.3 Drug Discovery and Design

Exploring chemical and biological spaces:

- **Molecular dynamics simulation:** Understanding how proteins fold and bind to drug molecules requires simulating atomic movements. Monte Carlo methods sample possible molecular configurations, computing binding affinities and predicting which drug candidates are worth expensive experimental testing. This accelerates drug discovery while reducing costs.

- **Clinical trial design:** Pharmaceutical companies use Monte Carlo simulation to design clinical trials, estimating statistical power under various scenarios. Simulations help determine sample sizes needed to detect treatment effects reliably, preventing under-powered trials that waste resources or miss effective treatments.

- **Dose optimization:** Finding optimal drug dosages involves balancing efficacy and toxicity under individual patient variability. Monte Carlo simulation explores dose-response relationships across patient populations, identifying regimens maximizing treatment benefit while minimizing risks.

17.5.4 Practical Advantages

Why Monte Carlo methods are indispensable:

- **Handle complexity:** Work when analytical solutions are impossible
- **Quantify uncertainty:** Provide probability distributions, not just point estimates
- **Scale naturally:** More samples improve accuracy predictably
- **Parallel computation:** Simulations run independently, leveraging modern computing

These applications demonstrate how Monte Carlo methods enable decision-making under uncertainty across finance, science, and healthcare—problems where exact answers are impossible but approximate probabilistic understanding is invaluable.

Key Takeaways

Key Takeaways 17

- **Monte Carlo** approximates expectations; variance control is essential.
- **MCMC** constructs dependent samples targeting complex posteriors.
- **Diagnostics** (ESS, R-hat) guide reliability of estimates.

Problems

Easy

Problem 17.1 (Self-Attention Intuition)**.** Explain why self-attention captures long-range dependencies.
Hint: Direct pairwise interactions.

Problem 17.2 (Positional Encoding)**.** Why do Transformers need positional encodings?
Hint: Permutation invariance of self-attention.

Problem 17.3 (Multi-Head Attention)**.** State the benefit of multiple attention heads.
Hint: Different representation subspaces.

Problem 17.4 (Masked Attention)**.** Explain the role of masking in causal attention.
Hint: Prevent future information leakage.

Medium

Problem 17.5 (Computational Complexity)**.** Derive the computational complexity of self-attention.
Hint: $O(n^2 d)$ for sequence length n, dimension d.

Problem 17.6 (LayerNorm vs. BatchNorm)**.** Compare LayerNorm and BatchNorm in Transformers.
Hint: Independence from batch; sequence-level statistics.

Hard

Problem 17.7 (Sparse Attention)**.** Design a sparse attention pattern and analyse complexity savings.
Hint: Local windows; strided patterns; $O(n \log n)$ or $O(n\sqrt{n})$.

Problem 17.8 (Attention Visualisation)**.** Propose methods to interpret attention weights and discuss limitations.
Hint: Attention rollout; gradient-based; correlation vs. causation.

Chapter 18

Confronting the Partition Function

This chapter addresses computational challenges in probabilistic models arising from intractable partition functions.

Learning Objectives

After studying this chapter, you will be able to:

1. Explain why partition functions are hard and where they arise.
2. Compare strategies like importance sampling, AIS, and contrastive methods.
3. Analyze bias/variance trade-offs in partition function estimation.
4. Implement practical estimators under compute constraints.

Intuition

Partition functions normalize probabilities by summing over exponentially many states. Estimators navigate this space by focusing effort on high-probability regions without exhaustive enumeration.

18.1 The Partition Function Problem 🌟

Many models have form:

$$p(\boldsymbol{x}) = \frac{1}{Z}\tilde{p}(\boldsymbol{x}) \tag{18.1}$$

where partition function $Z = \sum_{\boldsymbol{x}} \tilde{p}(\boldsymbol{x})$ or $Z = \int \tilde{p}(\boldsymbol{x})d\boldsymbol{x}$ is intractable.

18.1.1 Why It's Hard

Computing Z requires:

- Summing/integrating over all configurations
- Exponential in dimensionality
- Prohibitive for high-dimensional models

18.1.2 Impact

Cannot directly:

- Evaluate likelihood $p(\boldsymbol{x})$
- Compute gradients for learning
- Compare models

18.2 Contrastive Divergence 🌟

18.2.1 Motivation

For Restricted Boltzmann Machines (RBMs):

$$p(\boldsymbol{v}, \boldsymbol{h}) = \frac{1}{Z}\exp(-E(\boldsymbol{v}, \boldsymbol{h})) \tag{18.2}$$

Exact gradient requires expectations under model:

$$\frac{\partial \log p(\boldsymbol{v})}{\partial \theta} = -\mathbb{E}_{p(\boldsymbol{h}|\boldsymbol{v})}\left[\frac{\partial E}{\partial \theta}\right] + \mathbb{E}_{p(\boldsymbol{v},\boldsymbol{h})}\left[\frac{\partial E}{\partial \theta}\right] \tag{18.3}$$

18.2.2 CD-k Algorithm

Approximate second term with short MCMC chain (k steps):

1. Start from data: $\boldsymbol{v}_0 = \boldsymbol{v}$
2. Run k Gibbs steps
3. Use $\boldsymbol{v}_k$ for negative phase

Works surprisingly well despite being biased.

18.3 Noise-Contrastive Estimation difficultyInlineadvanced

18.3.1 Key Idea

Turn density estimation into binary classification:

- Distinguish data samples from noise samples
- Avoids computing partition function

18.3.2 NCE Objective

$$\mathcal{L} = \mathbb{E}_{p_{\text{data}}}[\log h(\boldsymbol{x})] + k \cdot \mathbb{E}_{p_{\text{noise}}}[\log(1 - h(\boldsymbol{x}))] \tag{18.4}$$

where:

$$h(\boldsymbol{x}) = \frac{p_{\text{model}}(\boldsymbol{x})}{p_{\text{model}}(\boldsymbol{x}) + k \cdot p_{\text{noise}}(\boldsymbol{x})} \tag{18.5}$$

18.3.3 Applications

- Word embeddings (word2vec)
- Language models
- Energy-based models

18.3.4 Notes and references

NCE as a technique to bypass partition functions is discussed in Goodfellow, Bengio, and Courville [GBC16a] and Prince [Pri23].

18.4 Score Matching 🌟

Match gradients of log-density (score function):

$$\psi(\boldsymbol{x}) = \nabla_{\boldsymbol{x}} \log p(\boldsymbol{x}) \tag{18.6}$$

Objective:

$$\mathcal{L} = \frac{1}{2}\mathbb{E}_{p_{\text{data}}}[||\psi_\theta(\boldsymbol{x}) - \nabla_{\boldsymbol{x}} \log p_{\text{data}}(\boldsymbol{x})||^2] \tag{18.7}$$

Avoids partition function since it cancels in gradient.

18.5 Real World Applications

Confronting the partition function—computing normalizing constants in probabilistic models—is a fundamental challenge. Practical applications require approximations and specialized techniques to make inference tractable in complex models.

18.5.1 Recommender Systems at Scale

Efficient scoring of millions of items:

- **YouTube video recommendations:** YouTube must score millions of videos for each user. Computing exact probabilities requires evaluating partition functions over all possible videos—computationally infeasible. Instead, systems use approximate methods like negative sampling and importance sampling, providing good recommendations efficiently. These approximations enable real-time personalization for billions of users.

- **E-commerce product ranking:** Online retailers face similar challenges ranking products. Models learn to score product-user compatibility, but exact probability computations are intractable. Contrastive learning methods approximate partition functions by sampling negative examples, enabling practical deployment at scale while maintaining recommendation quality.

- **Music playlist generation:** Streaming services create personalized playlists by modeling sequential song compatibility. Full probabilistic models would require intractable partition functions over all possible song sequences. Practical systems use locally normalized models and sampling-based approximations, generating engaging playlists efficiently.

18.5.2 Natural Language Processing

Handling large vocabularies efficiently:

- **Language model training:** Modern language models predict next words from vocabularies of 50,000+ tokens. Computing partition functions over all possible next words for every training example is expensive. Techniques like noise contrastive estimation and self-normalization make training practical, enabling language models that power translation, autocomplete, and conversational AI.

- **Neural machine translation:** Translation models generate target sentences word by word, considering vast numbers of possible continuations. Exact probability computation would require intractable partition functions. Beam search with approximate scoring enables practical translation systems, producing high-quality translations in real-time.

- **Named entity recognition:** Identifying people, places, and organizations in text involves structured prediction over exponentially many possible tag sequences. Conditional random fields require computing partition functions efficiently. The forward-backward algorithm provides exact computation for chain structures, enabling accurate entity extraction in applications from news analysis to medical record processing.

18.5.3 Computer Vision

Structured prediction in image understanding:

- **Semantic segmentation:** Labeling every pixel in images requires modeling dependencies between neighboring pixels. Fully modeling these dependencies involves intractable partition functions over pixel labelings. Practical systems use approximate inference (mean field approximation, pseudo-likelihood) or structured models with tractable partition functions (chain or tree structures).

- **Pose estimation:** Estimating human body poses involves predicting joint locations with anatomical constraints (arms connect to shoulders, legs have limited range). Models encoding these constraints have complex partition functions. Approximate inference techniques enable real-time pose estimation for applications from gaming to physical therapy.

- **Object detection:** Detecting objects requires scoring countless possible bounding boxes. Models learning to rank boxes face partition function challenges similar to recommendation systems. Techniques like contrastive learning and hard negative mining make training practical, enabling accurate detection in applications from autonomous driving to retail analytics.

18.5.4 Practical Solutions

Key strategies for handling partition functions:

- **Approximation methods:** Monte Carlo sampling, variational inference
- **Negative sampling:** Approximate partition functions using sampled negatives
- **Structured models:** Design models with tractable partition functions
- **Unnormalized models:** Use score-based approaches avoiding normalization

These applications show how confronting the partition function is not just a theoretical concern—it's a practical challenge requiring clever approximations to deploy probabilistic models at scale.

Key Takeaways

Key Takeaways 18

- **Partition functions** create intractable normalisers in many models.
- **Estimators** (IS, AIS, contrastive) trade bias and variance differently.
- **Practicality** depends on proposal quality and compute budget.

Problems

Easy

Problem 18.1 (MDP Definition)**.** Define the components of a Markov Decision Process.

Hint: States, actions, rewards, transition dynamics, discount factor.

Problem 18.2 (Value Function)**.** Explain the difference between $V(s)$ and $Q(s,a)$.
Hint: State value vs. action-value.

Problem 18.3 (Policy Types)**.** Contrast deterministic and stochastic policies.
Hint: Mapping vs. distribution over actions.

Problem 18.4 (Exploration vs. Exploitation)**.** Give two exploration strategies in RL.
Hint: ϵ-greedy; UCB; entropy regularisation.

Medium

Problem 18.5 (Bellman Equation)**.** Derive the Bellman equation for $Q(s,a)$.
Hint: Recursive relationship with successor states.

Problem 18.6 (Policy Gradient)**.** Explain why policy gradient methods are useful for continuous action spaces.
Hint: Direct parameterisation; differentiability.

Hard

Problem 18.7 (Actor-Critic Derivation)**.** Derive the advantage actor-critic update rule.
Hint: Baseline subtraction; variance reduction.

Problem 18.8 (Off-Policy Correction)**.** Analyse importance sampling for off-policy learning and its variance.
Hint: Likelihood ratio; distribution mismatch.

Chapter 19

Approximate Inference

This chapter explores methods for tractable inference in complex probabilistic models.

Learning Objectives

After studying this chapter, you will be able to:

1. Differentiate variational inference and sampling-based approaches.
2. Derive ELBO objectives and coordinate ascent updates for simple models.
3. Understand amortized inference and its benefits/limitations.
4. Evaluate approximation quality using diagnostics and bounds.

Intuition

Exact posteriors are rare. We instead optimize over a family of tractable distributions or draw dependent samples, trading bias and variance to approximate expectations we care about.

19.1 Variational Inference 🌟

19.1.1 Evidence Lower Bound (ELBO)

For latent variable model with intractable posterior $p(\boldsymbol{z}|\boldsymbol{x})$, approximate with $q(\boldsymbol{z})$:

$$\begin{aligned}
\log p(\boldsymbol{x}) &= \mathbb{E}_{q(\boldsymbol{z})}[\log p(\boldsymbol{x})] && (19.1)\\
&= \mathbb{E}_{q(\boldsymbol{z})}\left[\log \frac{p(\boldsymbol{x},\boldsymbol{z})}{p(\boldsymbol{z}|\boldsymbol{x})}\right] && (19.2)\\
&= \mathbb{E}_{q(\boldsymbol{z})}\left[\log \frac{p(\boldsymbol{x},\boldsymbol{z})}{q(\boldsymbol{z})}\right] + D_{KL}(q(\boldsymbol{z})\|p(\boldsymbol{z}|\boldsymbol{x})) && (19.3)\\
&\geq \mathbb{E}_{q(\boldsymbol{z})}\left[\log \frac{p(\boldsymbol{x},\boldsymbol{z})}{q(\boldsymbol{z})}\right] = \mathcal{L}(q) && (19.4)
\end{aligned}$$

Maximizing $\mathcal{L}(q)$ minimizes $D_{KL}(q(\boldsymbol{z})\|p(\boldsymbol{z}|\boldsymbol{x}))$.

19.1.2 Variational Family

Choose tractable family of distributions:
Mean field: Fully factorized

$$q(\boldsymbol{z}) = \prod_{i=1}^{n} q_i(z_i) \tag{19.5}$$

Structured: Allow some dependencies

$$q(\boldsymbol{z}) = \prod_{c} q_c(\boldsymbol{z}_c) \tag{19.6}$$

Trade-off between expressiveness and tractability.

19.1.3 Coordinate Ascent VI

Optimize each factor iteratively:

$$q_j^*(z_j) \propto \exp\left(\mathbb{E}_{q_{-j}}[\log p(\boldsymbol{z},\boldsymbol{x})]\right) \tag{19.7}$$

Guaranteed to converge to local optimum of ELBO.

19.1.4 Stochastic Variational Inference

Use stochastic gradients for scalability:

- Mini-batch data

- Monte Carlo estimation of expectations
- Reparameterization trick for low variance

19.2 Mean Field Approximation 🌟

19.2.1 Fully Factorized Approximation

Assume all variables independent:

$$q(\boldsymbol{z}) = \prod_{i=1}^{n} q_i(z_i) \tag{19.8}$$

19.2.2 Update Equations

For each variable:

$$\log q_j^*(z_j) = \mathbb{E}_{i \neq j}[\log p(\boldsymbol{z}, \boldsymbol{x})] + \text{const} \tag{19.9}$$

Iterate until convergence.

19.2.3 Properties

- Underestimates variance (overconfident)
- Computationally efficient
- Often good approximation in practice

19.3 Loopy Belief Propagation 🌟

19.3.1 Message Passing

On graphical models, pass messages between nodes:

$$m_{i \to j}(x_j) = \sum_{x_i} \psi(x_i, x_j)\psi(x_i) \prod_{k \in N(i) \setminus j} m_{k \to i}(x_i) \tag{19.10}$$

19.3.2 Beliefs

Compute marginals from messages:

$$b_i(x_i) \propto \psi(x_i) \prod_{j \in N(i)} m_{j \to i}(x_i) \tag{19.11}$$

19.3.3 Exact on Trees

For tree-structured graphs, converges to exact marginals.

19.3.4 Loopy Graphs

On graphs with cycles:

- May not converge
- Often gives good approximations
- Used in error-correcting codes, computer vision

19.4 Expectation Propagation 🌟

Approximates each factor with simpler distribution:

$$p(\boldsymbol{x}) = \frac{1}{Z} \prod_i f_i(\boldsymbol{x}) \approx \frac{1}{Z} \prod_i \tilde{f}_i(\boldsymbol{x}) \tag{19.12}$$

Iteratively refine approximations to match moments.
Better than mean field for multi-modal posteriors.

19.5 Real World Applications

Approximate inference makes complex probabilistic reasoning practical. When exact inference is intractable, approximate methods enable deploying sophisticated probabilistic models in real-world systems requiring fast, scalable inference.

19.5.1 Autonomous Systems

Real-time decision making under uncertainty:

- **Robot navigation in uncertain environments:** Robots operating in homes or warehouses face sensor noise and unpredictable obstacles. Approximate inference (particle filters, variational methods) enables real-time localization and mapping despite uncertainties. The robot continuously updates beliefs about its position and surroundings, making navigation decisions based on approximate posterior distributions computed in milliseconds.
- **Drone flight control:** Autonomous drones must track their position, velocity, and orientation while compensating for wind and sensor errors. Extended Kalman filters—a form of approximate inference—provide real-time state estimation enabling stable flight. This makes applications from package delivery to aerial photography practical.
- **Agricultural robots:** Farm robots use approximate inference to model crop health, soil conditions, and pest distributions from noisy sensor data. Variational inference enables processing data from multiple robots, building probabilistic maps guiding precision agriculture interventions like targeted watering or pesticide application.

19.5.2 Personalized Medicine

Tailoring treatment to individual patients:

- **Genomic data analysis:** Understanding disease risk from genetic variants requires integrating evidence across thousands of genes. Approximate inference in Bayesian models combines genetic data with clinical information, computing posterior probabilities for disease risk and treatment response. This enables precision medicine decisions about preventive care and drug selection.
- **Real-time patient monitoring:** ICU monitoring systems track dozens of vital signs, detecting deterioration early. Approximate inference in hierarchical models captures normal variation versus concerning trends, triggering alerts while avoiding false alarms that cause alarm fatigue among medical staff.
- **Cancer treatment optimization:** Tumor evolution models use approximate inference to predict how cancers respond to treatments and develop resistance.

These predictions help oncologists select treatment sequences maximizing long-term outcomes rather than just immediate tumor reduction.

19.5.3 Content Recommendation

Personalization at massive scale:

- **Real-time feed ranking:** Social media platforms rank posts for billions of users continuously. Approximate inference in probabilistic models estimates user preferences from sparse interactions, computing rankings in milliseconds. Variational methods enable scaling to massive user bases while capturing uncertainty in preference estimates.

- **Explore-exploit tradeoffs:** Recommendation systems balance showing proven content (exploit) versus trying new items (explore). Approximate Bayesian inference maintains uncertainty estimates about item quality, implementing principled exploration strategies like Thompson sampling. This prevents recommendation systems from getting stuck showing only popular content.

- **Cold start recommendations:** New users have minimal history. Approximate inference in hierarchical models shares information across users, providing reasonable recommendations immediately. As users interact, the system refines individual preference estimates through ongoing approximate inference.

19.5.4 Natural Language Systems

Understanding language at scale:

- **Document understanding:** Extracting structured information from documents (contracts, medical records, scientific papers) involves uncertain entity recognition and relation extraction. Approximate inference in structured models provides confidence estimates, flagging uncertain extractions for human verification while automating clear cases.

- **Conversational AI:** Chatbots maintain beliefs about conversation state and user intent through approximate inference. This handles ambiguity gracefully—when uncertain about user goals, systems ask clarifying questions rather than guessing wrongly.

- **Machine translation:** Modern translation uses approximate inference to explore possible translations efficiently. Beam search—a form of approximate inference—enables finding high-quality translations without exhaustively evaluating all possibilities.

19.5.5 Why Approximation Is Essential

Practical benefits of approximate inference:

- **Scalability:** Handle complex models on real-world data sizes
- **Speed:** Provide results fast enough for interactive applications
- **Flexibility:** Enable sophisticated models despite computational constraints
- **Uncertainty:** Maintain probabilistic reasoning benefits with practical efficiency

These applications demonstrate that approximate inference is not a compromise—it's what makes probabilistic modeling practical at scale.

Key Takeaways

Key Takeaways 19

- **VI vs. MCMC**: bias-variance trade-offs define suitability.
- **ELBO optimisation** turns inference into tractable learning.
- **Amortisation** speeds inference but can underfit the posterior.

Problems

Easy

Problem 19.1 (Why Approximate?)**.** Explain why exact inference is intractable in many models.
Hint: Partition function; high-dimensional integration.

Problem 19.2 (ELBO Connection)**.** Relate ELBO to KL divergence between q and p.
Hint: $\log p(x) = \text{ELBO} + D_{KL}(q||p)$.

Problem 19.3 (Mean-Field Assumption)**.** State the mean-field independence assumption.
Hint: Factored variational distribution.

Problem 19.4 (MCMC vs. VI)**.** Compare MCMC and variational inference trade-offs.
Hint: Asymptotic exactness vs. computational speed.

Medium

Problem 19.5 (Coordinate Ascent VI)**.** Derive the coordinate ascent update for a simple model.
Hint: Fix all but one factor; optimise w.r.t. remaining factor.

Problem 19.6 (Importance Sampling)**.** Explain how importance sampling estimates expectations.
Hint: Reweight samples from proposal distribution.

Hard

Problem 19.7 (Amortised Inference)**.** Analyse the trade-offs of amortised inference in VAEs.
Hint: Amortisation gap; scalability.

Problem 19.8 (Reparameterisation Gradients)**.** Derive the reparameterisation gradient for a Gaussian variational distribution.
Hint: $z = \mu + \sigma\epsilon$ where $\epsilon \sim \mathcal{N}(0, 1)$.

Chapter 20

Deep Generative Models

This chapter examines modern approaches to generating new data samples using deep learning.

Learning Objectives

After studying this chapter, you will be able to:

1. Compare VAEs, GANs, and flow/diffusion models conceptually and practically.
2. Write training objectives and sampling procedures for each class.
3. Evaluate generative models with proper metrics and qualitative checks.
4. Discuss trade-offs in likelihood, sample quality, and mode coverage.

Intuition

Generative models learn data distributions in complementary ways: explicit likelihoods, implicit adversarial training, or score-based diffusion. Each chooses a tractable learning signal that captures structure while managing complexity.

20.1 Variational Autoencoders (VAEs) 🌟

(See also Chapter 14 for detailed VAE coverage.)

20.1.1 Recap

VAE learns latent representation $\boldsymbol{z}$ and decoder $p_\theta(\boldsymbol{x}|\boldsymbol{z})$:

$$\max_{\theta,\phi} \mathbb{E}_{q_\phi(\boldsymbol{z}|\boldsymbol{x})}[\log p_\theta(\boldsymbol{x}|\boldsymbol{z})] - D_{KL}(q_\phi(\boldsymbol{z}|\boldsymbol{x})\|p(\boldsymbol{z})) \tag{20.1}$$

20.1.2 Conditional VAEs

Generate conditioned on class or attributes:

$$\max \mathbb{E}_{q(\boldsymbol{z}|\boldsymbol{x},y)}[\log p(\boldsymbol{x}|\boldsymbol{z},y)] - D_{KL}(q(\boldsymbol{z}|\boldsymbol{x},y)\|p(\boldsymbol{z})) \tag{20.2}$$

20.1.3 Disentangled Representations

β-VAE: Increase KL weight for disentanglement

$$\mathcal{L} = \mathbb{E}_q[\log p(\boldsymbol{x}|\boldsymbol{z})] - \beta D_{KL}(q(\boldsymbol{z}|\boldsymbol{x})\|p(\boldsymbol{z})) \tag{20.3}$$

20.2 Generative Adversarial Networks (GANs) 🌟

20.2.1 Core Idea

Two networks compete:

- **Generator** G: Creates fake samples from noise
- **Discriminator** D: Distinguishes real from fake

20.2.2 Objective

Minimax game:

$$\min_G \max_D \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))] \tag{20.4}$$

20.2.3 Training Procedure

Alternate updates:

1. **Update D:** Maximize discrimination

$$\max_D \mathbb{E}_{\boldsymbol{x}}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}}[\log(1 - D(G(\boldsymbol{z})))] \tag{20.5}$$

2. **Update G:** Minimize discrimination (or maximize log $D(G(\boldsymbol{z}))$)

$$\min_G \mathbb{E}_{\boldsymbol{z}}[\log(1 - D(G(\boldsymbol{z})))] \tag{20.6}$$

20.2.4 Training Challenges

Mode collapse: Generator produces limited variety

Training instability: Oscillations, non-convergence

Vanishing gradients: When discriminator too strong

20.2.5 GAN Variants

DCGAN: Deep Convolutional GAN with architectural guidelines

WGAN: Wasserstein GAN with improved training stability

StyleGAN: High-quality image generation with style control

Conditional GAN: Generate from class labels

CycleGAN: Unpaired image-to-image translation

20.3 Normalizing Flows 🌟

20.3.1 Key Idea

Transform simple distribution (e.g., Gaussian) through invertible mappings:

$$\boldsymbol{x} = f_\theta(\boldsymbol{z}), \quad \boldsymbol{z} \sim p_z(\boldsymbol{z}) \tag{20.7}$$

20.3.2 Change of Variables

Density transforms as:

$$p_x(\boldsymbol{x}) = p_z(f^{-1}(\boldsymbol{x})) \left| \det \frac{\partial f^{-1}}{\partial \boldsymbol{x}} \right| \tag{20.8}$$

or equivalently:

$$\log p_x(\boldsymbol{x}) = \log p_z(\boldsymbol{z}) - \log \left| \det \frac{\partial f}{\partial \boldsymbol{z}} \right| \tag{20.9}$$

20.3.3 Requirements

Function f must be:

- Invertible
- Have tractable Jacobian determinant

20.3.4 Flow Architectures

Coupling layers: Split dimensions and transform half conditioned on other half
Autoregressive flows: Each dimension depends on previous ones
Continuous normalizing flows: Use neural ODEs

20.3.5 Advantages

- Exact likelihood computation
- Exact sampling
- Stable training (no adversarial dynamics)

20.4 Diffusion Models 🌟

20.4.1 Forward Process

Gradually add noise over T steps:

$$q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1}) = \mathcal{N}(\boldsymbol{x}_t; \sqrt{1-\beta_t}\boldsymbol{x}_{t-1}, \beta_t \boldsymbol{I}) \tag{20.10}$$

Eventually $\boldsymbol{x}_T \approx \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$.

20.4.2 Reverse Process

Learn to denoise (reverse diffusion):

$$p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t) = \mathcal{N}(\boldsymbol{x}_{t-1}; \boldsymbol{\mu}_\theta(\boldsymbol{x}_t, t), \boldsymbol{\Sigma}_\theta(\boldsymbol{x}_t, t)) \tag{20.11}$$

20.4.3 Training

Predict noise $\boldsymbol{\epsilon}_\theta(\boldsymbol{x}_t, t)$ at each step:

$$\mathcal{L} = \mathbb{E}_{t,\boldsymbol{x}_0,\boldsymbol{\epsilon}}\left[\|\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\boldsymbol{x}_t, t)\|^2\right] \tag{20.12}$$

20.4.4 Sampling

Start from noise and iteratively denoise:

$$\boldsymbol{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(\boldsymbol{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\boldsymbol{x}_t, t)\right) + \sigma_t \boldsymbol{z} \tag{20.13}$$

20.4.5 Advantages

- High-quality generation (DALL-E 2, Stable Diffusion, Midjourney)
- Stable training
- Strong theoretical foundations
- Can condition on text, images, etc.

20.5 Applications and Future Directions 🌟

20.5.1 Current Applications

Image generation:

- Text-to-image (DALL-E, Stable Diffusion)
- Image editing and inpainting
- Super-resolution
- Style transfer

Text generation:

- Large language models (GPT family)
- Code generation
- Creative writing

Audio/speech:

- Text-to-speech
- Music generation

- Voice conversion

Video:

- Video prediction
- Video synthesis
- Animation

Scientific applications:

- Molecule design
- Protein structure prediction
- Materials discovery

20.5.2 Future Directions

- **Controllability:** Fine-grained control over generation
- **Efficiency:** Faster sampling and smaller models
- **Multi-modal:** Unified models across modalities
- **Reasoning:** Incorporating logical reasoning
- **Safety:** Preventing harmful content generation
- **Evaluation:** Better metrics for generation quality

20.5.3 Societal Impact

Generative models raise important considerations:

- Copyright and intellectual property
- Misinformation and deepfakes
- Job displacement in creative fields
- Environmental cost of large-scale training
- Equitable access to technology

Responsible development requires addressing these challenges while advancing capabilities.

20.6 Real World Applications

Deep generative models create new data samples, enabling applications from content creation to scientific discovery. Recent advances in GANs, VAEs, and diffusion models have made generation remarkably realistic and controllable.

20.6.1 Creative Content Generation

AI-powered creativity and design:

- **AI art and design tools:** Services like Midjourney, DALL-E, and Stable Diffusion let anyone create professional-quality images from text descriptions. Designers use these for rapid prototyping—generating dozens of concept variations in minutes rather than hours of manual work. Artists use them as creative partners, combining AI-generated elements with traditional techniques. This democratizes visual content creation while augmenting professional workflows.

- **Music composition:** Generative models create original music in various styles, from background scores for videos to experimental compositions. Services generate royalty-free music customized to desired mood, tempo, and instrumentation. Musicians use these tools for inspiration or to quickly produce demos, while content creators get affordable custom soundtracks.

- **Architectural and product design:** Generative models explore design spaces, proposing variations on building layouts or product designs. Architects generate floor plan alternatives considering constraints like lighting and space efficiency. Product designers iterate rapidly through form variations, accelerating the creative process from concept to prototype.

20.6.2 Scientific Discovery

Generating hypotheses and solutions:

- **Drug molecule design:** Generative models propose novel drug candidates with desired properties (binding to target proteins, good safety profiles, ease of synthesis). This explores chemical space more efficiently than trial-and-error synthesis, potentially accelerating drug discovery. Companies are using these models to design treatments for everything from cancer to infectious diseases.

- **Materials science:** Researchers use generative models to design new materials with specific properties—stronger alloys, better batteries, more efficient solar cells. The models learn relationships between molecular structure and properties, proposing novel materials for experimental validation. This could accelerate development of technologies for clean energy and sustainability.

- **Protein structure prediction and design:** Generative models help predict how proteins fold and design proteins with novel functions. This enables creating enzymes for industrial processes, developing new vaccines, and understanding disease mechanisms. AlphaFold's success in protein structure prediction demonstrates how generative models advance biological understanding.

20.6.3 Data Augmentation and Synthesis

Generating training data:

- **Synthetic medical images:** Medical datasets are limited by privacy concerns and rare diseases. Generative models create synthetic training data that looks realistic but doesn't correspond to real patients. This enables training better diagnostic models while protecting privacy and addressing data imbalances in rare conditions.

- **Simulation for autonomous vehicles:** Generative models create realistic synthetic driving scenarios—rare events like pedestrians jaywalking or vehicles running red lights. Self-driving cars train on these synthetic scenarios, becoming prepared for dangerous situations without risking real-world testing. This addresses the ”long tail” of rare but critical edge cases.

- **Video game content generation:** Game developers use generative models to create textures, terrain, character models, and even entire game levels. This reduces development costs and time while increasing content variety. Procedural generation creates unique experiences for each player rather than manually crafting every asset.

20.6.4 Personalization and Adaptation

Customized content for individuals:

- **Avatar creation:** Apps generate personalized avatars from photos, creating cartoon or stylized versions maintaining recognizable features. These appear in messaging apps, games, and virtual meetings, providing fun, privacy-conscious representations of users.

- **Text-to-speech personalization:** Generative models create natural-sounding speech in your own voice from text. This helps people who lose their voice due to illness preserve their vocal identity. It also enables personalized audiobook narration and accessible content in preferred voices.

- **Style transfer and image editing:** Apps apply artistic styles to photos, change seasons in landscape photography, or age/de-age faces realistically. These features make sophisticated image manipulation accessible to everyone, from professional photographers to casual social media users.

20.6.5 Transformative Impact

Why generative models matter:

- **Democratization:** Creative tools accessible to everyone, not just experts

- **Acceleration:** Rapid iteration and exploration of design spaces

- **Discovery:** Finding solutions in complex domains like chemistry and biology

- **Synthesis:** Creating training data and simulations otherwise unavailable

These applications show generative models are not just impressive demonstrations—they're practical tools transforming creative work, scientific discovery, and everyday applications.

Key Takeaways

Key Takeaways 20

- **Model families** differ in training signal and guarantees.
- **Evaluation** must consider likelihood, fidelity, diversity, and downstream utility.
- **Trade-offs** are inevitable: choose for the target application.

Problems

Easy

Problem 20.1 (VAE vs. GAN)**.** Compare the training objectives of VAEs and GANs.
Hint: Likelihood-based vs. adversarial.

Problem 20.2 (Normalising Flow Invertibility)**.** Why must normalising flows be invertible?
Hint: Exact likelihood computation via change of variables.

Problem 20.3 (Diffusion Process)**.** Describe the forward diffusion process in diffusion models.
Hint: Gradual addition of Gaussian noise.

Problem 20.4 (Sampling Speed)**.** Compare sampling speed across VAEs, GANs, and diffusion models.
Hint: Single pass vs. iterative refinement.

Medium

Problem 20.5 (Flow Jacobian)**.** Derive the change-of-variables formula for a normalising flow.
Hint: $\log p(x) = \log p(z) + \log|\det \frac{\partial f}{\partial z}|$.

Problem 20.6 (Denoising Score Matching)**.** Explain how diffusion models learn the score function.
Hint: Predict noise; connection to $\nabla_x \log p(x)$.

Hard

Problem 20.7 (Coupling Layer Design)**.** Design an invertible coupling layer and prove its properties.
Hint: Affine transformations; partition dimensions.

Problem 20.8 (Guidance Trade-off)**.** Analyse the trade-off between sample quality and diversity in classifier-free guidance.
Hint: Guidance scale; conditional vs. unconditional scores.

Glossary

accuracy Proportion of correct predictions over all predictions in a classification task.. 208

attention mechanism A technique that allows models to focus on relevant parts of the input when making predictions.. 195, 197

backpropagation An algorithm for training neural networks that computes gradients by propagating errors backward through the network.. 198

BLEU Bilingual Evaluation Understudy; an n-gram overlap metric for machine translation quality.. 210

computer vision A field of artificial intelligence that enables computers to interpret and understand visual information.. 2

exploding gradient problem A problem in deep networks where gradients become exponentially large as they propagate backward, causing unstable training.. 158, 189

gradient clipping Rescaling or capping gradients to control exploding gradients during training.. 190

gradient descent An optimization algorithm that iteratively adjusts parameters in the direction of steepest descent of the loss function.. 150

long short-term memory A type of recurrent neural network architecture designed to overcome the vanishing gradient problem.. 191

mini-batch A small subset of the training data used in each iteration of gradient descent, typically containing 32-256 examples.. 150, 152

natural language processing A field of artificial intelligence that focuses on the interaction between computers and human language.. 2

neural network A computing system inspired by biological neural networks, consisting of interconnected nodes (neurons).. 1

precision Fraction of predicted positives that are true positives.. 208

recall Fraction of actual positives that are correctly identified.. 208

receptive field The spatial extent in the input that influences a unit's activation in a later layer.. 169

recurrent neural network A type of neural network designed for sequential data, where connections form directed cycles.. 186, 187, 193

ROUGE Recall-Oriented Understudy for Gisting Evaluation; a set of metrics for automatic summarization evaluation.. 210

stochastic gradient descent A variant of gradient descent that updates parameters using the gradient from a single example at a time.. 151

vanishing gradient problem A problem in deep networks where gradients become exponentially small as they propagate backward, making training difficult.. 158, 189

Index

Bibliography

[Ama98] Shun-ichi Amari. “Natural gradient works efficiently in learning”. In: *Neural Computation* 10.2 (1998), pp. 251–276.

[BCB14] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. “Neural machine translation by jointly learning to align and translate”. In: *arXiv preprint arXiv:1409.0473* (2014).

[Bis06] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[Cho+14] Kyunghyun Cho et al. “Learning phrase representations using RNN encoder-decoder for statistical machine translation”. In: *arXiv preprint arXiv:1406.1078* (2014).

[Dev+18] Jacob Devlin et al. “BERT: Pre-training of deep bidirectional transformers for language understanding”. In: *arXiv preprint arXiv:1810.04805* (2018).

[DHS11] John Duchi, Elad Hazan, and Yoram Singer. “Adaptive subgradient methods for online learning and stochastic optimization”. In: *Journal of Machine Learning Research*. Vol. 12. 2011, pp. 2121–2159.

[GG16] Yarin Gal and Zoubin Ghahramani. “Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning”. In: *International Conference on Machine Learning (ICML)*. PMLR. 2016, pp. 1050–1059.

[GBC16a] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL: `http://www.deeplearningbook.org`.

[GBC16b] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning Book: Optimization for Training Deep Models*. https://www.deeplearningbook.org/contents/optimization.html. Accessed 2025-10-14. 2016.

[GBC16c] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning Book: Sequence Modeling – Recurrent and Recursive Nets*. https://www.deeplearningbook.org/contents/rnn.html. Accessed 2025-10-14. 2016.

[Goo+14] Ian Goodfellow et al. “Generative adversarial nets”. In: *Advances in neural information processing systems*. 2014, pp. 2672–2680.

[He+16] Kaiming He et al. “Deep residual learning for image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. “Long short-term memory”. In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[Hua+16] Gao Huang et al. “Deep Networks with Stochastic Depth”. In: *European Conference on Computer Vision (ECCV)*. Springer, 2016, pp. 646–661.

[IS15] Sergey Ioffe and Christian Szegedy. “Batch normalization: Accelerating deep network training by reducing internal covariate shift”. In: *arXiv preprint arXiv:1502.03167* (2015).

[KB14] Diederik P Kingma and Jimmy Ba. “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980* (2014).

[KW13] Diederik P Kingma and Max Welling. “Auto-encoding variational bayes”. In: *arXiv preprint arXiv:1312.6114* (2013).

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “ImageNet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[LeC+89] Yann LeCun et al. “Backpropagation applied to handwritten zip code recognition”. In: *Neural computation* 1.4 (1989), pp. 541–551.

[LN89] Dong C Liu and Jorge Nocedal. “On the limited memory BFGS method for large scale optimization”. In: *Mathematical Programming* 45.1-3 (1989), pp. 503–528.

[Nes83] Yurii Nesterov. “A method for solving the convex programming problem with convergence rate O(1/k^2)”. In: *Soviet Mathematics Doklady* 27 (1983), pp. 372–376.

[Pol64] Boris T. Polyak. “Some methods of speeding up the convergence of iteration methods”. In: *USSR Computational Mathematics and Mathematical Physics* 4.5 (1964), pp. 1–17.

[Pri23] Simon J. D. Prince. *Understanding Deep Learning*. MIT Press, 2023. URL: `https://udlbook.github.io/udlbook/`.

[Rad+19] Alec Radford et al. “Language models are unsupervised multitask learners”. In: *OpenAI blog* 1.8 (2019), p. 9.

[RM51] Herbert Robbins and Sutton Monro. “A stochastic approximation method”. In: *The Annals of Mathematical Statistics* 22.3 (1951), pp. 400–407.

[RFB15] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. “U-net: Convolutional networks for biomedical image segmentation”. In: *International Conference on Medical image computing and computer-assisted intervention*. Springer. 2015, pp. 234–241.

[RHW86] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. “Learning representations by back-propagating errors”. In: *Nature* 323.6088 (1986), pp. 533–536.

[Sil+16] David Silver et al. “Mastering the game of Go with deep neural networks and tree search”. In: *Nature* 529.7587 (2016), pp. 484–489.

[Sri+14] Nitish Srivastava et al. “Dropout: a simple way to prevent neural networks from overfitting”. In: vol. 15. 1. 2014, pp. 1929–1958.

[TH12] Tijmen Tieleman and Geoffrey Hinton. *Lecture 6.5—RMSProp: Divide the gradient by a running average of its recent magnitude*. COURSERA: Neural Networks for Machine Learning. 2012.

[Vas+17] Ashish Vaswani et al. “Attention is all you need”. In: *Advances in neural information processing systems*. 2017, pp. 5998–6008.

[Wik25a] Wikipedia contributors. *Attention (machine learning)*. `https://en.wikipedia.org/wiki/Attention_(machine_learning)`. Accessed 2025-10-14. 2025.

[Wik25b] Wikipedia contributors. *Recurrent neural network*. https://en.wikipedia.org/wiki/Recurrent_neural_network. Accessed 2025-10-14. 2025.

[Yun+19] Sangdoo Yun et al. “CutMix: Regularization Strategy to Train Strong Classifiers with Localizable Features”. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. 2019, pp. 6023–6032.

[Zha+24a] Aston Zhang et al. *Dive into Deep Learning: Attention Mechanisms and Transformers*. https://d2l.ai/chapter_attention-mechanisms-and-transformers/index.html. Accessed 2025-10-14. 2024.

[Zha+24b] Aston Zhang et al. *Dive into Deep Learning: Optimization*. https://d2l.ai/chapter_optimization/index.html. Accessed 2025-10-14. 2024.

[Zha+24c] Aston Zhang et al. *Dive into Deep Learning: Recurrent Neural Networks*. https://d2l.ai/chapter_recurrent-neural-networks/index.html. Accessed 2025-10-14. 2024.

[Zha+18] Hongyi Zhang et al. “mixup: Beyond Empirical Risk Minimization”. In: *International Conference on Learning Representations (ICLR)*. 2018.