

Mathematics of Generative AI

Michael Chertkov

Applied Mathematics, University of Arizona, Tucson, AZ 85721, USA

December 14, 2025

Contents

1 Linear Algebra (of AI)	10
1.1 Foundations of Representing Data	10
1.1.1 Vectors	10
1.1.2 Matrices: Representing Linear Transformations.	11
1.1.3 Convolution: Bridging Linear Algebra and Applications in AI	13
1.1.4 Tensors: The Generalization	14
1.1.5 Applications in Generative AI – Mechanics of Transformers	21
1.2 Matrix Decompositions	27
1.2.1 Singular Value Decomposition	27
1.2.2 Reduced Representation of a Single Data Point with SVD	30
1.2.3 Eigen-Decomposition	34
1.2.4 Connecting SVD and ED for Symmetric Positive-Definite Matrices . .	35
2 Calculus and Differential Equations (in AI)	38
2.1 Automatic Differentiation	38
2.1.1 Forward vs. Reverse Mode AD	39
2.2 Differential Equations: Foundations and Links to AI	43
2.2.1 Ordinary Differential Equations (ODEs) – Primer	44
2.2.2 Regression – Direct and ODE-Based	46
2.2.3 Second-Order ODE	51
2.3 System of Linear ODEs	54
2.3.1 Homogeneous ODEs	55
2.3.2 Inhomogeneous ODEs	57
2.3.3 Dynamics over Graph	59
2.3.4 Time-Ordered Exponential	62
3 Optimization (in AI)	65
3.1 Starting Example — Logistic Regression	66
3.1.1 The Logistic Regression Model	66
3.1.2 Linear Logistic Regression and Its Limitations	67
3.1.3 Why Linear Logistic Regression Fails for Non-Linearly Separable Data	67
3.1.4 Gradient Descent for Logistic Regression: The Vector Field	68
3.1.5 Nonlinear Logistic Regression via Feature Engineering	70
3.2 Convex Optimization – Primer	71
3.2.1 Variety of (Non-Convex) Landscapes	71

3.2.2	Convexity: A Guiding Light in AI Optimization	74
3.2.3	Convexity of Logistic Regression	75
3.2.4	Constrained Optimization and Lagrange Multipliers	76
3.3	Gradient Descent and Its Essential AI Variants	77
3.3.1	Gradient Descent (GD)	77
3.3.2	Stochastic Gradient Descent (SGD)	78
3.3.3	Momentum-Based Methods	78
3.3.4	Projected Gradient Descent (PGD)	79
3.3.5	Adaptive Learning Rate Methods	79
3.3.6	Why the Second-Order Methods are no go in AI?	84
3.4	Regularization & Sparsity	85
3.4.1	Compressed Sensing and Sparse Optimization	86
3.4.2	Regularization and Its Importance in AI	88
3.4.3	Sparsity for Inference Acceleration	89
3.5	(*) Optimization of Transformers – GenAI Example	90
3.5.1	Loss Function and Optimization Objective	91
3.5.2	Optimization Process: From Backpropagation to SGD Variants	91
3.5.3	How Does the Loss Function Handle Growing Input Sequences?	91
3.5.4	Ongoing Advances in Transformer Optimization	95
4	Neural Networks & Deep Learning	99
4.1	Neural Network Mechanics	100
4.1.1	Perceptron – Historical Remark	100
4.1.2	NN with a Single Fully Connected Hidden Layer	101
4.1.3	Interpolation vs. Extrapolation: Polynomial Regression vs Neural Networks	104
4.1.4	Simple Convolutional Neural Network	107
4.2	Neural Architectures	111
4.2.1	From CNN to ResNet – the Power of Skip Connections	113
4.2.2	From Residual Networks to Neural ODEs: A 2D Spiral Example	117
4.3	Universal Geometric Principles of Deep Learning	122
4.3.1	Discovery of Flat Regions in the Energy Landscape	123
4.3.2	Dynamic Selection of Low-Dimensional Manifolds in Deep Networks	127
4.4	Further Reading and Roadmap to the Rest of the Book	130
5	Probability and Statistics	134
5.1	Primer for Probability Spaces & Random Variables	136
5.1.1	Probability Spaces: The Foundation	137
5.1.2	Random Variables	138
5.1.3	Expectation and Moments	139
5.1.4	Data-Driven Probability: Empirical Distributions	139
5.1.5	Transformations of Random Variables	140
5.1.6	A First Normalizing Flow in 1D	142
5.2	Transforming Probability Distributions	143
5.2.1	Change of Variables in One Dimension	143

5.2.2	From Spiky to Smooth: Kernel Density Estimation	144
5.2.3	From Gaussian to Arbitrary Distributions: Normalizing Flows	146
5.2.4	Normalizing Flows and Optimal Transport (Preview)	147
5.3	Multivariate Random Variables	148
5.3.1	Random Vectors, Joint Distributions, and Independence	149
5.3.2	Algebraic and Linear Operations on Random Vectors	150
5.3.3	Multivariate Gaussian Distributions	151
5.3.4	Empirical Multivariate Statistics	153
5.4	From Aggregate Behavior to Rare Events	155
5.4.1	The Central Limit Theorem: Weak Form	155
5.4.2	Large Deviations: Tail Form of the CLT	157
5.4.3	When CLT Fails: Heavy Tails and Stable Laws	158
5.4.4	Rare Events in Many Trials: The Poisson Limit	159
5.4.5	Beyond Sums: Extreme Value Theorems	161
6	Entropy and Information Theory	163
6.1	Conditional Probability and Bayes' Rule	164
6.1.1	Conditional Probability, Joint Laws, and Bayes' Rule	164
6.1.2	Discrete Bayes in a Toy Medical Diagnosis Model	165
6.1.3	Exploring Test Quality: Sensitivity and Specificity	166
6.1.4	From Naïve Bayes to Neural Networks	167
6.2	Entropy: Quantifying Uncertainty	170
6.2.1	Definition and Interpretations of Entropy	171
6.2.2	Mutual Information	173
6.2.3	KL Divergence: Comparing Distributions	176
6.2.4	Cross-Entropy and Its Connection to KL Divergence	178
6.2.5	Wasserstein Distance: Geometry of Probability Distributions	180
6.3	Information Theory and Neural Networks	183
6.3.1	Source Coding Theorem (Lossless Compression)	184
6.3.2	Neural Networks as Encoding Schemes	186
6.3.3	Autoencoders and Nonlinear Compression	187
6.3.4	The Information Bottleneck Principle and U-Net as a Nonlinear Compressor	190
6.3.5	Channel Coding Theorem and Its Application to Neural Networks	192
6.3.6	Efficient Memory and Neural Network Storage	194
7	Stochastic Processes	198
7.1	Exact Sampling	199
7.1.1	Inverse Transform Sampling	199
7.1.2	Exact Sampling from Multivariate Distributions via Chain Rule	202
7.2	Importance Sampling and its Applications	203
7.2.1	General Formulation of Importance Sampling	203
7.2.2	Importance Sampling for Posterior Estimation	204
7.2.3	Adaptive Importance Sampling and the Cross-Entropy Method	206
7.3	Diffusion and Brownian Motion	210

7.3.1	Diffusion from Brownian Motion	210
7.3.2	From the Stochastic Differential Equation to the Path Integral	211
7.3.3	Generalization: Diffusion with Drift Induced by a Potential	216
7.4	Markov Chains	220
7.4.1	Arrow of Time and Dynamic Programming	223
7.5	Markov Chains Meet Sampling: MCMC	227
7.6	Beyond Markov via Auto-Regressive Modeling	231
7.6.1	Randomness in Next Token Generation	232
7.6.2	Auto-Regressive Modeling as an Expanding Markov Process	232
8	Energy Based (Graphical) Models	235
8.1	Inference	236
8.1.1	Graphical Models	236
8.1.2	Variational Methods	240
8.1.3	Neural Decoding of Low-Density Parity-Check Codes	245
8.1.4	Variational Auto-Encoders	247
8.2	Learning	251
8.2.1	Likelihood	251
8.2.2	Local Methods: Pseudo-Log-Likelihood and Interaction Screening	256
8.2.3	Restricted Boltzmann Machine	258
8.2.4	From Graphical Models to Graph Neural Networks	262
9	Synthesis	266
9.1	Score-Based Diffusion Models	267
9.1.1	Bridge Diffusion	272
9.2	A Unified View: Generative Models as Diffusions	272
9.2.1	GANs as Implicit Diffusion Models	272
9.2.2	Variational Autoencoders as Diffusion Models	275
9.3	Diffusion Models and Dynamic Phase Transitions	277
9.3.1	U-Turn Diffusion and Memorization Dynamics	277
9.3.2	Dynamic Phase Transitions in High Dimensions	277
9.3.3	Open Problems	278
9.4	From MDP to Reinforcement Learning	278
9.4.1	Markov Decision Processes	279
9.4.2	Reinforcement Learning	281
9.4.3	Physics-Informed and AI-enabled Reinforcement Learning	283
9.4.4	RL and Generative AI	288
9.5	Synthesis of Diffusion and Reinforcement Learning	290
9.5.1	A short pre-history of “integrable” SOC	291
9.5.2	From SOC to Path-Integral Diffusion (PID)	291
9.5.3	Three levels of integrability	292
9.5.4	PID viewed from Reinforcement Learning	293
9.6	Sampling Decisions	294
9.6.1	From <i>Artificial</i> to <i>Physical</i> Time	294
9.6.2	Generative Flow Networks in a Nutshell	294

9.6.3	Decision Flow: an Integrable, Diffusion-like Extension of GFN	295
9.7	Path Forward	299
9.7.1	Work in Progress (by the author) and Open Directions	299
9.7.2	Further Ideas on a Grand Unification of Generative Models	300
9.7.3	Downstream Applications: Where the Mathematics of AI Meets the Real World	301

Introduction to the "Mathematics of Generative AI"

This book aims to build a clear and modern bridge between the rapidly evolving practice of Generative AI and the mathematical principles that underlie it. Its primary audience is *advanced STEM undergraduates, graduate students, and research-minded AI practitioners* who have seen core undergraduate mathematics (linear algebra, differential equations, probability, optimization) but now seek an integrated view of how these ideas power state-of-the-art generative models. Rather than treating mathematics and AI as separate subjects, the book presents them as a single, mutually reinforcing narrative – mathematics illuminated by AI, and AI clarified through mathematics.

What Makes This Book Different?

Most applied mathematics textbooks build upward: fundamentals first, applications last. This book takes a more fluid and strategically timed approach. While we do *not* open with full Generative AI (GenAI) case studies or advanced architectures, we introduce the motivating phenomena of modern GenAI – diffusion models, transformers, high-dimensional representation learning – *as soon as the relevant mathematical groundwork is in place*. In this way, the appearance of each AI example is synchronized with the mathematical ideas that illuminate it. This “application-aligned” route helps the reader maintain a sense of purpose: every mathematical concept is introduced because it is **required** to understand an insight, mechanism, or algorithm that matters today. Since the field of GenAI is a moving target, the book emphasizes not completeness but **relevance**, focusing on mathematical structures that currently shape research frontiers.

A Selective Approach to Mathematical Foundations

Generative AI relies on a remarkably broad mathematical toolkit: linear algebra, optimization, stochastic calculus, variational principles, statistical mechanics, and more. This book does *not* attempt a comprehensive survey. Instead, it takes a *curated, selective approach*, choosing mathematical ideas that directly empower the reader to understand diffusion models, denoising, score matching, autoencoders, transformers, and the stochastic control/transport interpretations that unify many of these themes. The emphasis throughout is on *conceptual clarity* and *practical utility*: What does this piece of mathematics buy us in GenAI? Why does this identity, inequality, or operator show up again and again? Where rigorous treatments would distract from intuition, we point the reader to standard references and keep the exposition focused on insight and mechanism.

Learning by Doing

The book follows a *learning-by-doing* philosophy. Each chapter contains exercises that blend:

- core mathematical manipulations;
- conceptual questions that probe understanding;

- exploratory tasks based on recent AI techniques;
- computational experiments in Python.

Many readers find that intuition for generative models develops most naturally through *experimentation* – visualizing diffusions, probing loss surfaces, inspecting denoisers, and comparing implementations. The book therefore encourages curiosity, creativity, and iterative experimentation as essential complements to theory.

This philosophy is concretely implemented through *tight integration with computational resources*. All theoretical examples, exercises, and the majority of the figures presented in the text are computationally realized via linked Jupyter/Python notebooks. This seamless integration allows the reader to immediately test concepts, visualize complex functions, and modify models. As a continuously evolving resource, the current version of the living book (pdf file), along with the complete collection of notebooks organized by chapter, are maintained and updated on our public repository: <https://github.com/mchertkov/Mathematics-of-Generative-AI-Book>.

Computational Exploration: Scope and Intent

The computational components accompanying many exercises are intentionally lightweight: small, transparent scripts designed to run on an ordinary laptop. Their purpose is to illuminate ideas – not to reproduce production – scale training and inference pipelines. Topics such as distributed training, large-scale optimization, or multi-GPU pipelines are beyond the scope of this book and are better learned in specialized courses or through dedicated self-study. Our computational aim here is to support mathematical insight, not engineering.

Acknowledgments

The December 2025 edition of this living book reflects the insight and generosity of many colleagues, collaborators, and students. Their contributions – ranging from lectures and notes to code and exercises – shaped the material in ways that a single author could not.

Robert Ferrando – a graduate student in Applied Mathematics at the University of Arizona and my co-instructor for Math 496T/Spring 2025 at University of Arizona – deserves special thanks. Robert advised me on countless technical points, delivered two guest lectures, and compiled the complete set of exercise solutions in the Spring of 2025. His insight, enthusiasm, and talent for explaining subtle ideas were indispensable; without his help these notes would not have come to life.

I am equally grateful to my colleagues **Jason Aubrey** and **Arvind Suresh** for sharing their relevant notes and offering timely advice as the material of Math496T evolved. Arvind also contributed two guest lectures on practical neural-network implementation in PYTORCH, enriching the applied component of the course.

Heartfelt thanks go to the **Math 496T students**. Their curiosity, sharp questions, and vigilant spotting of errors shaped the exposition and pushed the project far beyond what a single author could achieve.

AI-assisted tools (including large language models) are also used in the preparation and editing of the text. Their role is acknowledged as part of modern mathematical and computational practice, and their contributions – while carefully curated – help accelerate iteration and broaden perspective.

A Living and Collaborative Resource – Envisioning the Future

This text is deliberately designed as a *living book*, updated through regular teaching cycles, research developments, and feedback from students and colleagues across academia, national laboratories, and industry.

The Mathematics of Generative AI will continue to evolve alongside the field itself. As diffusion models, score-based samplers, reinforcement-learning formulations, and transformer-based architectures continue to advance, new editions of this book will incorporate the emerging mathematics that supports them. Readers are warmly invited to participate by:

- proposing new exercises, illustrations, and computational explorations;
- suggesting clarifications or alternative explanations;
- summarizing recent research papers for inclusion;
- sharing implementations or extensions of the included notebooks.

These who contribute improvements – whether conceptual, computational, or expository – will be acknowledged as collaborators in this ongoing effort. The hope is that this book becomes not just a resource but a community project: a shared effort to understand the mathematical ideas driving the most transformative technological development of our time.

Chapter 1

Linear Algebra (of AI)

1.1 Foundations of Representing Data

1.1.1 Vectors

A vector is an ordered collection of numbers (or elements) that can represent points, directions, or quantities in space. Mathematically, a vector in n -dimensional real space is represented as:

$$v = [v_1, v_2, \dots, v_n]^\top \in \mathbb{R}^n$$

where v_i are the components of the vector; n is dimensionality of the vector; $[v_1, v_2, \dots, v_n]$ is the notation we use for the row-vector; and $^\top$ is the transposition turning raw vector to column vector and vice versa. Later in the text, we will use the same notation, such as v , interchangeably for both a row vector and a column vector. Any potential ambiguities will be explicitly clarified when they arise.

Key Operations:

- **Addition:** $(u + v)_i = u_i + v_i$, u and v are vectors of the same dimensionality
- **Scalar multiplication:** $(cv)_i = cv_i$, where c is a scalar
- **Product:** $uv^\top = \sum_{i=1}^n u_i v_i$, where u and v are co-dimensional vectors
- **Norm:** $\|v\| = \sqrt{vv^\top}$

Vectors are used to represent data points (e.g., pixel intensities in an image, word embeddings in NLP) or transformations (e.g., directions of gradients in optimization).

1.1.2 Matrices: Representing Linear Transformations.

A matrix is a 2D array of numbers that generalizes vectors to multiple dimensions. A matrix $A \in \mathbb{R}^{m \times n}$ can be represented as:

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

Key Operations:

- **Addition:** Element-wise addition (for matrices with the same dimensions) –

$$(A + B)_{ij} = A_{ij} + B_{ij}.$$

- **Element-Wise Multiplication:** The element-wise (Hadamard) product of matrices A and B (with the same dimensions) is denoted by \odot and computed as:

$$(A \odot B)_{ij} = A_{ij}B_{ij}.$$

- **Multiplication:** Matrix-vector and matrix-matrix multiplication –

$$(AB)_{ij} = \sum_{k=1}^n A_{ik}B_{kj}, \quad (Av)_i = \sum_{j=1}^n A_{ij}v_j,$$

where the inner dimensions of A and B must match.

- **Transpose:** The transpose of a matrix is defined as:

$$(A^\top)_{ij} = A_{ji}.$$

- **Inverse:** The inverse of a square and invertible matrix A , denoted as A^{-1} , satisfies:

$$AA^{-1} = A^{-1}A = I,$$

where I is the identity matrix, defined as $I_{ii} = 1$ and $I_{ij} = 0$ for $i \neq j$.

Why do we associate matrices with linear algebra? Because matrices serve as fundamental tools for representing linear transformations in a general form. A $n \times m$ dimensional matrix acting on a m -dimensional vector represents **linear** transformations such as rotations, rescaling, and projections of the vector:

1. Rotation: For example, a 3D rotation matrix around the z -axis by an angle θ (in radians) is given by:

$$R_z(\theta) = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

If we apply the matrix to the unit vector aligned with the x -axis, $R_z(\theta)[1, 0, 0]^\top$, the result is the unit vector rotated by the angle θ in the (x, y) -plane.

2. Re-scaling: A re-scaling transformation adjusts the magnitude of vectors along specified axes. For example, the following matrix rescales vectors in the x - and y -directions by factors s_x and s_y , respectively:

$$S = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}.$$

Applying this matrix to a vector $x = [x, y]^\top$ results in:

$$y = Sx = \begin{bmatrix} s_x x \\ s_y y \end{bmatrix}.$$

For example, if $s_x = 2$ and $s_y = 0.5$, the vector $[1, 2]^\top$ is transformed into:

$$y = \begin{bmatrix} 2 & 0 \\ 0 & 0.5 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}.$$

3. Projection: A projection transformation maps vectors onto a subspace. For example, a projection onto the x -axis in 2D is given by the matrix:

$$P_1 = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}.$$

Applying this matrix to a vector $[x, y]^\top$ gives:

$$y = P_1 x = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x \\ 0 \end{bmatrix}.$$

This operation removes the second y -component of the vector, effectively projecting it onto the first x -axis.

Similarly, projection onto the y -axis can be achieved using:

$$P_2 = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}.$$

Summarizing/rephrasing – any linear map $\mathcal{L} : \mathbb{R}^n \rightarrow \mathbb{R}^m$ can be uniquely expressed in terms of a matrix $A \in \mathbb{R}^{m \times n}$ such that for any vector $x \in \mathbb{R}^n$, the image $\mathcal{L}(x)$ is given by $\mathcal{L}(x) = Ax$. This representation encapsulates the aforementioned essential operations like scaling, rotations, and projections, and is indispensable in various applications, from solving systems of linear equations to encoding (some) transformations in Neural Networks (NN). Moreover, matrices also facilitate iterative processes, such as those encountered in diffusion models, where repeated applications of matrix multiplications are key to de-noising and generating data.

Exercise 1.1.1 (Matrix Multiplication and Elliptical Dynamics). (a) Consider $x_0 = (a, 0)^\top$, a column vector in 2D. Design a 2×2 matrix A such that its repeated application to x_0 :

$$x_t = A^t x_0, \quad t = 1, 2, \dots,$$

results in all x_t lying on an ellipse with a semi-axis ratio of a/b . Here, A^t represents the matrix A raised to the power t , meaning t -fold matrix multiplication.

- (b) Can you ensure that as $t \rightarrow \infty$, the trajectory of x_t covers the entire ellipse? In other words, is it possible to design A such that the process explores all points on the ellipse in the limit of infinite iterations?

1.1.3 Convolution: Bridging Linear Algebra and Applications in AI

Convolution is a fundamental operation in linear algebra and signal processing, playing a pivotal role in many applications, including modern AI architectures. Here, we introduce convolution and demonstrate its relevance in practical settings, particularly its compactness and efficiency compared to other forms of linear transformations.

Convolution combines two inputs f and g , producing an output that reflects their interaction. In a one-dimensional case, for functions f and g defined on \mathbb{R} , the convolution $(f * g)(t)$ is defined as:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t - \tau) d\tau.$$

In discrete settings, such as in linear algebra – where discreteness is represented via indices – convolution is often applied to vectors or matrices.

Convolution should be viewed as a particular form of a linear transformation. Consider the following example: the convolution of the vector $x \in \mathbb{R}^n$ and a kernel (filter) $k \in \mathbb{R}^m$, where $m < n$, resulting in $y \in \mathbb{R}^{n-m+1}$:

$$y_i = \sum_{j=1}^m k_j \cdot x_{i+j-1}, \quad i = 1, 2, \dots, n - m + 1.$$

This operation can be represented as a structured matrix-vector multiplication, where matrix $A(k)$ acts on x to produce y :

$$y = A(k) \cdot x, \quad A(k) = \begin{bmatrix} k_1 & 0 & \cdots & 0 \\ k_2 & k_1 & \cdots & 0 \\ \vdots & k_2 & \ddots & \vdots \\ k_m & \vdots & \cdots & k_1 \\ 0 & k_m & \ddots & \vdots \\ \vdots & 0 & \cdots & k_m \end{bmatrix}.$$

In this formulation, each entry y_i of the output vector y is obtained by computing the dot product of the kernel k with a corresponding window of the input vector x . This emphasizes that convolution is fundamentally a linear transformation that is computationally efficient and compact.

Example 1.1.1 (Convolutional Neural Networks). *Let us discuss a fundamental building block of Convolutional Neural Networks (CNNs): the linear part of a single convolutional layer. While a full CNN architecture typically involves multiple convolutional, pooling, and fully connected layers, here we restrict our discussion to the operation of a single convolutional layer to illustrate its role in feature extraction.*

Consider an input image $x \in \mathbb{R}^{n \times n}$. A convolutional filter $k \in \mathbb{R}^{m \times m}$ slides across x , producing the output:

$$y_{i,j} = \sum_{h,w=1}^m k_{h,w} \cdot x_{i+h-1,j+w-1}, \quad i,j = 1, \dots, n-m+1.$$

Example 1.1.2 (Image to Tensor). Consider a grayscale image $x \in \mathbb{R}^{4 \times 4}$ and a filter $k \in \mathbb{R}^{2 \times 2}$:

$$x = \begin{bmatrix} 1 & 2 & 3 & 0 \\ 4 & 5 & 6 & 1 \\ 7 & 8 & 9 & 0 \\ 1 & 0 & 2 & 3 \end{bmatrix}, \quad k = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}.$$

The convolution of x with the filter k becomes:

$$\begin{aligned} y &= \begin{bmatrix} (1 \cdot 1 + 2 \cdot 0 + 4 \cdot 0 + 5 \cdot -1) & (2 \cdot 1 + 3 \cdot 0 + 5 \cdot 0 + 6 \cdot -1) & (3 \cdot 1 + 0 \cdot 0 + 6 \cdot 0 + 1 \cdot -1) \\ (4 \cdot 1 + 5 \cdot 0 + 7 \cdot 0 + 8 \cdot -1) & (5 \cdot 1 + 6 \cdot 0 + 8 \cdot 0 + 9 \cdot -1) & (6 \cdot 1 + 1 \cdot 0 + 9 \cdot 0 + 0 \cdot -1) \\ (7 \cdot 1 + 8 \cdot 0 + 1 \cdot 0 + 0 \cdot -1) & (8 \cdot 1 + 9 \cdot 0 + 0 \cdot 0 + 2 \cdot -1) & (9 \cdot 1 + 0 \cdot 0 + 2 \cdot 0 + 3 \cdot -1) \end{bmatrix} \\ &= \begin{bmatrix} -4 & -4 & 2 \\ -4 & -4 & 6 \\ 7 & 6 & 6 \end{bmatrix}. \end{aligned}$$

We can also re-state this operation as a linear transformation without index shifts:

$$\begin{aligned} y_{i,j} &= \sum_{i',j'=1}^4 A_{i,j,i',j'} x_{i',j'}, \quad i,j = 1, \dots, 3, \\ A_{i,j,i',j'} &= \begin{cases} k_{i'-i+1,j'-j+1}, & \text{if } 1 \leq i' - i + 1 \leq m \text{ and } 1 \leq j' - j + 1 \leq m, \\ 0, & \text{otherwise.} \end{cases} \end{aligned}$$

Two remarks are in order. First, notice that the number of operations required to express elements of matrix y via matrix x in the form of a linear transformation is larger than in the form of a convolution (4 vs. 2). This emphasizes that convolution, when possible, can be a much more compact representation than a general linear transformation.

Second, A in the last formula is an example of a new type of object – a tensor – which we will discuss in the next subsection.

1.1.4 Tensors: The Generalization

The word "tensor" originates from the Latin word tendere, which means "to stretch." The term was introduced into mathematics and physics to describe objects that generalize the notion of scalars, vectors, and matrices to higher dimensions – we also call it rank or order – capturing properties related to "stretching" or "deformation."

A tensor \mathcal{T} of rank k can be represented as:

$$\mathcal{T} \in \mathbb{R}^{d_1 \times d_2 \times \dots \times d_k}$$

where d_i is the size along the i -th dimension.

Rank:

- Scalars: Rank 0.
- Vectors: Rank 1.
- Matrices: Rank 2.
- Higher-rank tensors: $k \geq 3$.

Representation

Tensors are particularly powerful tools for representing multi-dimensional data. The following example and exercise demonstrates their application in representing RGB images, video clips, and NLP embeddings.

Machine Learning Tensors vs. Mathematical Tensors

The word *tensor* is used in two different ways in mathematics, physics, and machine learning. Because this book moves between formal linear algebra and practical implementations (NumPy, PyTorch, JAX), it is important to clarify the distinction.

1. Classical Tensors (Multilinear Algebra). In its classical mathematical meaning, a tensor is a *multilinear object*:

$$T \in V^{\otimes r} \otimes (V^*)^{\otimes s}.$$

A rank- (r, s) tensor transforms according to specific rules under changes of basis. Examples include scalars, vectors, covectors, bilinear forms, stress tensors, and curvature tensors.

The defining characteristic is:

A mathematical tensor obeys coordinate transformation laws.

2. Tensors in Machine Learning (Multi-Index Arrays). In modern ML libraries (NumPy, PyTorch, TensorFlow, JAX), a *tensor* is simply a **numerical array** with shape

$$(d_1, d_2, \dots, d_k).$$

No geometric structure or transformation law is assumed. For example: - Images: $\mathbb{R}^{H \times W \times 3}$, - Video clips: $\mathbb{R}^{H \times W \times 3 \times T}$, - Embedding batches: $\mathbb{R}^{L \times D}$, are all called “tensors” in the ML sense.

Thus:

In ML practice: tensor = multi-dimensional numerical array.

3. Why This Distinction Matters Here. Throughout this book, unless clearly indicated otherwise, the word *tensor* refers to the ML meaning (a multi-index array). This is the natural and useful notion for: - image and video data representations, - neural-network computations, - attention mechanisms, - batch processing and automatic differentiation.

The classical, multilinear-algebraic notion reappears later when discussing continuous-time models, differential operators, or geometry-driven aspects of diffusion processes.

Example 1.1.3. Representing RGB Images: Theory and Implementation

RGB images are commonly represented as tensors in $\mathbb{R}^{H \times W \times 3}$, where:

- H : Height of the image (number of rows),
- W : Width of the image (number of columns),
- 3: The three color channels: Red, Green, and Blue.

For instance, a color image of size 256×256 may be represented as a tensor

$$\mathcal{T} \in \mathbb{R}^{256 \times 256 \times 3}.$$

Each pixel channel typically stores an 8-bit intensity value between 0 and 255. Figure 1.1 illustrates the three views used in this exercise: the original grayscale MNIST image, its RGB version obtained via a colormap, and the extracted green channel whose average intensity we compute below.

Theoretical Formulation.

- To extract the green channel (second slice along the third dimension):

$$\mathcal{T}_{\text{green}} = \mathcal{T}[:, :, 2].$$

- To compute the average green-channel intensity:

$$\text{avg_green} = \frac{1}{HW} \sum_{i=1}^H \sum_{j=1}^W \mathcal{T}_{i,j,2}.$$

See the accompanying Jupyter notebook `Ex1-1-1image.ipynb` for the full computational workflow.

Exercise 1.1.2 (Tensor Representation of Video Clips and NLP Embeddings). In this exercise we extend the tensor viewpoint of Example 1.1.3 (RGB images) to (1) videos represented as four-dimensional arrays and (2) sequences of word embeddings represented as matrices. Throughout Chapter 1, the term “tensor” refers to a multidimensional numerical array used in applied machine learning practice – not necessarily to a multi-linear-algebra tensor.

1. Videos as 4D tensors: $\mathbb{R}^{H \times W \times 3 \times T}$

A video of resolution 1920×1080 , recorded at 30 fps for 10 seconds, may be represented as

$$\mathcal{T} \in \mathbb{R}^{1080 \times 1920 \times 3 \times 300},$$

where $\mathcal{T}_{i,j,c,t}$ denotes the intensity of pixel location (i, j) , color channel $c \in \{1, 2, 3\}$, in frame t .

Exercise 1.1.1: RGB representation and green channel

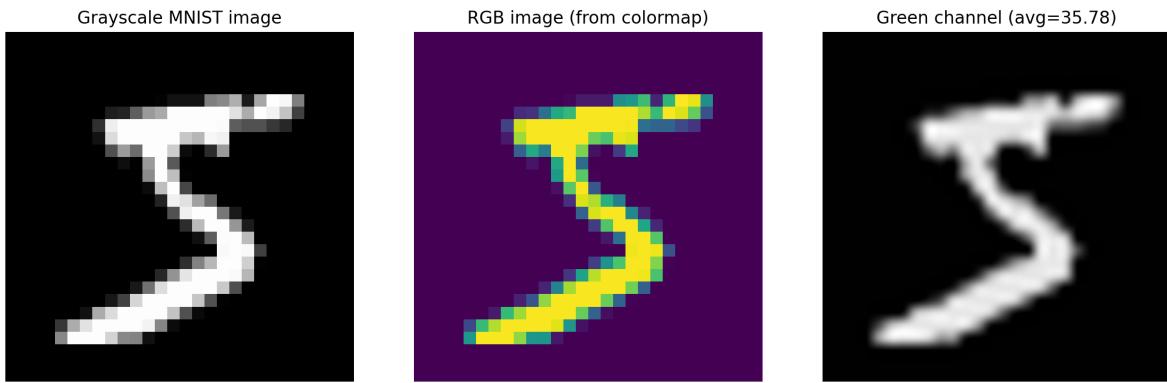


Figure 1.1: Three-panel illustration for Example 1.1.1. **Left:** Original MNIST grayscale image. **Middle:** RGB image obtained via a colormap. **Right:** Extracted green channel, annotated with its average intensity. These views illustrate the tensor representation of images and motivate the extraction-and-averaging operations defined in the exercise.

Pixel Intensity. Two common summaries of RGB brightness are:

$$I_{\text{grayscale}} = 0.2989R + 0.5870G + 0.1140B, \quad I_{\text{average}} = \frac{R + G + B}{3}.$$

Tasks:

- (a) Propose a formula for the mean grayscale intensity over all pixels and all frames whose timestamps lie in the interval $[t_1, t_2]$. (Assume frame t corresponds to time $t/30$ seconds.)
- (b) How would this mean intensity change if only the green channel G is used, instead of the grayscale combination of R, G, B ?

2. **Word embeddings as 2D and 3D tensors:** $\mathbb{R}^{D \times L}$

A sentence of length L may be represented by a matrix of word embeddings $\mathcal{T} \in \mathbb{R}^{D \times L}$, obtained by multiplying an embedding matrix $E \in \mathbb{R}^{D \times V}$ by one-hot word encodings. For example, if “cat” is the third word in the vocabulary, then

$$\text{Embedding(cat)} = E_{:,3}.$$

Tasks:

- (a) Write a formula for the weighted average embedding

$$e_{\text{avg}} = \sum_{i=1}^L w_i \mathcal{T}_{:,i}, \quad w_i \geq 0, \quad \sum_{i=1}^L w_i = 1.$$

- (b) Suggest how to choose the weight vector w to emphasize a particular position in the sequence (e.g., the first or last word).

(c) Suppose the weights depend on embedding dimension d and position i , yielding weights $w_{d,i}$. How does the formula for the weighted embedding change?

Note. Only theoretical reasoning and formulas are required; no code is needed.

Tensor Operations: Direct Product and Contraction-Based Product

Tensors serve as versatile tools for representing and manipulating multi-dimensional data. In this subsection, we focus on two fundamental tensor operations: the **direct product**, which creates higher-rank tensors, and the **contraction-based product**, which generalizes matrix multiplication by summing over shared indices. These operations are widely used in applications ranging from physics to modern AI.

The **direct product**, or tensor product, combines two tensors to produce a higher-rank tensor. Let $\mathcal{T}_1 \in \mathbb{R}^{d_1 \times \dots \times d_k}$ and $\mathcal{T}_2 \in \mathbb{R}^{d_{k+1} \times \dots \times d_{k+m}}$. Their direct product $\mathcal{T}_1 \otimes \mathcal{T}_2 \in \mathbb{R}^{d_1 \times \dots \times d_k \times d_{k+1} \times \dots \times d_{k+m}}$ is defined element-wise as:

$$(\mathcal{T}_1 \otimes \mathcal{T}_2)_{i_1, \dots, i_{k+m}} = (\mathcal{T}_1)_{i_1, \dots, i_k} \cdot (\mathcal{T}_2)_{i_{k+1}, \dots, i_{k+m}}.$$

The rank of the resulting tensor is the sum of the ranks of the input tensors, making it a powerful operation for constructing multi-linear expressions.

The **contraction-based product** reduces the rank of the resulting tensor by summing over one or more shared indices between the input tensors. This operation generalizes the familiar matrix product. For example, if $\mathcal{T}_1 \in \mathbb{R}^{d_1 \times d_2 \times d_3}$ and $\mathcal{T}_2 \in \mathbb{R}^{d_3 \times d_4}$, contracting the third index of \mathcal{T}_1 with the first index of \mathcal{T}_2 gives:

$$\mathcal{T}_{i_1, i_2, i_4} = \sum_{i_3=1}^{d_3} (\mathcal{T}_1)_{i_1, i_2, i_3} \cdot (\mathcal{T}_2)_{i_3, i_4}.$$

This operation reduces the rank of the resulting tensor by the number of contracted indices.

Examples of Tensor Operations:

- **Direct Product of Vectors:** Consider two vectors $u \in \mathbb{R}^n$ and $v \in \mathbb{R}^m$. Their direct product is a rank-2 tensor (or matrix):

$$u \otimes v = A \in \mathbb{R}^{n \times m}, \quad A_{ij} = u_i v_j.$$

This operation constructs a bi-linear representation of the two vectors without reducing dimensionality.

- **Contraction with a Vector:** Given a rank-3 tensor $\mathcal{T} \in \mathbb{R}^{d_1 \times d_2 \times d_3}$ and a vector $v \in \mathbb{R}^{d_3}$, contracting the third index of \mathcal{T} with v yields:

$$\mathcal{M}_{i_1, i_2} = \sum_{i_3=1}^{d_3} \mathcal{T}_{i_1, i_2, i_3} \cdot v_{i_3}.$$

The result is a rank-2 tensor (or matrix). For instance, if \mathcal{T} represents time-series data across multiple sensors, contracting with v can aggregate measurements across channels.

- **Contraction with a Matrix:** Let $A \in \mathbb{R}^{d_3 \times d_4}$. Contracting the third index of \mathcal{T} with the first index of A gives:

$$\mathcal{T}'_{i_1, i_2, i_4} = \sum_{i_3=1}^{d_3} \mathcal{T}_{i_1, i_2, i_3} \cdot A_{i_3, i_4}.$$

Here, A acts as a linear transformation on the third dimension of \mathcal{T} .

Here are some worked examples of tensor operations.

Example 1.1.4 (Direct Product of Vectors). *Let \mathbf{u} and \mathbf{v} be two vectors:*

$$\mathbf{u} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \quad \mathbf{v} = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$$

The **direct product** (or *tensor product*) $\mathbf{u} \otimes \mathbf{v}$ results in a higher-dimensional tensor, which in this case is a 2×2 matrix. According to the definition of the direct product, each element of the resulting tensor is formed by multiplying each element of \mathbf{u} with each element of \mathbf{v} :

$$\mathbf{u} \otimes \mathbf{v} = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \otimes \begin{bmatrix} 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 1 \cdot 3 & 1 \cdot 4 \\ 2 \cdot 3 & 2 \cdot 4 \end{bmatrix} = \begin{bmatrix} 3 & 4 \\ 6 & 8 \end{bmatrix}$$

Thus, the result of the direct product is a 2×2 matrix:

$$\begin{bmatrix} 3 & 4 \\ 6 & 8 \end{bmatrix}$$

This is a rank-2 tensor, where the dimensions correspond to the input vectors \mathbf{u} and \mathbf{v} .

Example 1.1.5 (Direct Product of Matrices). *Let A and B be two matrices:*

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \quad B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

The direct product $A \otimes B$ is a higher-order tensor in $\mathbb{R}^{2 \times 2 \times 2 \times 2}$, whose (i, j, k, l) entry is the (i, j) entry of A times the (k, l) entry of B .

Example 1.1.6 (Contraction of Two Matrices). *Let's take two matrices A and B :*

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}, \quad B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

The tensor contraction (matrix multiplication) is done by summing over the common index k :

$$A \cdot B = \sum_k a_{ik} b_{kj}$$

So, for matrices, this looks like:

$$A \cdot B = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix}$$

Example 1.1.7 (Contraction on Higher-Dimensional Tensors). *Let's consider two tensors T and S :*

$$T_{ijk} = \begin{bmatrix} T_{111} & T_{112} \\ T_{121} & T_{122} \end{bmatrix}, \quad S_{klm} = \begin{bmatrix} S_{111} & S_{112} \\ S_{121} & S_{122} \end{bmatrix}$$

The contraction of T and S over the index k is:

$$T_{ij} \circ S_{jlm} = \sum_k T_{ijk} S_{klm}$$

Thus, the result of contracting along the index k will produce a new tensor with indices i , j , and m . The resulting tensor is:

$$T \circ S = \begin{bmatrix} T_{111}S_{111} + T_{112}S_{211} & T_{111}S_{112} + T_{112}S_{212} \\ T_{121}S_{111} + T_{122}S_{211} & T_{121}S_{112} + T_{122}S_{212} \end{bmatrix}$$

To see how to perform these tensor operations in Python using the `torch` package, please refer to the Jupyter notebook `TensorOperations.ipynb`.

Exercise 1.1.3 (Einstein Summation and Computational Efficiency). *Einstein Summation Notation (ESN) provides a compact, index-based way to express tensor operations, directly translating into efficient GPU kernels. This exercise tests your ability to translate between traditional summation and ESN, and addresses core efficiency concerns in modern deep learning hardware.*

1. **ESN Translation:** Express the following two fundamental operations using Einstein Summation Notation:

- (1) The matrix-vector product $y = Ax$, where $A \in \mathbb{R}^{m \times n}$ and $x \in \mathbb{R}^n$.
- (2) The squared Frobenius norm of a matrix A : $\|A\|_F^2 = \sum_{i=1}^m \sum_{j=1}^n A_{ij}^2$.

2. **Contraction Dimensionality:** Given the tensor contraction $C_{ikl} = A_{ij}B_{jkl}$, assuming all dimensions are D :

- What is the rank (number of indices) of the resulting tensor C ?
- What is the final dimensionality of C (e.g., $\mathbb{R}^{d_1 \times d_2 \times \dots}$)?

3. **Hardware Efficiency:** In GPU programming, why is **memory contiguity** (how elements are stored and accessed in memory) often more important for overall performance than the theoretical Floating-point OPeration count (FLOPs)?

To check your ESN logic against practical implementation, refer to the Jupyter notebook `TensorOperations.ipynb`.

Convolution vs Tensor Operations

While convolution, discussed above in Section 1.1.3, is sometimes conflated with the tensor product or contraction, it has distinct characteristics:

- **Tensor Product:** Combines two tensors $\mathcal{T}_1 \in \mathbb{R}^{d_1 \times d_2}$ and $\mathcal{T}_2 \in \mathbb{R}^{d_3 \times d_4}$ into a higher-dimensional tensor $\mathcal{T}_1 \otimes \mathcal{T}_2 \in \mathbb{R}^{d_1 \times d_2 \times d_3 \times d_4}$.
- **Contraction:** Reduces the dimensionality of tensors by summing over shared indices, e.g., the matrix product AB contracts indices of $A \in \mathbb{R}^{n \times m}$ and $B \in \mathbb{R}^{m \times p}$ to yield $C \in \mathbb{R}^{n \times p}$.
- **Convolution:** Applies a kernel in a sliding window manner, emphasizing locality and weight sharing.

From Data Structures to Architecture: The Role of Tensors

We have established the core building blocks of data representation: vectors, matrices for transformations, and tensors as the high-dimensional generalization. All Generative AI models, from simple CNNs to complex Transformers, operate exclusively on these tensor structures. The next step is to see how these elements – specifically the concepts of vector projection and attention – are woven together to create the complex, multi-layered mechanisms in the *Transformer* model.

1.1.5 Applications in Generative AI – Mechanics of Transformers

Vector Representations and Matrix Transformations in Generative Diffusion Models

In generative diffusion models, data points are represented as vectors and undergo a sequence of matrix transformations to progressively refine noisy inputs into meaningful outputs, such as generating new images. This iterative de-noising process can be described as:

$$x_{t+1} = A_t x_t + b_t, \quad t = 0, \dots, T,$$

where:

- x_0 is the noisy input that initializes the de-noising process.
- x_T is the final generated image or output.
- $A_t \in \mathbb{R}^{n \times n}$ is a transformation matrix.
- $b_t \in \mathbb{R}^n$ is a bias term.

The transformation matrix A_t and bias term b_t are nonlinear functions of x_t and t , often incorporating stochastic components such as Wiener noise and Neural Networks (NNs). These parameters are learned during training to effectively model the complex dynamics of the de-noising process. This framework enables the generation of high-quality synthetic data by iteratively refining the noise over time.

High-Dimensional Feature Interactions with Tensors in Transformers:

The transformer architecture, introduced in 2017 by Vaswani et al. in their seminal work 'Attention is All You Need' [1], revolutionized AI by leveraging self-attention mechanisms and feed-forward layers to model complex dependencies across input sequences. For a detailed and intuitive explanation of the underlying principles, readers are referred to the online tutorial <https://jalammar.github.io/illustrated-transformer/>, which provides visual insights into the architecture.

Transformers are foundational in modern generative AI, leveraging tensors to model dependencies across tokens in a sequence. These tokens are processed through a combination of linear tensor operations (index contractions, including convolutions) and nonlinear functions to predict the next token in a sequence.

Tokens and Their Role in Sequence Generation: The input sequence is represented as a matrix $X = \{t_1, t_2, \dots, t_n\} \in \mathbb{R}^{n \times d}$, where n is the sequence length, $t_i \in \mathbb{R}^d$ is the embedding of the i -th token, and d is the embedding dimension. The process of predicting the next token t_{n+1} involves evolving a "token-to-be-predicted" vector $\hat{t}_{n+1} \in \mathbb{R}^d$, which stabilizes over iterative applications of the transformer mechanism.

1. **Embedding and Initialization:** The vector $\hat{t}_{n+1}^{(0)}$ is initialized, typically as the "average" of the embeddings:

$$\hat{t}_{n+1}^{(0)} = \frac{1}{n} \sum_{i=1}^n t_i, \quad \hat{t}_{n+1}^{(0)} \in \mathbb{R}^d.$$

2. **Attention Mechanism and Update:** The matrix of known tokens $X \in \mathbb{R}^{n \times d}$ interacts with $\hat{t}_{n+1}^{(k)} \in \mathbb{R}^d$ through the self-attention mechanism. At each step k , the matrices of queries (Q), keys (K), and values (V) are computed as:

$$Q = \hat{t}_{n+1}^{(k)} W_Q, \quad K = X W_K, \quad V = X W_V,$$

where $W_Q, W_K, W_V \in \mathbb{R}^{d \times d}$ are learned weight matrices, and $\hat{t}_{n+1}^{(k)}$ should be treated as a row vector (check the dimensionality of the vector-matrix product to convince yourself of this). Here, $Q \in \mathbb{R}^d$ is the query vector for the token-to-be-predicted; $K, V \in \mathbb{R}^{n \times d}$ are the key and value matrices for the known tokens.

The attention weights are calculated using the softmax function as:

$$\alpha_i^{(k)} = \text{softmax} \left(\frac{Q \cdot K}{\sqrt{d}} \right)_i := \frac{\exp \left(\frac{QK_{i,:}^\top}{\sqrt{d}} \right)}{\sum_{j=1}^n \exp \left(\frac{QK_{j,:}^\top}{\sqrt{d}} \right)} \in [0, 1],$$

where $K_{i,:}$ denotes the i th row of K , which is transposed to become the i th column of K^\top , and appears as a column vector in the computation of $\alpha_i^{(k)}$. ¹ These weights

¹The softmax in computing $\alpha_i^{(k)}$ may also be applied by computing QK^\top/\sqrt{d} first, and then applying $e^{x_i}/\sum_i e^{x_i}$ to each component of the resulting vector.

represent the importance of each known token t_i in predicting t_{n+1} . The aggregated output is obtained by contracting the attention weights with the value vectors:

$$z^{(k)} = \sum_{i=1}^n \alpha_i^{(k)} V_i, \quad z^{(k)} \in \mathbb{R}^d.$$

Note that the attention mechanism – arguably the most important part of the transformer construction – incorporates both linear transformations and nonlinear operations, enabling the model to capture complex relationships in the data.

- 3. Nonlinear Transformation + Normalization:** The aggregated output $z^{(k)}$ undergoes a nonlinear transformation to update $\hat{t}_{n+1}^{(k)}$:

$$\hat{t}_{n+1}^{(k+1)} = \text{LayerNorm}(\sigma(W_2\sigma(W_1 z^{(k)}) + b)),$$

where:

- $W_1, W_2 \in \mathbb{R}^{d \times d}$ are learned weight matrices,
- $b \in \mathbb{R}^d$ is a bias vector,
- $\sigma(\cdot)$ is a point-wise activation function, such as Gaussian Error Linear Unit (GELU), defined as:

$$\sigma(x) = x \cdot \Phi(x),$$

where $\Phi(x) = \frac{1}{2} \left(1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right) \right)$ is the cumulative distribution function of a standard Gaussian.

Layer Normalization ensures stability during iterations and is defined as:

$$\text{LayerNorm}(x) = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} \odot \gamma + \beta,$$

where:

- $\mu = \frac{1}{d} \sum_{i=1}^d x_i$ is the mean of the input x ,
- $\sigma^2 = \frac{1}{d} \sum_{i=1}^d (x_i - \mu)^2$ is the variance of the input,
- $\gamma, \beta \in \mathbb{R}^d$ are learnable parameters for scaling and shifting,
- ϵ is a small constant to ensure numerical stability,
- \odot denotes element-wise multiplication.

This combination of nonlinear transformations and normalization ensures that the model learns stable and expressive representations at each iteration.

- 4. Multi-Head Attention:** To capture diverse patterns in the input data, transformers employ multi-head attention, which divides the embeddings into multiple subspaces (or heads), allowing the model to focus on different aspects of the data simultaneously.

The following construct serves as a unified extension of steps 1–3 described above, adapted for the case where multi-head attention is applied.

Each head independently computes attention, and the outputs are concatenated and projected back into the original dimension:

$$Q_h = \hat{t}_{n+1}^{(k)} W_Q^h, \quad K_h = X W_K^h, \quad V_h = X W_V^h, \quad h = 1, \dots, H,$$

where H is the number of attention heads, $Q_h, K_h, V_h \in \mathbb{R}^{n \times d_h}$, and $W_Q^h, W_K^h, W_V^h \in \mathbb{R}^{d \times d_h}$ are learned weight matrices for head h . The dimension of each head is $d_h = \frac{d}{H}$.

The attention weights for each head are computed as:

$$\alpha_{i,h}^{(k)} = \text{softmax}_i \left(\frac{Q_h K_h^\top}{\sqrt{d_h}} \right), \quad \alpha_{i,h}^{(k)} \in [0, 1],$$

where $\alpha_{i,h}^{(k)}$ is component of a tensor representing the importance (or attention) of the i -th token in the sequence relative to the query vector for head h . Here, $Q_h \in \mathbb{R}^{1 \times d_h}$ and $K_h \in \mathbb{R}^{n \times d_h}$ interact to produce $\alpha_h^{(k)} \in \mathbb{R}^{n \times 1}$, encoding relationships between the query and all tokens in the sequence.

The transition from the attention weights $\alpha_h^{(k)}$ to the aggregated output $z_h^{(k)}$ can be understood as a **tensor embedding process**:

$$z_h^{(k)} = \sum_{i=1}^n \alpha_{i,h}^{(k)} V_{h,i}, \quad z_h^{(k)} \in \mathbb{R}^{d_h}.$$

Here, $V_h \in \mathbb{R}^{n \times d_h}$ is the value matrix, and each $V_{h,i} \in \mathbb{R}^{d_h}$ represents the embedding of the i -th token for head h . The operation combines the attention weights $\alpha_{i,h}^{(k)}$ with the embeddings $V_{h,i}$, effectively projecting the sequence-level relationships (encoded in $\alpha_h^{(k)}$) back into the token embedding space of dimension d_h . This embedding process embeds **token relationships** as weighted contributions to the output vector, thereby incorporating context into the representation.

The final multi-head attention output is obtained by concatenating the outputs from all heads:

$$z^{(k)} = \text{Concat}(z_1^{(k)}, z_2^{(k)}, \dots, z_H^{(k)}) W_O \in \mathbb{R}^d, \quad W_O \in \mathbb{R}^{(d_1+d_2+\dots+d_H) \times d}.$$

Here, $\text{Concat}(\cdot) \in \mathbb{R}^{d_1+d_2+\dots+d_H}$ is projected back to the original embedding dimension d via W_O . This ensures that the multi-head attention captures diverse contextual features across multiple subspaces of the token embeddings while preserving the original dimensionality of the data.

After the multi-head attention mechanism computes the aggregated output $z^{(k)}$, it undergoes a nonlinear transformation (see above) to refine and update the representation of the token-to-be-predicted, $\hat{t}_{n+1}^{(k+1)}$.

5. Convergence and Stabilization: The iterations in k proceed until $\hat{t}_{n+1}^{(k)}$ converges to a stable vector $\hat{t}_{n+1}(X)$. The stabilized vector is then used to compute the probabilities of the next token over the vocabulary:

$$p(t_{n+1}|X) = \text{softmax}(W_{\text{out}}\hat{t}_{n+1}(X) + b_{\text{out}}),$$

where $W_{\text{out}} \in \mathbb{R}^{d \times v}$ projects the embedding to the vocabulary size v (enumerating all valid tokens); and $b_{\text{out}} \in \mathbb{R}^v$ is bias vector.

Exercise 1.1.4 (Multi-Head Attention with Prescribed Weights and Ternary Alphabet). Consider a sequence of tokens $\{t_1, t_2, t_3\}$ embedded as

$$X = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \in \mathbb{R}^{3 \times 2},$$

where $n = 3$ (sequence length) and $d = 2$ (embedding dimension). The tokens t_1, t_2, t_3 correspond to three states in the vocabulary $\mathcal{A} = \{a, b, c\}$, with:

$$t_1 \rightarrow a, \quad t_2 \rightarrow b, \quad t_3 \rightarrow c.$$

These embeddings represent features associated with each state and will guide the computation of the next token in the sequence, t_4 . Assume the transformer has two, $H = 2$, attention heads with the following prescribed weights:

$$\begin{aligned} W_Q^1 &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, & W_K^1 &= \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, & W_V^1 &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \\ W_Q^2 &= \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, & W_K^2 &= \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}, & W_V^2 &= \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}. \end{aligned}$$

The token-to-be-predicted $\hat{t}_4 \in \mathbb{R}^2$ is initialized as:

$$\hat{t}_4^{(0)} = \frac{1}{3}(t_1 + t_2 + t_3) = \begin{bmatrix} 3 \\ 4 \end{bmatrix}.$$

1. Query, Key, and Value Computation: Compute the query vector Q_h , key matrix K_h , and value matrix V_h for each head $h = 1, 2$ using:

$$Q_h = \hat{t}_4^{(0)} W_Q^h, \quad K_h = X W_K^h, \quad V_h = X W_V^h.$$

2. Attention Weights: Compute the attention weights $\alpha_{i,h}$ for each token $i = 1, 2, 3$ and each head $h = 1, 2$ using:

$$\alpha_{i,h} = \text{softmax}_i \left(\frac{Q_h \cdot K_h^\top}{\sqrt{d_h}} \right), \quad \text{where } d_h = d/H = 1.$$

3. **Aggregated Output per Head:** Compute the aggregated output $z_h^{(0)}$ for each head $h = 1, 2$ using:

$$z_h^{(0)} = \sum_{i=1}^n \alpha_{i,h} V_{h,i}.$$

4. **Multi-Head Attention Output:** Concatenate $z_1^{(0)}$ and $z_2^{(0)}$, and project back to the original dimension $d = 2$ using:

$$z^{(0)} = \text{Concat}(z_1^{(0)}, z_2^{(0)}) W_O, \quad W_O = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

5. **Nonlinear Transformation:** Verify the updated value of $\hat{t}_4^{(1)}$ after applying the non-linear transformation:

$$\hat{t}_4^{(1)} = \text{LayerNorm} \left(\text{ReLU} \left(W_2 \text{ReLU}((W_1 z^{(0)})^\top) + b \right) \right),$$

where $W_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$, $W_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$, $b = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$, and $\text{ReLU}(x) = \max(0, x)$ (ReLU activation). When computing LayerNorm, assume that γ and β are constant vectors of all 1's and all 0's, respectively.

6. **Stabilization:** Perform parts 1-5 once more on $\hat{t}_4^{(1)}$ and compute the "stabilized" \hat{t}_4 . (Actual stabilization of $\hat{t}_4^{(k)}$ to \hat{t}_4 would happen at $k \rightarrow \infty$.)

7. **Probabilistic Step:** Recall that the vocabulary (alphabet) is ternary – $\mathcal{A} = \{a, b, c\}$ – and compute the probabilities of the next token using the softmax function:

$$\begin{aligned} p(t_4 = a | t_1, t_2, t_3) &= \frac{\exp((W_{out}\hat{t}_4)_1)}{\sum_{j=1}^3 \exp((W_{out}\hat{t}_4)_j)}, \\ p(t_4 = b | t_1, t_2, t_3) &= \frac{\exp((W_{out}\hat{t}_4)_2)}{\sum_{j=1}^3 \exp((W_{out}\hat{t}_4)_j)}, \\ p(t_4 = c | t_1, t_2, t_3) &= \frac{\exp((W_{out}\hat{t}_4)_3)}{\sum_{j=1}^3 \exp((W_{out}\hat{t}_4)_j)}, \end{aligned}$$

where

$$W_{out} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & -1 \end{bmatrix} \in \mathbb{R}^{3 \times 2}.$$

Compute these probabilities explicitly and explain how the output distribution depends on the attention mechanism and token embeddings.

This exercise demonstrates the application of multi-head attention with explicit tensor operations and a ternary vocabulary, showcasing the transformer's ability to model dependencies across tokens.

Pivoting from Representation to Structure: The Power of Decompositions

Section 1.1 focused on defining the algebraic objects – vectors, matrices, and tensors – that *represent* data and *execute* core AI operations like convolution and attention. However, simply representing data is not enough. To compress, analyze, and gain insight into the geometric structure of this high-dimensional data, we need powerful tools. Matrix decompositions, specifically the Singular Value Decomposition (SVD) and Eigen-Decomposition (ED), provide the mathematical framework for breaking down complex matrices into simpler, interpretable components, which is the focus of Section 1.2.

1.2 Matrix Decompositions

1.2.1 Singular Value Decomposition

Singular Value Decomposition (SVD) is a powerful linear algebra technique that can be applied to a batch of data points to extract a reduced-dimensional representation for individual data points. It works by identifying principal directions of variation in the data and allows projecting each data point onto a smaller set of basis vectors while retaining most of the relevant information.

Consider a dataset represented as a matrix $X \in \mathbb{R}^{n \times d}$, where n is the number of data points and d is the dimensionality of each data point. We assume that $n > d$. The Singular Value Decomposition (SVD) of X is given by:

$$X = USV^\top,$$

where:

- $U \in \mathbb{R}^{n \times n}$ is the left singular matrix, whose columns form an orthonormal basis for the space spanned by the rows of X .
- $S \in \mathbb{R}^{n \times d}$ is a diagonal matrix containing the singular values, which indicate the significance of the corresponding directions.
- $V \in \mathbb{R}^{d \times d}$ is the right singular matrix, whose columns form an orthonormal basis for the space spanned by the columns of X .

Both U and V are orthogonal matrices, meaning $U^\top = U^{-1}$ and $V^\top = V^{-1}$, and therefore they are invertible.

Steps to Compute SVD:

The key property of the SVD is that the columns of V (right singular vectors) are the eigenvectors of $X^\top X$, and the squares of the singular values σ_i^2 are the eigenvalues of $X^\top X$:

$$X^\top X = VS^\top U^\top USV^\top = VS^\top SV^\top$$

$$= V \begin{bmatrix} \sigma_1 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & \sigma_2 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_d & 0 & \cdots & 0 \end{bmatrix} \begin{bmatrix} \sigma_1 & 0 & \cdots & 0 \\ 0 & \sigma_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_d \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \end{bmatrix} V^\top = V \text{diag}(\sigma_1^2, \dots, \sigma_d^2) V^\top,$$

where we took into account the orthogonality of U .

Values of the diagonal matrix are conventionally ordered in **descending** order, with the largest singular values appearing first:

$$\sigma_1^2 \geq \cdots \geq \sigma_d^2.$$

Since $X^\top X \in \mathbb{R}^{d \times d}$ is symmetric and positive semi-definite (as it is constructed as the product of a matrix and its transpose), it has real, non-negative eigenvalues $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_d \geq 0$ and an orthonormal set of eigenvectors v_1, v_2, \dots, v_d ². Therefore, by computing the eigenvalues and eigenvectors of $X^\top X$, we can directly obtain the singular values ($\sigma_i = \sqrt{\lambda_i}$) and the right singular vectors (the columns of V).

Once the singular values and V are determined, the left singular vectors (columns of U) can be computed using the following relation:

$$XV = US.$$

Expanding this for each singular vector v_i of V :

$$Xv_i = \sigma_i u_i,$$

where σ_i is the singular value corresponding to v_i , and u_i is the left singular vector. Thus:

$$u_i = \frac{1}{\sigma_i} Xv_i \quad \text{for } \sigma_i \neq 0.$$

The matrix U can then be constructed by normalizing u_i for all i .

For singular values corresponding to $\sigma_i = 0$, the remaining columns of U can be completed to form an orthonormal basis for \mathbb{R}^m .

²The symmetry of $X^\top X$ ensures that it can be diagonalized by an orthogonal matrix, as guaranteed by the spectral theorem. The spectral theorem states that any real symmetric matrix can be decomposed as $V\Lambda V^\top$, where V is an orthogonal matrix whose columns are the eigenvectors of the matrix, and Λ is a diagonal matrix whose entries are the corresponding real eigenvalues. The positive semi-definiteness of $X^\top X$ follows from the fact that for any vector $z \in \mathbb{R}^d$, the quadratic form $z^\top X^\top X z = \|Xz\|^2 \geq 0$, meaning all eigenvalues of $X^\top X$ are non-negative. This guarantees that Λ has only non-negative entries. The orthonormal eigenvectors correspond to the standard diagonalization of symmetric matrices, meaning that $X^\top X$ can be rewritten as $V\Lambda V^\top$, where V is an orthogonal matrix ($V^\top V = I$) and Λ is a diagonal matrix containing the eigenvalues of $X^\top X$. The fact that V is orthogonal ensures that the eigenvectors form an orthonormal basis of \mathbb{R}^d .

Summary of Steps:

1. Compute $X^\top X$ and find its eigenvalues λ_i and eigenvectors v_i .
2. The singular values are $\sigma_i = \sqrt{\lambda_i}$.
3. Construct V from the eigenvectors v_i of $X^\top X$.
4. Compute U using $U = \frac{1}{\sigma_i}XV$, normalizing the columns as needed.
5. Assemble S , U , and V to complete the SVD: $X = USV^\top$.

Exercise 1.2.1 (Finding the SVD of a Simple Matrix). *Find SVD of the matrix:*

$$X = \begin{bmatrix} 1 & 0 & 0 & 0 & 2 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 \end{bmatrix}.$$

Exercise 1.2.2 (SVD for Matrix Completion: Low-Rank Approximation and Prediction). *Matrix completion is a classical application of the Singular Value Decomposition (SVD). Suppose we observe a partially filled user – item rating matrix $R \in \mathbb{R}^{m \times n}$, where rows correspond to users, columns to items, and entries represent observed ratings. Missing entries are denoted by ?.*

A small illustrative example is:

$$R = \begin{bmatrix} 5 & ? & 3 & 1 \\ 4 & 2 & ? & ? \\ ? & 5 & 4 & ? \\ 1 & ? & 2 & 3 \end{bmatrix}.$$

The core idea of SVD-based matrix completion is that user preferences and item attributes often lie near a low-rank subspace. Thus, one approximates R by retaining only the leading k singular values:

$$R_k \approx U_k S_k V_k^\top, \quad k \ll \min(m, n).$$

Missing entries are then predicted using the reconstructed approximation R_k .

Tasks.

1. **Low-rank approximation.** Assume the full matrix R were known. (a) Write explicitly how to obtain its truncated SVD $R_k = U_k S_k V_k^\top$. (b) Explain why keeping only the top k singular values amounts to projecting R onto the subspace of maximal variance.
2. **Predicting missing entries.** Suppose we only observe the entries that are not ?. Describe a procedure (theoretical, not computational) for estimating the missing entries using:

$$\min_{X \in \mathbb{R}^{m \times n}} \|P_\Omega(X - R)\|_F^2 \quad \text{subject to} \quad \text{rank}(X) \leq k,$$

where P_Ω is the projection onto observed entries.

Explain why alternating between (i) filling missing entries using the current X , and (ii) recomputing a rank- k SVD of X , converges to a low-rank completion.

3. **Interpretation.** In the movie-rating context, explain the meaning of the factors U_k , S_k , V_k :
- What do the rows of U_k represent?
 - What do the columns of V_k represent?
 - Why does low-rank structure correspond to the existence of “latent” genres or user preference dimensions?
4. **Optional (conceptual): Connection to the Netflix Prize.** Briefly summarize why low-rank methods (including SVD-based ones) historically performed well for large-scale recommendation systems such as the Netflix dataset.

Note. This exercise is theoretical; no code implementation is required. The goal is to connect the SVD to a widely used practical application that depends on low-rank structure, prediction, and latent representation learning.

1.2.2 Reduced Representation of a Single Data Point with SVD

Let $X \in \mathbb{R}^{n \times d}$ be a dataset whose rows $x_i \in \mathbb{R}^d$ represent individual data points. A central goal in data analysis and machine learning is to obtain a low-dimensional representation $z_i \in \mathbb{R}^k$, with $k \ll d$, such that z_i preserves the most informative geometric structure of x_i . This is essential in applications involving compression, denoising, and feature extraction. The Singular Value Decomposition (SVD) of X ,

$$X = USV^\top,$$

provides a principled way to construct such reduced representations. The columns of V are right singular vectors, representing principal directions of variability in the dataset, while the singular values $\sigma_1 \geq \sigma_2 \geq \dots$ quantify how much variance is captured in each direction. To reduce the dimensionality of a single point x_i , we project it onto the top k singular directions:

$$z_i = x_i V_k,$$

where $V_k \in \mathbb{R}^{d \times k}$ contains the first k columns of V . The reduced vector $z_i \in \mathbb{R}^k$ captures the dominant features of x_i while ignoring lower-variance structure.

Example 1.2.1 (Dimensionality Reduction on MNIST). To illustrate these concepts, the accompanying Python notebook `MNIST-SVD.ipynb` performs SVD on a small batch of MNIST digits (flattened into vectors in \mathbb{R}^{784}). Each image is centered by subtracting the batch mean, and the SVD of the resulting data matrix is computed.

Fig. 1.2 provides a visual summary:

1. the mean image of the batch,
2. the first two right singular vectors (“eigendigits”),
3. rank- k reconstructions of a sample image for $k = 5, 20, 50$.

These reconstructions demonstrate how increasing the number of retained components improves the fidelity of the approximation, with coarse global structure captured by small k and finer details emerging as k grows.

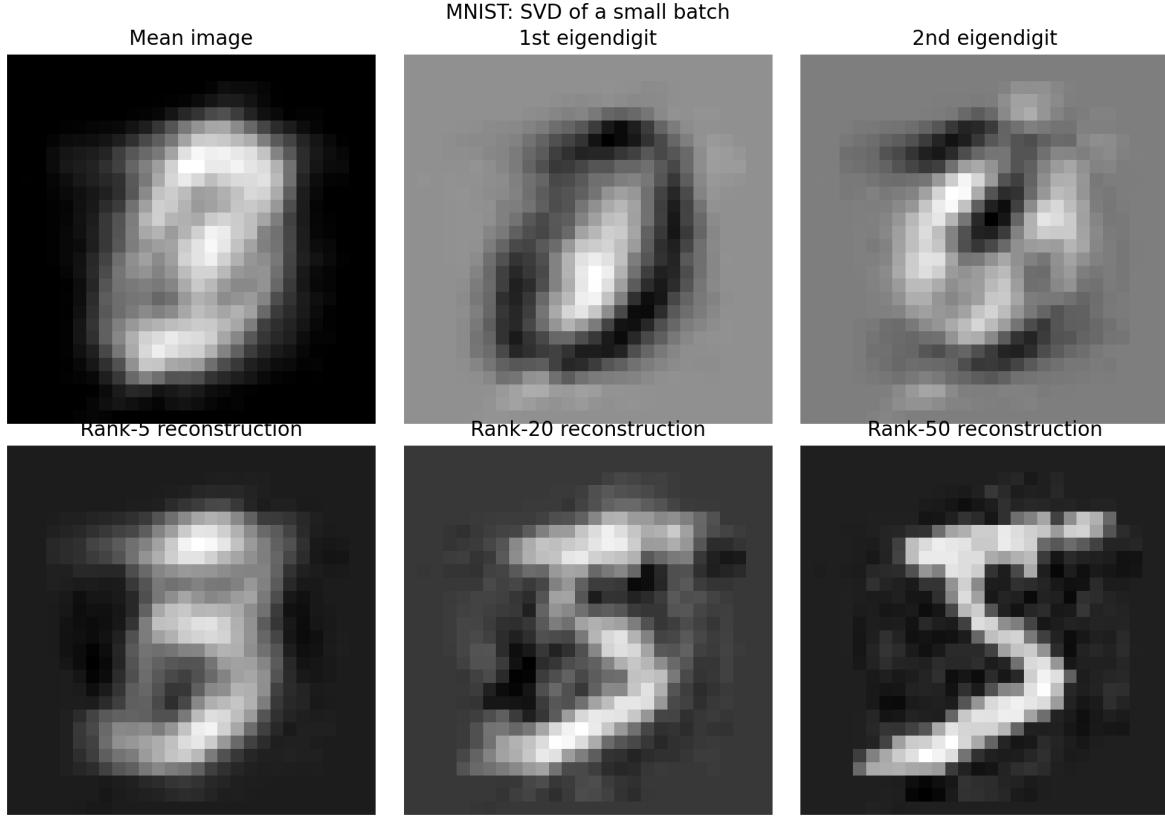


Figure 1.2: Mean image, principal directions, and rank- k reconstructions. Top row: mean image and the first two principal directions (eigen-digits) obtained from the SVD of a small MNIST batch. Bottom row: reconstructions of a sample digit using rank-5, rank-20, and rank-50 approximations. The leading singular vectors capture large-scale geometric patterns and strokes typical of handwritten digits.

Spectral Diagnostics and Interpretability Beyond reconstructions, the singular value spectrum provides insight into the intrinsic dimensionality and structure of image datasets. The rich spectral analysis in Fig. 1.3 shows four complementary views:

1. **Single-batch spectrum** (upper left): normalized singular values exhibit rapid decay, reflecting a strong low-rank structure typical of handwritten digits.
2. **Multiple random batches** (upper right): several spectra computed from different random subsets of MNIST. Their remarkable alignment indicates that the dominant principal directions are stable across samples—an important empirical fact that justifies using SVD on small batches.
3. **Per-digit spectra** (lower left): classes differ in intrinsic geometric complexity. For example, the digit “1” exhibits a sharper spectral decay, reflecting its nearly one-dimensional structure, while digits such as “8” or “2” require more components to capture curved strokes.

4. **Cumulative variance curve** (lower right): the curve

$$E_k = \frac{\sum_{j=1}^k \sigma_j^2}{\sum_{j=1}^d \sigma_j^2}$$

quantifies how quickly variance is captured. For MNIST, a modest number of singular directions (typically $k \approx 40\text{--}60$) already explains the vast majority of variance.

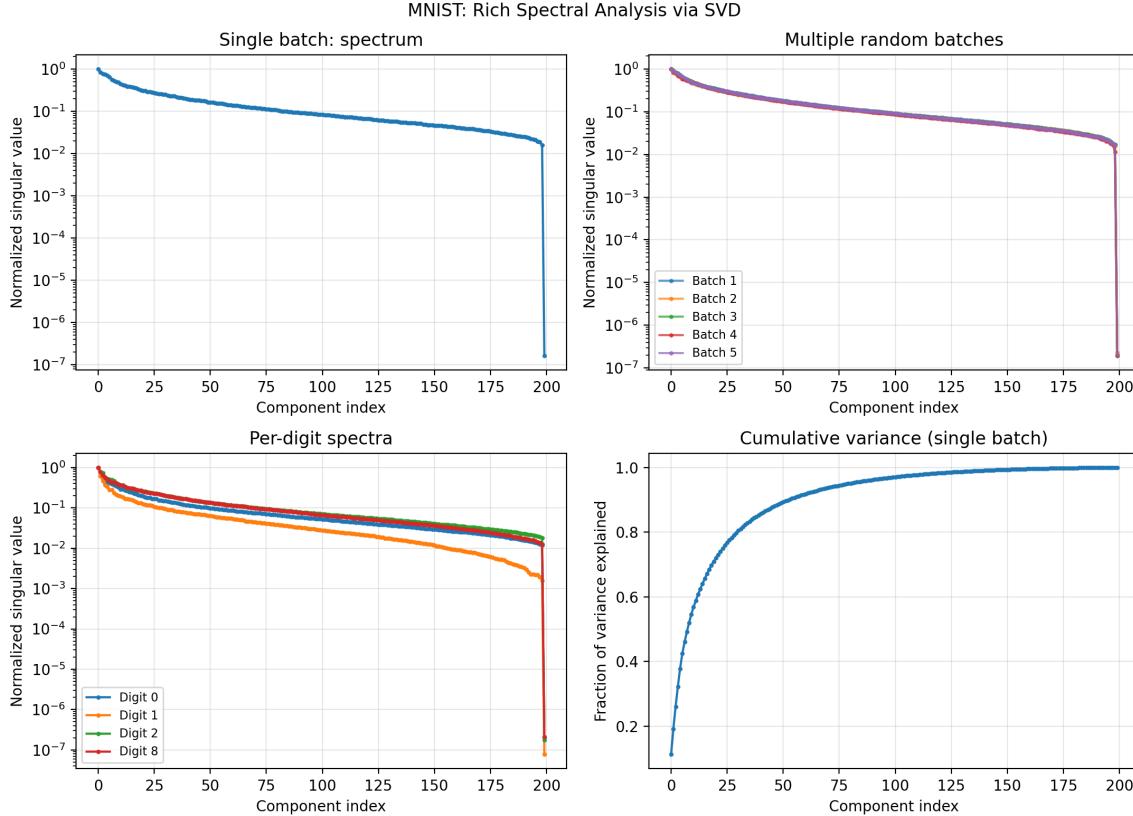


Figure 1.3: **MNIST: rich spectral analysis via SVD.** Upper left: spectrum from a single batch. Upper right: spectra from multiple random batches, demonstrating spectral stability across samples. Lower left: per-digit spectra for digits $\{0, 1, 2, 8\}$, revealing differences in geometric complexity. Lower right: cumulative variance curve for a canonical batch, showing that a relatively small number of components captures most of the dataset's structure.

Key Properties

- **Dimensionality Reduction:** The projection $z_i = x_i V_k$ compresses data from d to k dimensions while preserving dominant directions.
- **Information Retention:** Singular values quantify explained variance; keeping the first k singular vectors ensures that the reduced representation retains most of the dataset's structure.

- **Interpretability:** Eigendigits reveal meaningful global patterns (strokes, spatial templates) learned directly from the data.
- **Spectral Stability:** Similar spectra across random batches and characteristic differences across digit classes highlight both the universality and the specificity of data-driven structure in MNIST.

Why SVD Matters for Generative AI

Although the Singular Value Decomposition (SVD) is introduced in this chapter as a linear-algebraic tool for dimensionality reduction, it plays a conceptual role throughout modern Generative AI. This sidebar highlights several reasons why understanding SVD helps in interpreting and designing generative models of AI.

1. Covariance, Gaussian Geometry, and Whitening. The starting point for diffusion models and many Variation Auto Encoders (VAE) discussed later in the book is a Gaussian reference distribution. The covariance matrix admits an eigen-decomposition:

$$\Sigma = V\Lambda V^\top,$$

and SVD reveal the principal axes along which the data varies. Whitening and preconditioning — routine steps in deep learning — depend on this diagonalization. Understanding these transformations clarifies why diffusion models prefer isotropic priors and why Gaussian scores have the form $\nabla_x \log p(x) = -\Sigma^{-1}(x - \mu)$.

2. Intrinsic Dimension and Spectral Decay. A rapidly decaying singular spectrum indicates that the dataset lies near a low-dimensional manifold. This observation connects directly to the practice of:

- choosing small latent dimensions in VAEs,
- using Low-Rank Approximations (LORA) in transformers,
- compressing neural representations without significant performance loss.

SVD thus provides the first quantitative measure of *intrinsic dimension*.

3. Conditioning, Stiffness, and Gradient Scales. Diffusion models evolve probability densities along continuous-time paths. Low-variance directions (small singular values) correspond to stiff directions in the reverse-time Stochastic Ordinary Differential Equation (SODE), often requiring careful noise schedules or preconditioning. Interpreting these stiff modes through SVD gives intuition about why some data manifolds are harder to sample than others.

4. Score Approximation and Data Geometry. The score of the data distribution, $\nabla_x \log p_{\text{data}}(x)$, plays a central role in score-based generative modeling. For Gaussian data, SVD shows that the score has larger magnitude in low-variance directions, explaining why diffusion models denoise anisotropically even with isotropic noise.

5. Low-Rank Structure in Attention Mechanisms. Although not expressed through SVD explicitly, the attention matrices in transformers frequently exhibit approximate low-rank structure. This has enabled:

- spectral compression,
- linear-attention variants,
- efficient fine-tuning (LoRA),
- structured low-rank updates in large models.

Understanding SVD prepares the reader for these ideas in later chapters.

Overall: SVD is more than a tool for dimensionality reduction. It provides a geometric lens through which many aspects of generative modeling become clearer: the structure of data manifolds, conditioning of sampling processes, Gaussian reference models, and the spectral behavior of network representations. This makes SVD a cornerstone concept for the mathematics of modern Generative AI.

1.2.3 Eigen-Decomposition

Eigen-Decomposition (ED) is a fundamental matrix factorization technique applicable to square matrices. Given a square matrix $A \in \mathbb{R}^{d \times d}$, eigen-decomposition expresses A in terms of its eigenvalues and eigenvectors. An eigenvalue-eigenvector pair (λ, v) satisfies the equation:

$$Av = \lambda v,$$

where $\lambda \in \mathbb{C}$ is the eigenvalue, and $v \in \mathbb{C}^d$ is the corresponding eigenvector³. The eigenvectors represent directions that remain invariant under the linear transformation defined by A , and the eigenvalues describe the scaling along these directions.

In the general case, A may have complex eigenvalues and eigenvectors, even if the entries of A are real. This occurs when A has non-symmetric or non-diagonalizable properties, such as in rotations or other non-conservative transformations. Eigenvalues can be repeated (degenerate) or distinct, and not all square matrices are guaranteed to have a full set of linearly independent eigenvectors. However, for matrices that are diagonalizable (no zero eigenvalues), ED expresses A as:

$$A = Q\Lambda Q^{-1},$$

where:

- $Q \in \mathbb{C}^{d \times d}$ contains the eigenvectors of A as columns.
- $\Lambda \in \mathbb{C}^{d \times d}$ is a diagonal matrix with the eigenvalues of A on its diagonal.

³Here, \mathbb{C} represents the system of complex numbers. While in this living book we primarily work with the system of real numbers, \mathbb{R} , which is naturally a subset of the system of complex numbers, we introduce complex numbers to highlight their generality. This generality is particularly valuable in the context of the ED, where complex eigenvalues and eigenvectors often arise even for real-valued matrices.

1.2.4 Connecting SVD and ED for Symmetric Positive-Definite Matrices

The relationship between SVD and ED becomes apparent when considering symmetric positive-definite matrices, such as the aforementioned covariance matrix of a dataset: $\Sigma = \frac{1}{n}X^\top X$. The covariance matrix $\Sigma \in \mathbb{R}^{d \times d}$ is symmetric and positive semi-definite, meaning its eigenvalues are non-negative. Eigen-decomposition of Σ yields:

$$\Sigma = V\Lambda V^\top,$$

where:

- V contains the eigenvectors of Σ (the principal directions of the data).
- Λ contains the eigenvalues of Σ (the variance explained by each principal direction).

For symmetric positive-definite matrices, the eigenvalues in Λ are equivalent to the squared singular values from the SVD of X . Thus, for such matrices:

$$\Sigma = VS^2V^\top,$$

establishing that SVD and ED are closely related.

Exercise 1.2.3 (Comparing SVD and ED). *Consider the symmetric positive-definite matrix (covariance matrix):*

$$\Sigma = \begin{bmatrix} 4 & 2 \\ 2 & 3 \end{bmatrix}.$$

1. *Compute the ED of Σ :*

$$\Sigma = V\Lambda V^\top,$$

where Λ contains eigenvalues and V contains eigenvectors. Perform this computation manually or using software.

2. *Compute the SVD of Σ :*

$$\Sigma = US^2U^\top,$$

where S contains singular values ($\sqrt{\lambda_i}$). Perform this computation manually or using software.

3. *Verify the relationship between eigenvalues (Λ) and singular values (S).*

Example 1.2.2 (Graphs and the Graph Laplacian). *Symmetric matrices naturally arise in the study of graphs, particularly through the graph Laplacian, which encodes important structural properties of the graph. Consider an undirected graph $G = (V, E)$, where V is the set of nodes and E is the set of edges. Let $n = |V|$ denote the number of nodes.*

The adjacency matrix $A \in \mathbb{R}^{n \times n}$ of G is defined as:

$$A_{ij} = \begin{cases} 1, & \text{if there is an edge between nodes } i \text{ and } j, \\ 0, & \text{otherwise.} \end{cases}$$

The degree matrix $D \in \mathbb{R}^{n \times n}$ is a diagonal matrix where D_{ii} equals the degree of node i , defined as $D_{ii} = \sum_{j=1}^n A_{ij}$.

The graph Laplacian $L \in \mathbb{R}^{n \times n}$ is then given by:

$$L = D - A.$$

- **Symmetry and Semi-Definiteness:** L is symmetric and positive semi-definite.
- **Eigenvalues:** The eigenvalues of L provide insight into the structure of G :
 - The smallest eigenvalue is always 0.
 - The multiplicity of the 0 eigenvalue corresponds to the number of connected components in the graph.

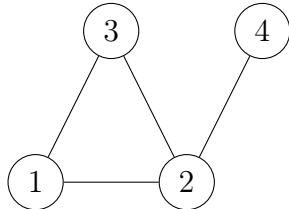


Figure 1.4: An example graph with 4 nodes. Nodes 1, 2, and 3 form a triangle, and node 4 is connected to node 2.

Exercise 1.2.4 (Understanding the Graph Laplacian). Consider the graph shown in Figure 1.4. Perform the following tasks:

1. Write down the adjacency matrix A for the graph.
2. Compute the degree matrix D .
3. Derive the graph Laplacian $L = D - A$.
4. Find the eigenvalues of L and verify the multiplicity of the 0 eigenvalue.
5. Use the eigenvectors corresponding to the smallest two nonzero eigenvalues to embed the graph in a 2D space. Discuss how the embedding reflects the structure of the graph.

From Static Structure to Dynamic Change

Chapter 1 provided the essential algebraic toolkit for high-dimensional data: the geometry of vectors, the mechanics of transformations, and the analysis offered by SVD and ED. Crucially, *Linear Algebra describes the static state and structure of data*. However, for an AI model to learn, adapt, and generate new content, it must change its internal weights – a process that involves movement and optimization. To model this dynamic change, we must shift from algebra to calculus. Chapter 2 will introduce

Automatic Differentiation (AD), the foundational technique for calculating the direction of change (the gradient), and *Differential Equations*, the mathematical language which is used (in particular) for modeling change over time.

Chapter 2

Calculus and Differential Equations (in AI)

2.1 Automatic Differentiation

Automatic differentiation (AD) is the backbone of modern scientific computing and machine learning. Its central idea is simple but powerful: *a complex function is nothing more than a composition of elementary operations, and the chain rule can be applied systematically and efficiently.* AD differs from symbolic differentiation (which often produces large expressions) and from numerical finite differences (which suffer from truncation and rounding errors). Instead, AD evaluates derivatives *exactly up to machine precision* while keeping the computational cost controlled.

Today, AD powers all major deep-learning frameworks, including PYTORCH, TENSORFLOW, and JAX. In this section we introduce the core principles of AD, illustrate them through a computational graph for a simple multivariate function, and then show how modern AD systems compute gradients efficiently in practice.

Example 2.1.1 (Decomposing a Function into a Directed Acyclic Graph). *Consider the scalar function*

$$f(x_1, x_2) = \frac{\sin(x_1 + x_2)}{x_1}.$$

To expose AD's structure, we rewrite the computation as a sequence of elementary operations:

$$w_1 = x_1, \quad w_2 = x_2, \quad w_3 = w_1 + w_2, \quad w_4 = \sin(w_3), \quad w_5 = \frac{w_4}{w_1}. \quad (2.1)$$

These steps form a Directed Acyclic Graph (DAG), also called a computational graph. Each node carries a value computed from its parents, while each edge encodes the dependency structure. This viewpoint is fundamental to AD: instead of differentiating the full expression for f symbolically, AD applies the chain rule locally along the edges of the graph. During forward execution, values flow from inputs to output; in reverse mode (backpropagation), sensitivities propagate from the output back to all inputs. Fig. 2.1 visualizes both modes. Differentiating f means expressing the differential

$$df = dw_5 = \partial_{x_1} f dx_1 + \partial_{x_2} f dx_2.$$

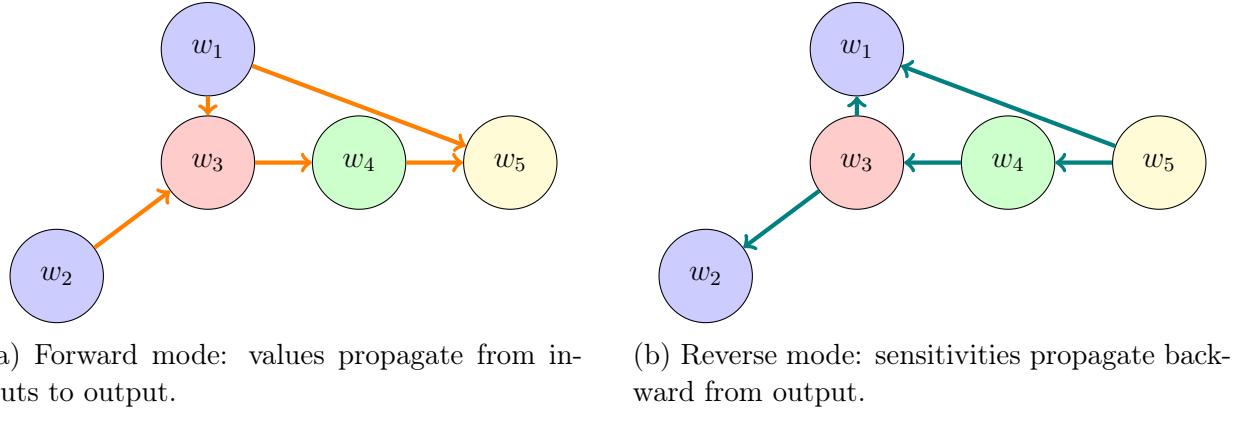


Figure 2.1: Forward-mode (left) and reverse-mode (right) propagation on the computational graph of $f(x_1, x_2) = \sin(x_1 + x_2)/x_1$. Orange arrows illustrate the flow of primal values in forward execution; teal arrows illustrate the flow of adjoints (gradients) in backpropagation.

Using chain-rule expansions along the graph gives

$$\begin{aligned} dw_5 &= \frac{\partial w_5}{\partial w_1} dw_1 + \frac{\partial w_5}{\partial w_4} dw_4, \\ dw_4 &= \frac{\partial w_4}{\partial w_3} dw_3, \\ dw_3 &= \frac{\partial w_3}{\partial w_1} dw_1 + \frac{\partial w_3}{\partial w_2} dw_2. \end{aligned}$$

After substitution, one obtains the explicit gradient formulas:

$$\partial_{x_1} f = \frac{\partial w_5}{\partial w_1} + \frac{\partial w_5}{\partial w_4} \frac{\partial w_4}{\partial w_3} \frac{\partial w_3}{\partial w_1}, \quad \partial_{x_2} f = \frac{\partial w_5}{\partial w_4} \frac{\partial w_4}{\partial w_3} \frac{\partial w_3}{\partial w_2}. \quad (2.2)$$

2.1.1 Forward vs. Reverse Mode AD

The computational graph in Fig. 2.1 highlights the two fundamental modes of automatic differentiation. Both modes apply the chain rule locally along the graph, but they differ in the direction in which derivatives are propagated, and therefore in their computational costs.

Forward Mode (many inputs \rightarrow few outputs). Forward mode propagates differentials *with* the computational graph—from inputs toward the output. For each input direction dx_i , forward mode computes the associated chain of partial derivatives all the way to the output. This makes it efficient when:

$$(\# \text{ of inputs}) \ll (\# \text{ of outputs}).$$

Typical use cases include:

- sensitivity analysis of models with a small number of parameters;

- Jacobian–vector products;
- probing which input features influence a model’s prediction.

Conceptually, forward mode answers: “*If I nudge some inputs, how does the output change?*”

Reverse Mode (one or few outputs → many inputs). Reverse mode propagates sensitivities *against* the graph—from the (often scalar) output back to all inputs. Instead of computing all derivatives for all inputs *independently*, reverse mode reuses intermediate adjoints, making it dramatically more efficient when:

$$(\# \text{ of outputs}) \ll (\# \text{ of inputs}).$$

This is the case when training neural networks, where the loss is a scalar and the number of parameters ranges from millions to tens of billions. Reverse mode therefore underlies backpropagation.

Conceptually, reverse mode answers: “*How does this particular output depend on every input or parameter?*”

Forward vs. Reverse: summary of complexity. Let $n = \#\text{inputs}$ and $m = \#\text{outputs}$. Then:

$$\text{Forward mode cost} \sim \mathcal{O}(n), \quad \text{Reverse mode cost} \sim \mathcal{O}(m).$$

Thus:

Scenario	Preferred AD mode
$n \ll m$	Forward mode
$m \ll n$	Reverse mode (backprop)

Common pitfalls in interpreting AD.

- **AD is not symbolic differentiation.** AD never constructs or simplifies algebraic expressions; it differentiates the *execution trace* of a program.
- **AD is not finite differences.** AD is exact to machine precision—there is no truncation error from numerical differencing.
- **Forward vs. reverse are not interchangeable.** They compute the same mathematical derivatives but have vastly different runtime and memory characteristics.
- **Reverse-mode AD must store intermediates.** Backprop requires access to all intermediate values, which is why memory usage grows with model depth. Techniques such as check-pointing and recomputation address this.

The next example shows how modern AD systems (here in PYTORCH) use reverse-mode AD in practice and how their performance compares to finite differences.

Example 2.1.2 (AD in Practice via Python and PyTorch). *We provide a Jupyter/Python notebook AD.ipynb that illustrates three core aspects of AD:*

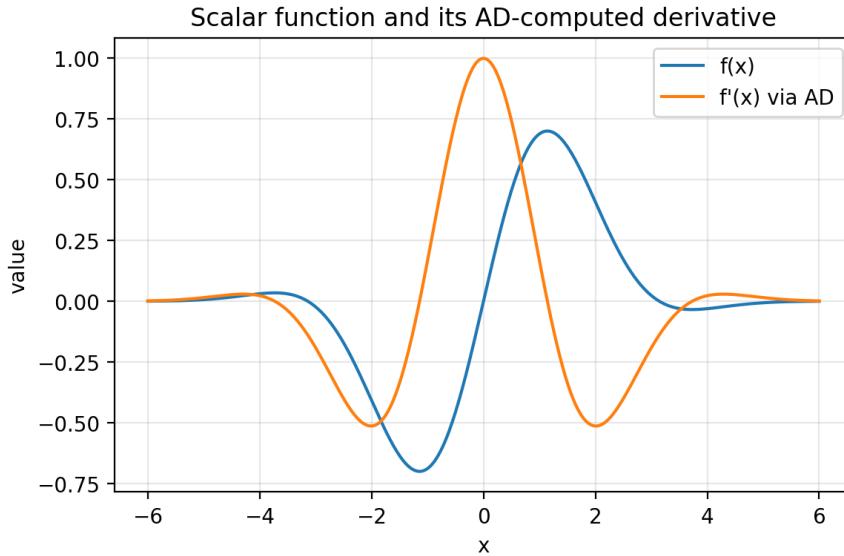


Figure 2.2: Scalar function $f(x) = \sin(x)e^{-x^2/5}$ and its derivative $f'(x)$ computed via reverse-mode AD in PyTorch. Generated by `AD.ipynb`.

1. **Automatic differentiation of a 1D function.** We evaluate the scalar function

$$f(x) = \sin(x) e^{-x^2/5}$$

on a grid and compute $f'(x)$ using reverse-mode AD. Fig. 2.2 shows both curves.

2. **Gradient computation for a multivariate function.** For the function

$$k(x, y, z) = x^2y + yz + z^3,$$

the notebook compares the analytic gradient with the PyTorch AD gradient.

3. **Runtime scaling: finite differences vs. reverse-mode AD.** For a d -dimensional function, finite differences require d function evaluations, while reverse-mode AD requires only one backward pass. The runtime comparison in Fig. 2.3 illustrates this difference on a log-log scale.

All figures are automatically saved in the `figs/` directory.

Why Automatic Differentiation Matters for Modern Generative AI

Automatic differentiation (AD) is not just a technical tool for computing gradients – it is one of the mathematical *pillars* of modern Generative AI. Almost every contemporary model, from diffusion models to transformers to large vision–language systems, relies critically on AD in several ways:

- **Training deep networks.** Reverse-mode AD (backpropagation) enables efficient optimization of models with millions or billions of parameters by computing

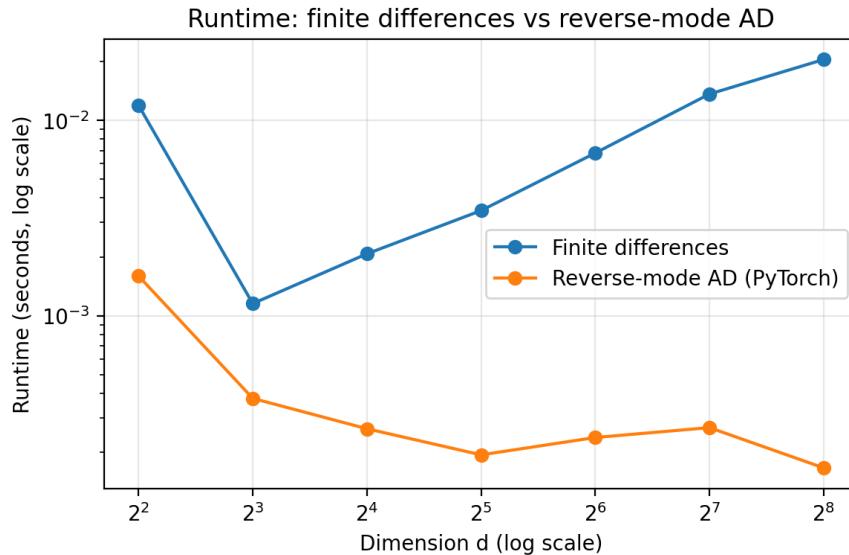


Figure 2.3: Runtime comparison between finite differences and PyTorch reverse-mode AD for increasing dimension d . Reverse-mode AD scales dramatically better. Generated by `AD.ipynb`.

the gradient of a scalar loss in a single backward sweep.

- **Learning vector fields for generation.** Diffusion models, flow-matching models, and continuous normalizing flows all require gradients of log-densities or vector fields. These gradients are learned by minimizing squared-error losses whose evaluation depends directly on AD.
- **Score estimation.** In score-based generative modeling, the objective is to regress onto $\nabla_x \log p_t(x)$. AD makes it possible to differentiate through neural networks that parameterize the score, the noise schedule, or the drift of a reverse-time SDE.
- **Differentiating through ODE/SDE solvers.** Many modern samplers differentiate through numerical ODE or SDE integrators. AD is essential for implicit layers, neural ODEs, and diffusion-based likelihood estimation.
- **Architectural design and meta-learning.** Transformer attention maps, activation functions, normalization layers, and even optimizers (e.g., Adam) are differentiable components whose internal gradients must be computed automatically for end-to-end learning.

In short, understanding AD is understanding how modern generative models *learn*, *optimize*, and *evolve* during training. It is one of the hidden engines enabling the scalability and performance of today’s systems.

Exercise 2.1.1 (The Chain Rule, Backpropagation, and AD in Practice). *This exercise*

bridges the theoretical application of the Chain Rule (Backpropagation) with its practical implementation using PyTorch’s Automatic Differentiation (AD).

1. **Theoretical Backpropagation Derivation.** Consider a simple two-layer network $f(x) = W_2\sigma(W_1x)$, where $W_1 \in \mathbb{R}^{h \times d}$, $W_2 \in \mathbb{R}^{m \times h}$, and $\sigma(\cdot)$ is an element-wise activation function.
 - (a) Write the full Jacobian $J = \frac{\partial f}{\partial x}$ as a product of matrix and diagonal matrix Jacobians.
 - (b) If L is a scalar loss function, use the chain rule to derive the gradient $\frac{\partial L}{\partial W_1}$ in terms of W_2 , the derivative σ' , and the loss gradient $\frac{\partial L}{\partial f}$. This is the essence of backpropagation.
2. **Computational Graph and Gradient Verification.** This part builds on the computational examples in the notebook `AD.ipynb`.
 - (a) **Constructing a DAG:** Consider the function $h(x, y) = \ln(x^2 + y^2) + \sin((x + y)^2)$. Construct a computational graph (DAG) for h by introducing intermediate variables and then compute ∇h manually using reverse-mode AD.
 - (b) **PyTorch Verification:** Modify the function $k(x, y, z) = x^2y + yz + z^3$ by adding a new nonlinearity, e.g., replace z^3 with $\tanh(z^3)$ or add a term such as $x \exp(y)$. Compute the analytic gradient and verify it against PyTorch AD.
3. **Hardware and Scaling Insight.**
 - (a) **Forward vs. Reverse Mode Scaling:** Use the runtime-comparison cell in `AD.ipynb` and extend it to dimensions $d = 512, 1024, 2048$ (if your machine allows). Plot the results and comment on the scaling of finite differences versus automatic differentiation.
 - (b) **Gradient Stability:** What is the fundamental problem (related to the product of Jacobians) that the backpropagation algorithm introduces in very deep networks, leading to the vanishing/exploding gradient problem? Briefly discuss how one simple technique, like gradient clipping, helps mitigate this issue.

2.2 Differential Equations: Foundations and Links to AI

Following our exploration of Automatic Differentiation, which provides the instantaneous rate of change (the gradient), we now turn to *Differential Equations* – the language used to describe change and dynamics over time. While classical deep learning focused on static mapping from input to output, the frontier of Generative AI, including Diffusion Models, Flow Matching, and Neural ODEs, fundamentally relies on modeling continuous-time dynamics. This section first reviews the mathematical foundations of Ordinary Differential Equations (ODEs) using symbolic methods. We then bridge this classical view to the modern AI challenge of *learning dynamics from data* (ODE-based regression), and finally examine the structure and solution of higher-order systems on the example of the second order ODEs.

2.2.1 Ordinary Differential Equations (ODEs) – Primer

An Ordinary Differential Equation (ODE) is an equation involving derivatives of a function with respect to a single independent variable:

$$\frac{dx}{dt} = f(x, t), \quad x \in \mathbb{R}^n, \quad t \in \mathbb{R}. \quad (2.3)$$

Often, a shorthand notation is used where \dot{x} represents $\frac{dx}{dt}$.

ODEs play a crucial role in modeling time-dependent phenomena across science and engineering, including motion, heat transfer, and population dynamics.

Integration: The Simplest Case $\dot{x} = f(t)$

Let us begin with the simplest case where $n = 1$ and the right-hand side f does not depend on x . In this case, Eq. (2.3) becomes:

$$\frac{dx}{dt} = f(t).$$

Integrating this equation gives the symbolic solution:

$$x(t) = \int f(t) dt + \text{const.} \quad (2.4)$$

The integral on the right-hand side is called an indefinite integral or anti-derivative. While differentiation is straightforward and always produces elementary expressions for well-defined functions, *integration is inherently more challenging*. In many cases, the anti-derivative of $f(t)$ cannot be expressed in terms of elementary functions. This distinction highlights the complexity of integration, which remains a significant area of mathematical study and justifies the need for numerical methods.

Exercise 2.2.1. Consider two “integrable” examples :

$$\frac{dx}{dt} = \begin{cases} \sin(at), & (I); \\ \exp(at), & (II), \end{cases}$$

where $a \in \mathbb{R}$.

1. Solve these ODEs analytically, fixing the integration constant such that $x(0) = x_0$.
2. Analyze the asymptotic behavior of the solutions as $t \rightarrow +\infty$, considering the dependence on the initial condition x_0 and the parameter a . Does the solution grow, decay, or remain bounded?

Autonomous ODEs: $\dot{x} = f(x)$

Another important one-dimensional case arises when f depends only on x (and not explicitly on t). This is referred to as an *autonomous ODE*. The equation can often be solved by separating variables:

$$\frac{dx}{f(x)} = dt.$$

Integrating both sides yields an implicit solution:

$$\int \frac{dx}{f(x)} = t + \text{const.}$$

As with the previous case, the anti-derivatives involved may or may not have elementary function forms. For autonomous systems, the focus shifts to qualitative analysis, such as the existence and stability of fixed points.

Exercise 2.2.2.

$$\frac{dx}{dt} = \begin{cases} \sin(ax), & (I); \\ \exp(ax), & (II), \end{cases}$$

where $a \in \mathbb{R}$.

1. Solve these ODEs analytically, fixing the integration constant such that $x(0) = x_0$.
2. Analyze the asymptotic behavior of the solutions as $t \rightarrow +\infty$, considering the dependence on x_0 and a .
3. Explore the existence of fixed points. If they exist, analyze their stability under small perturbations of the initial condition $x(0)$.
4. Design a function $f(x)$ such that the ODE serves as a binary classifier, separating the domain $x > 0$ and $x < 0$. Modify the function to create a time-evolving boundary (e.g., oscillatory behavior).

Symbolic Programming: Addressing Non-Integrable ODEs

The most general one-dimensional equation, where $f(x, t)$ is an arbitrary function of both x and t , is typically not integrable. We cannot present the solution explicitly or even implicitly in terms of algebraic equation(s). This challenge opens the door to two primary methodological avenues: *symbolic methods* and *approximation techniques*. We first discuss the former.

Symbolic programming (SP) focuses on representing mathematical expressions as abstract symbols and manipulating them algebraically. The analytic form of the integral in Eq. (2.4) exemplifies this approach. SP contrasts sharply with differential programming techniques like Automatic Differentiation (AD), which we discussed in the preceding section. While AD is universally applicable and excels in numerical computation, symbolic integration is constrained by fundamental limitations: many integrals and more generally equations simply cannot be expressed in terms of elementary functions. However, when such a closed-form

symbolic representation *does* exist – for instance, when an integral can be expressed via elementary or easily tabulable functions – this representation becomes extraordinarily efficient, bypassing the need for computationally costly numerical solvers and providing exact mathematical insight.

Role of Symbolic Programming in AI

While Automatic Differentiation (AD) and numerical methods form the basis of training in AI, the role of *Symbolic AI* is growing rapidly, driven by the increasing need for interpretation and rigor in Generative AI models. This resurgence is primarily seen in two areas:

- 1. Scientific Discovery:** Modern AI is increasingly employed for *symbolic regression*, where the objective shifts from merely predicting data to discovering the underlying governing physical law (such as an ODE or PDE) in a closed, analytic form. This leverages the inherent exactness and interpretability of symbolic expressions.
- 2. Reasoning and Interpretability:** Integrating symbolic solvers with Large Language Models (LLMs) allows generative models to perform complex mathematical and logical reasoning with *guaranteed correctness*. This ability overcomes the hallucination issues inherent in purely probabilistic models, which are often constructed without any application-specific *inductive* bias. The powerful synergy between numerical generation (AD) and logical verification (SP) is crucial for building reliable and trustworthy AI systems.

The second approach to solving non-integrable ODEs – powerful *approximation* techniques, such as linearizations – is discussed later in Section 2.3 (in the context of higher-dimensional systems).

2.2.2 Regression – Direct and ODE-Based

An important feature of the ODE (2.3) is that once the function $f(x, t)$ is known and the initial condition $x(0)$ is fixed, the solution $x(t)$ at $t > 0$ (of the so-called Cauchi, initial value problem) is unique and well-defined, even if finding it analytically is challenging. However, what if $f(x, t)$ is unknown, but we have access to a sequence of measurements $x(t_1), x(t_2), \dots$ at specific times t_1, t_2, \dots ?

This setting is called *regression*¹. The goal is to predict the dynamics of $x(t)$ given an initial condition $x(0)$ and the observed sequence of measurements.

In this subsection, we explore two fundamentally different approaches:

- 1. Direct regression:** Learn $x(t)$ directly as a function of t .

¹Mathematical regression is called regression because it originated in studies, by statistician Galton in the late 19th century, involving the tendency of data to revert toward the mean, or *regress*. Galton was studying the relationship between the heights of parents and their children.

- 2. ODE-based regression:** Learn $f(x, t)$ from the data using the structure of the ODE.
It is a continuous time version of the so-called auto-regression.

To illustrate the benefits of ODE-based regression, we consider the example of an over-damped oscillator.

Over-Damped Oscillator

Consider the one-dimensional linear ODE, commonly referred to as the *over-damped oscillator*:

$$\dot{x}(t) = -\gamma x(t) + g(t), \quad (2.5)$$

where:

- $x(t)$ is the state variable at time t ,
- $\gamma > 0$ is the damping coefficient, ensuring an over-damped response,
- $g(t)$ is an external forcing term.

This ODE models systems where the state responds slowly to external inputs due to significant damping. Such dynamics are common in physics and engineering, and they provide a foundation for understanding time-dependent processes in AI, such as optimization and NN dynamics.

Direct Regression vs. ODE-Based Regression

1. **Direct Regression:** Directly fit $x(t)$ using a chosen regression model (e.g., polynomial basis, splines). While straightforward, this approach often struggles with noisy data and fails to uncover underlying dynamics.
2. **ODE-Based Regression:** Instead of fitting $x(t)$, ODE-based regression estimates the parameters γ and $g(t)$ in Eq. (2.5) – which is our equations based – quantitative – model in this example. By leveraging the structure of the ODE, this approach offers:
 - A dynamic relationship between $\dot{x}(t)$ and $x(t)$, reducing over-fitting risks.
 - Robustness to noisy measurements of $x(t)$.
 - Improved generalization to cases where $g(t)$ is complex or time-varying, and also to the ranges of t under-represented or missing in training.
 - Natural incorporation of the initial condition $x(0)$.

Example 2.2.1. *Direct vs ODE-based Regressions* Assume the true dynamics are governed by:

$$\dot{x}(t) = -2x(t) + \sin(t), \quad x(0) = 1,$$

which is a special case of Eq. (2.5). The analytical solution of the equation is:

$$x(t) = e^{-2t} + \frac{1}{\sqrt{5}} (\sin(t) - 2 \cos(t)). \quad (2.6)$$

We simulate $x(t)$ at discrete times $t_i = 0, 0.1, \dots, 5$, adding Gaussian noise $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$, $x(t_i) \rightarrow x(t_i) + \epsilon_i$ to mimic measurement errors.

Direct Regression: Fit the noisy data $x(t)$ by a polynomial of degree d $P(t) = a_0 + a_1 t + a_2 t^2 + \dots + a_d t^d$ solving a least-square optimization problem:

$$\min_{a_0, \dots, a_d} \sum_{i=1}^N (x_i - P(t_i))^2.$$

(We will give more details on the underlying optimization scheme in the chapter devoted to optimization.) While this approach approximates the trajectory, it cannot reveal γ or $g(t)$.

ODE-Based Regression: Using numerical differentiation, estimate $\dot{x}(t)$ and fit $-\gamma x(t) + g(t)$ using the structure of Eq. (2.5). Specifically, we do it in two steps:

1. Since the data consists of discrete observations, the time derivative dx/dt is estimated using finite differences: $\dot{x}_i \approx (x_{i+1} - x_{i-1})/(t_{i+1} - t_{i-1})$, $i = 1, \dots, N-1$.
2. This numerical derivative is then used to set up and solve the least-square optimization problem:

$$(\theta^*, g_1^*, \dots, g_N^*) = \arg \min_{\theta, g_1, \dots, g_N} \sum_{i=1}^{N-1} (\dot{x}_i + \gamma x_i - g_i)^2.$$

This approach not only captures the dynamics but also enables parameter estimation – $\theta^*, g_1^*, \dots, g_N^*$.

This example is implemented in the accompanying notebook `regression.ipynb`. The resulting trajectories from direct and ODE-based regression are illustrated in Figs. 2.4, 2.5.

Exercise 2.2.3 (Regression and Parameter Estimation for a New ODE). This exercise explores the fundamental advantages of learning underlying dynamics (ODE-based regression) compared to simple function approximation (direct regression) when dealing with noisy, time-series data.

Consider the following ordinary differential equation:

$$\dot{x}(t) = -\gamma x(t) + \cos(2t), \quad x(0) = 2.$$

You are given noisy measurements of the trajectory $x(t)$ and are asked to estimate the parameter γ , whose true value is -3 . Use the provided notebook `regression-python.ipynb` to implement and compare both regression methods.

1. **Conceptual Advantage: Why \dot{x} over $x(t)$?**

- (a) Explain the fundamental mathematical reason why trying to fit the instantaneous rate of change $\dot{x}(t)$ is less susceptible to integration error and systematic bias than trying to fit the trajectory $x(t)$ directly, especially when the data contains additive noise.
- (b) Based on your explanation, hypothesize which method (ODE-based or Direct) should yield a more accurate estimate of the parameter γ and why.

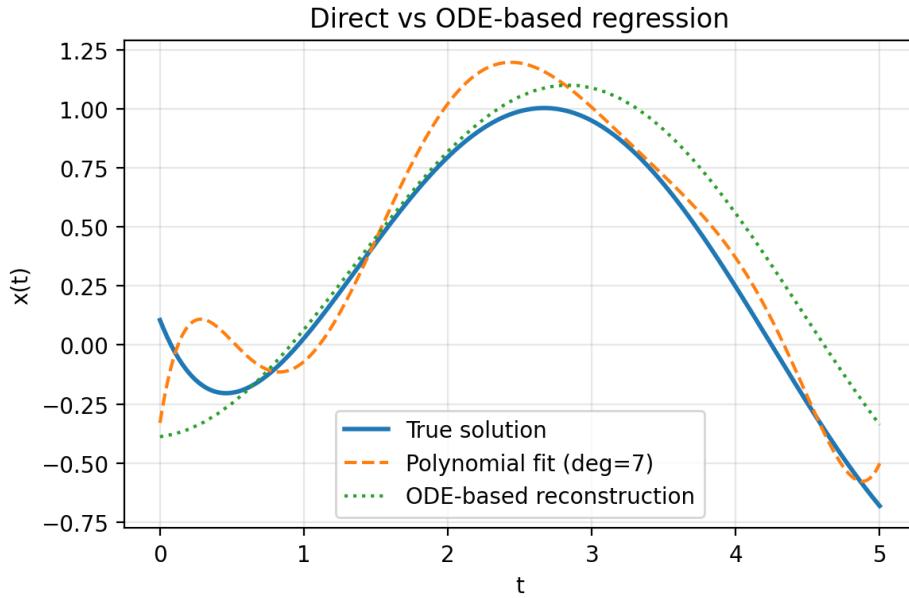


Figure 2.4: Direct polynomial regression vs. ODE-based regression for the over-damped oscillator. The ODE-based fit exploits the dynamical structure and typically generalizes better, especially away from the training window or under higher noise. Generated by `regression.ipynb`.

2. Computational Comparison of Methods: Modify the notebook as necessary to accommodate this new example. Analyze the performance of direct regression and ODE-based regression methods under different levels of additive noise in the data (e.g., consider noise levels $\sigma = 0.1, 0.5, 1$). Compare the following metrics:

- (a) The accuracy of the estimated parameter γ .
- (b) The fidelity of the reconstructed trajectory over the full time domain.

Provide a detailed discussion of your findings, ensuring they support or refute your hypothesis from part 1(b).

3. Heuristic Refinement of γ : The estimation relies on optimization. Without using black-box optimization software (i.e., focusing on the mathematical model), propose a heuristic approach to refine the estimated parameter γ . For example, this might involve iterative adjustments based on the residuals of the ODE constraint or analyzing the discrepancy between the model and the data at specific points. Evaluate the effectiveness of your heuristic by comparing the refined parameter to the true value ($\gamma = -3$) and discussing its advantages and limitations.

Additional Notes: When implementing the ODE-based regression, ensure you properly incorporate the influence of the known forcing term $\cos(2t)$ in your model. Explain any modifications made to the notebook and how they address the specific challenges of this example.

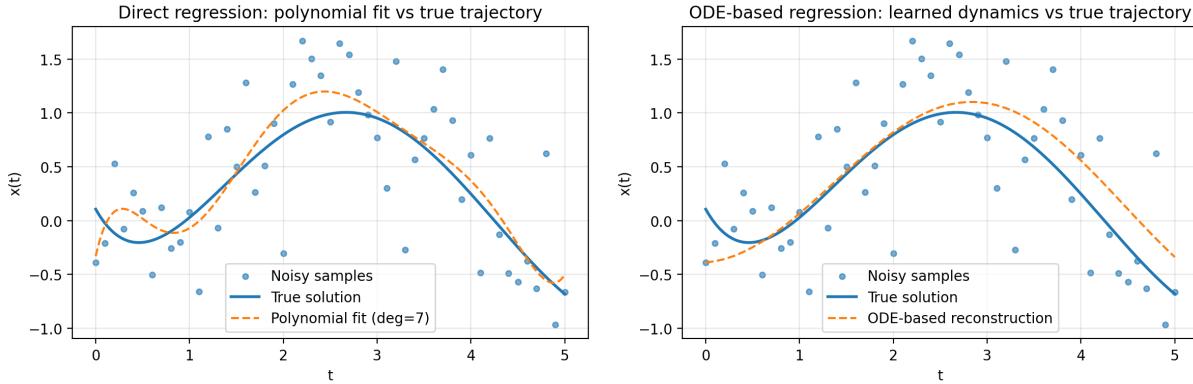


Figure 2.5: Left: direct polynomial regression fit to noisy data. Right: ODE-based regression using the structure $\dot{x}(t) \approx -\gamma x(t) + \sin t$. Both figures generated by `regression.ipynb`.

Neural ODEs and Continuous-Time Auto-Regression

The over-damped oscillator regression example illustrates a simple but very general idea: instead of fitting the trajectory $t \mapsto x(t)$ directly, we fit the *dynamics*

$$\dot{x}(t) = f(x(t), t; \theta)$$

and then *generate* trajectories by integrating the learned vector field forward in time. This viewpoint appears repeatedly in modern *generative*- and *representation*-learning models.

In a *Neural ODE*, the right-hand side $f(x, t; \theta)$ is represented by a neural network. Given observations of trajectories (or a loss that depends on the final state), one learns the parameters θ so that the solution of $\dot{x} = f(x, t; \theta)$ matches the data or minimizes the loss. *Residual networks* (ResNets) can be interpreted as discrete-time approximations of such continuous-time dynamics.

Continuous-time auto-regression. Classical auto-regressive models predict future values x_{t+1} from past values (x_t, x_{t-1}, \dots) via a discrete recurrence. Learning $f(x, t; \theta)$ instead corresponds to a *continuous-time* version of auto-regression: once the dynamics are known, the entire future trajectory is obtained by solving the ODE, not by iterating an explicit discrete map.

From ODEs to generative flows. In likelihood-based generative modeling, *continuous normalizing flows* use ODEs of the form

$$\dot{x}(t) = f(x(t), t; \theta)$$

to transport a simple reference distribution (e.g., a Gaussian) into a complex target distribution. Both the transformation of samples and the evolution of log-densities are governed by the learned vector field f . This connects directly to the finite-dimensional ODE examples in this section.

Later chapters will revisit these ideas in higher dimensions and in stochastic settings (SDEs, diffusion models, and path-integral viewpoints), where the learned dynamics become the core mechanism for sampling and generation.

2.2.3 Second-Order ODE

In Eq. (2.3), x is a state variable of dimension n , and the equation is first-order in derivatives. However, the concepts of order (with respect to derivatives) and dimensionality of the state space n are interrelated and can often be traded off. Let us illustrate this with a second-order one-dimensional ODE:

$$\frac{d^2x}{dt^2} + \gamma \frac{dx}{dt} + f(x) = 0, \quad x(t) \in \mathbb{R}. \quad (2.7)$$

By introducing a two-dimensional state variable $y(t) = (x(t), \dot{x}(t))$, where $\dot{x}(t) = \frac{dx}{dt}$, the second-order ODE can be rewritten as a system of first-order ODEs:

$$\frac{dy}{dt} = \frac{d}{dt} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} y_2 \\ -\gamma y_2 - f(y_1) \end{bmatrix}.$$

This reformulation not only simplifies numerical and analytical approaches to solving the equation but also provides geometric insight into the system's behavior by analyzing trajectories in the two-dimensional phase space.

The second-order ODE in Eq. (2.7) (or equivalently the system of first-order ODEs) significantly enriches the variety of possible solution behaviors. For example, if we choose $f(x) = \omega^2 x$, the equation becomes:

$$\frac{d^2x}{dt^2} + \gamma \frac{dx}{dt} + \omega^2 x = 0,$$

which describes the dynamics of a *damped harmonic oscillator*.

Solving the Damped Harmonic Oscillator

To solve the damped harmonic oscillator, assume a solution of the form $x(t) = e^{\lambda t}$. Substituting into the equation gives the characteristic equation:

$$\lambda^2 + \gamma\lambda + \omega^2 = 0.$$

The roots of this quadratic equation are:

$$\lambda_{1,2} = -\frac{\gamma}{2} \pm \sqrt{\frac{\gamma^2}{4} - \omega^2}.$$

The nature of the solution depends on the discriminant $\Delta = \frac{\gamma^2}{4} - \omega^2$:

- **Over-damped Case ($\Delta > 0$):** The roots $\lambda_{1,2}$ are real and distinct, leading to a solution of the form:

$$x(t) = C_1 e^{\lambda_1 t} + C_2 e^{\lambda_2 t}.$$

The motion decays monotonically without oscillations.

- **Critically Damped Case ($\Delta = 0$):** The roots $\lambda_{1,2}$ are real and equal, leading to a solution:

$$x(t) = (C_1 + C_2 t) e^{-\frac{\gamma}{2} t}.$$

This represents the fastest decay to equilibrium without oscillation.

- **Under-damped Case ($\Delta < 0$):** The roots $\lambda_{1,2}$ are complex conjugates, $\lambda_{1,2} = -\frac{\gamma}{2} \pm i\sqrt{\omega^2 - \frac{\gamma^2}{4}}$, leading to an oscillatory solution:

$$x(t) = e^{-\frac{\gamma}{2}t} (C_1 \cos(\omega_d t) + C_2 \sin(\omega_d t)),$$

where $\omega_d = \sqrt{\omega^2 - \frac{\gamma^2}{4}}$ is the damped angular frequency.

Phase Portrait and Conservation Law(s)

Let us now see how the solution looks from the two-dimensional perspective, that is, by simultaneously considering both the coordinate x of the harmonic oscillator and its rate of change — the velocity \dot{x} . This leads to the notion of the *phase portrait*, which offers a convenient and intuitive way of visualizing the dynamics. Let us illustrate this on a slightly more general case than the harmonic oscillator. Instead of the linear Hookean force $-\omega^2 x$, we will introduce a potential force of general position, while simplifying the setup by discussing the undamped case:

$$\frac{d^2x}{dt^2} = -\frac{dU(x)}{dx}, \quad (2.8)$$

where $U(x)$ is the potential energy associated with the force $f(x) = -\frac{dU(x)}{dx}$. In the case of the harmonic potential, $U(x) = \frac{1}{2}\omega^2 x^2$, Eq. (2.8) describes sustained oscillations without decay.

In general, for any bounded-from-below potential $U(x)$ (i.e., $\forall x, U(x) > \text{const}$), Eq. (2.8) represents a *conservative dynamical system*. These systems are termed *conservative* because, even if Eq. (2.8) cannot be solved analytically for arbitrary $U(x)$, we can establish a conservation law. Specifically, the total energy $E(x, \dot{x}) = U(x) + \frac{1}{2}\dot{x}^2$ is conserved:

$$\dot{x} \frac{d}{dt} \dot{x} + \frac{d\dot{x}}{dt} \frac{d}{d\dot{x}} U(x) = 0 \Rightarrow \frac{d}{dt} \left(\frac{1}{2}\dot{x}^2 + U(x) \right) = 0,$$

thus proving $\frac{d}{dt}E(x, \dot{x}) = 0$.

The phase portraits shown in Fig. 2.6 are generated by the Jupyter/Python notebook `double-well.ipynb`, which computes the total energy $E(x, \dot{x})$ on a grid and plots its level sets.

Double-Well Potential: Dynamics and Phase Portraits

A double-well potential is a classic example of a nonlinear system with rich dynamics. The potential is given by:

$$U(x) = \frac{x^4}{4} - \frac{x^2}{2}, \quad (2.9)$$

which has two minima at $x = \pm 1$ and a local maximum at $x = 0$. This potential models systems with bistable states, such as a particle trapped in two wells separated by a barrier. The corresponding equation of motion is:

$$\frac{d^2x}{dt^2} = -\frac{dU(x)}{dx} = -x^3 + x. \quad (2.10)$$

Phase Portraits in Different Regimes. The phase portraits for the double-well potential illustrate the rich dynamics of the system. Depending on the initial energy, the trajectories exhibit distinct behaviors:

- **Low Energy:** For energies below the barrier height ($E < \frac{1}{4}$), the particle remains confined to one of the wells. The phase portrait consists of closed orbits around $x = \pm 1$.
- **Barrier-Crossing Energy:** At the critical energy $E = \frac{1}{4}$, the particle reaches the top of the barrier at $x = 0$ and can transition between wells.
- **High Energy:** For $E > \frac{1}{4}$, the particle explores the entire potential landscape, crossing between wells repeatedly.

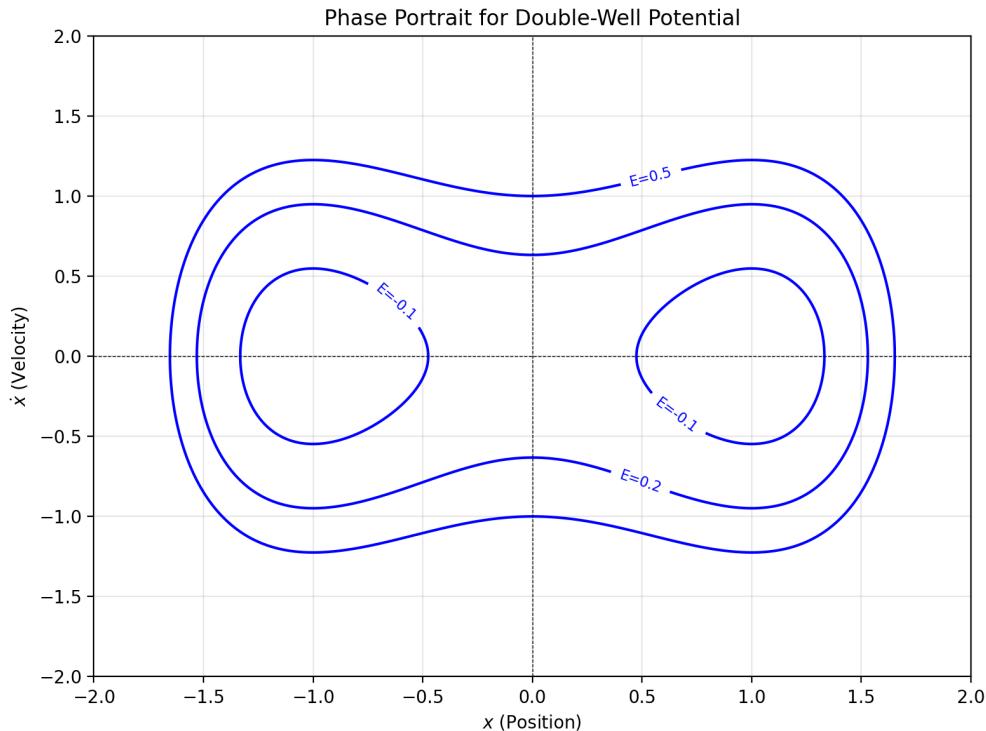


Figure 2.6: Phase portraits for the double-well potential $U(x) = \frac{1}{4}x^4 - \frac{1}{2}x^2$ at different energy levels. Contours of constant total energy $E(x, \dot{x})$ are shown in the (x, \dot{x}) plane. Generated by `double-well.ipynb`.

Damped Dynamics in the Double-Well Potential

Adding damping to the double-well system introduces energy dissipation, leading to relaxation into one of the potential wells. The equation of motion becomes:

$$\frac{d^2x}{dt^2} + \gamma \frac{dx}{dt} - x + x^3 = 0, \quad (2.11)$$

where $\gamma > 0$ represents the damping coefficient. The dynamics depend on the initial condition and the value of γ :

- **Low Damping ($\gamma \ll 1$):** The system exhibits oscillatory behavior before settling into one of the wells.
- **High Damping ($\gamma \gg 1$):** The system relaxes monotonically into one of the wells without oscillations.

This damped double-well system is widely used in physics – e.g., tunneling phenomena – and machine learning – e.g., modeling binary classification. In the latter case the two wells correspond to two distinct states or classes, with the damping term ensuring convergence to a stable state.

Exercise 2.2.4 (Damped Double-Well and Nonlinear Classification). *Consider the damped double-well system given by Eq. (2.11):*

1. *Numerically simulate the system for $\gamma = 0.1$, $\gamma = 1$, and $\gamma = 5$, with initial conditions $x(0) = 0.5$ and $\dot{x}(0) = 0$. Plot the trajectories $x(t)$. (You may use and modify the Python/Jupyter notebook `double-well.ipynb`, which was used to generate Fig. 2.6.)*
2. *Generate the phase portraits for each γ . Interpret the impact of damping on the trajectories in the (x, \dot{x}) -space.*
3. *Discuss the relevance of this system to nonlinear classification. Suppose $x > 0$ corresponds to one class and $x < 0$ corresponds to another. How does the damping influence the classification boundary and convergence?*
4. *Extend the system to a time-dependent classification boundary given by $x(t) = \cos(\omega t)$. Simulate the dynamics and discuss the implications for adaptive classification.*

From Low-Dimensional Dynamics to High-Dimensional AI

The analysis of second-order ODEs and the phase portrait method have shown how low-dimensional systems can exhibit rich, interpretable dynamics, such as oscillations and bistable states. However, in the realm of Generative AI, the challenge is typically reversed: we deal with extremely *high-dimensional* state vectors (e.g., millions of parameters or pixel values), where the governing dynamics are nearly always *nonlinear*. Since analytical solutions are impossible and direct numerical simulations of complex nonlinear systems are too slow at scale, we must rely on powerful approximation techniques. A cornerstone of these techniques is *linearization*. The remainder of this section pivots to the study of *Systems of Linear ODEs*, which, despite their simplicity, provide the fundamental mathematical machinery for understanding the local behavior, stability, and evolution of the vast, high-dimensional dynamical systems inherent in modern AI models.

2.3 System of Linear ODEs

Ordinary Differential Equations (ODEs) play a foundational role in understanding the dynamics of many systems in Artificial Intelligence (AI). They provide a mathematical framework for describing how quantities evolve over time in contexts such as optimization, data processing, and machine learning models. Some notable examples include:

- **Gradient Descent Dynamics:** The gradient descent algorithm, a cornerstone of optimization, can be viewed as a discrete approximation to a continuous gradient flow:

$$\frac{dx}{dt} = -\nabla f(x),$$

where $f(x)$ is the objective function being minimized.

- **Neural Network (NN) Training:** The training dynamics of NNs can often be modeled using ODEs, particularly in the case of continuous-time gradient flows, enabling theoretical insights into optimization and convergence behavior.

In practical AI applications, such as the examples above, the ODEs of interest typically involve high-dimensional vectors and are often nonlinear. However, as discussed in the previous section, solving nonlinear ODEs analytically is generally infeasible, even in one dimension. This necessitates the use of approximation methods and computational techniques, many of which rely on linearization of the equations.

Given the centrality of these challenges, this section focuses on systems of high-dimensional linear ODEs. By understanding linear systems, we build a foundation for tackling more complex nonlinear dynamics through approximation and computational strategies.

2.3.1 Homogeneous ODEs

We aim to solve a system of linear Ordinary Differential Equations (ODEs) that describes the time evolution of the vector $x \in \mathbb{R}^n$:

$$\frac{dx}{dt} = Ax, \quad (2.12)$$

where $A \in \mathbb{R}^{n \times n}$ is a fixed, time-independent matrix. The system is subject to the initial condition

$$x(0) = x_0.$$

When $n = 1$, A is a scalar, and the solution is straightforward:

$$x(t) = \exp(tA)x_0. \quad (2.13)$$

Remarkably, this solution generalizes to the case of $n > 1$, where $\exp(tA)$ is the *matrix exponential*, defined as:

$$\exp(tA) = \sum_{k=0}^{\infty} \frac{(tA)^k}{k!}.$$

The series converges for any finite-dimensional square matrix A , ensuring the matrix exponential is well-defined. Substituting $x(t) = \exp(tA)x_0$ into Eq. (2.12) verifies that this indeed satisfies the equation:

$$\frac{dx}{dt} = A \exp(tA)x_0 = Ax(t).$$

Example 2.3.1 (Linear Diffusion on a Short 1D Grid). *A simple but instructive instance of Eq. (2.12) arises from a finite-difference discretization of the 1D heat equation on a short spatial grid. Let $x(t) \in \mathbb{R}^n$ collect the temperature at n grid points, and consider*

$$\dot{x}(t) = Ax(t),$$

where $A \in \mathbb{R}^{n \times n}$ is the discrete Laplacian with (approximate) Neumann boundary conditions:

$$A_{ii} = -2, \quad A_{i,i+1} = A_{i,i-1} = 1, \quad i = 2, \dots, n-1,$$

with suitable edge modifications at $i = 1$ and $i = n$.

Starting from a spiky initial condition (all zeros except a single large entry in the middle), the solution $x(t)$ evolves by progressively smoothing out the spike as heat diffuses along the grid. This illustrates how a matrix with negative diagonal and positive off-diagonal entries can generate a contracting semigroup $x(t) = \exp(tA)x_0$ that drives the system toward a more uniform state.

Fig. 2.7 shows snapshots $x(t)$ at several times, while Fig. 2.8 displays the full space-time evolution as a heatmap. Both figures are generated by the Python notebook `linear_systems.ipynb`.

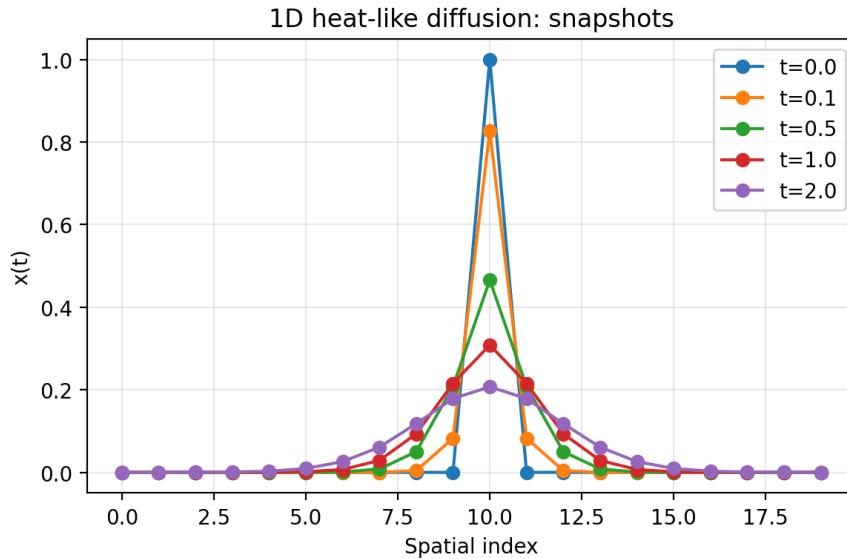


Figure 2.7: Snapshots of the state $x(t)$ for the 1D discrete heat equation on a short grid, starting from a single spike. Over time the profile smooths out, illustrating diffusion governed by a linear system $\dot{x} = Ax$. Generated by `linear_systems.ipynb`.

Exercise 2.3.1 (Properties of the Matrix Exponential). *Prove that:*

1. if $AB - BA = 0$, then $\exp(A + B) = \exp(A)\exp(B) = \exp(B)\exp(A)$;
2. if $\det(P) \neq 0$, then $\exp(PAP^{-1}) = P\exp(A)P^{-1}$;
3. $\det(\exp(A)) = \exp(\text{Tr}(A))$. In your proof, you may assume that A is diagonalizable. For an extra challenge – prove the statement without the assumption that A is diagonalizable.

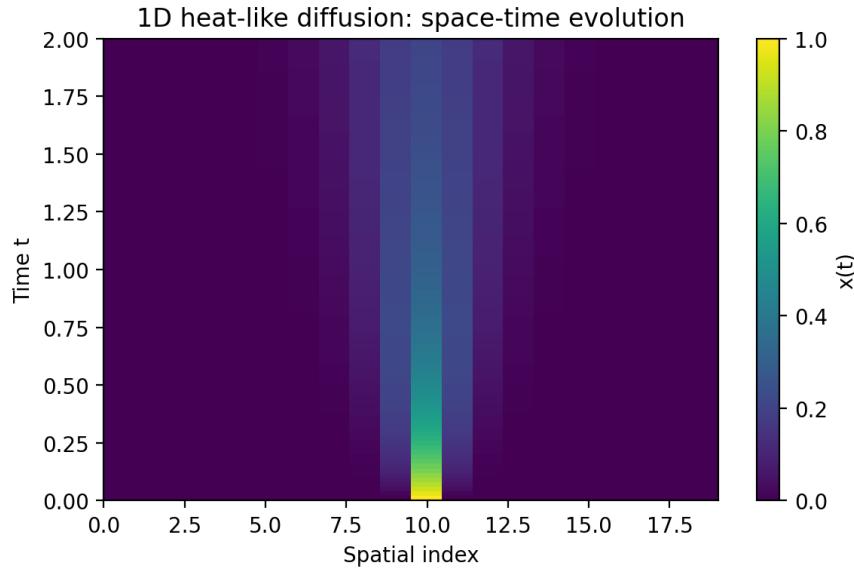


Figure 2.8: Space–time evolution of the 1D heat-like system from Example 2.3.1. The vertical axis is time and the horizontal axis is the spatial index. The initial spike spreads out as heat diffuses across the grid. Generated by `linear_systems.ipynb`.

2.3.2 Inhomogeneous ODEs

Now consider the generalization to an inhomogeneous system:

$$\frac{dx}{dt} = Ax + b(t),$$

where $b(t) \in \mathbb{R}^n$ is a time-dependent vector. The solution method leverages the linear structure of Eq. (2.13). Substituting $x(t) = \exp(tA)\tilde{x}(t)$ into the inhomogeneous equation leads to an ODE for $\tilde{x}(t)$:

$$\frac{d\tilde{x}}{dt} = \exp(-tA)b(t).$$

This equation is straightforward to integrate, and reversing the substitution yields the solution:

$$x(t) = \exp(tA)x_0 + \int_0^t \exp((t-\tau)A)b(\tau) d\tau,$$

where the first term solves the homogeneous equation, and the second term accounts for the inhomogeneous component.

Case of Constant b : If $b(t) = b$ is constant, the time-dependent integral simplifies, resulting in the algebraic expression:

$$x(t) = \exp(tA)x_0 + A^{-1}(\exp(tA) - I)b, \quad (2.14)$$

which expresses the solution using elementary matrix functions (exponential and inverse).

Example 2.3.2 (Forced Linear System in \mathbb{R}^3). Consider the system

$$\dot{x}(t) = Ax(t) + b, \quad x(t) \in \mathbb{R}^3,$$

with a stable matrix $A \in \mathbb{R}^{3 \times 3}$ (all eigenvalues have negative real part) and a constant forcing vector b . In this setting, Eq. (2.14) predicts that the solution is a sum of a transient term $\exp(tA)x_0$ and a steady-state term

$$\bar{x} = -A^{-1}b,$$

to which all trajectories converge as $t \rightarrow \infty$.

The notebook `linear_systems.ipynb` constructs such an A and b , integrates the ODE numerically, and overlays the time series of each component $x_i(t)$ with its corresponding steady-state value \bar{x}_i . As shown in Fig. 2.9, the trajectories exhibit an initial transient followed by convergence to the constant equilibrium \bar{x} .

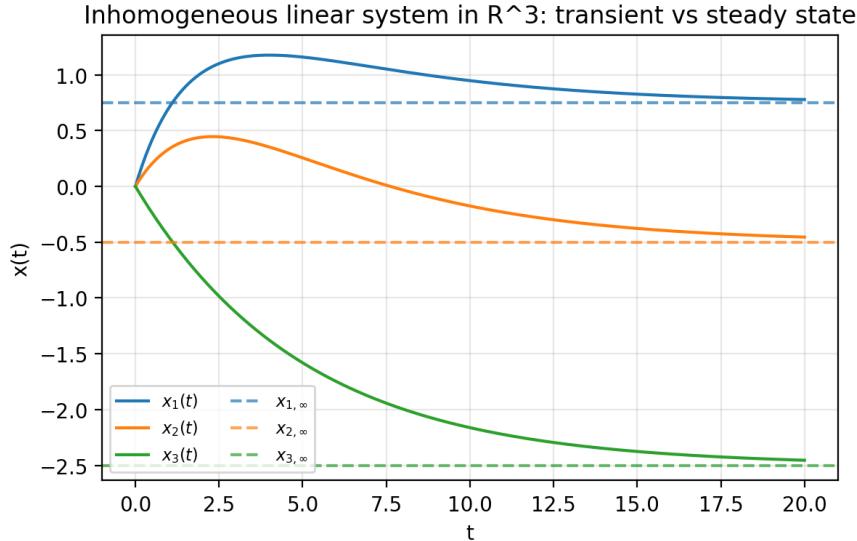


Figure 2.9: Forced linear system in \mathbb{R}^3 : components $x_1(t), x_2(t), x_3(t)$ (solid lines) and their steady-state values \bar{x}_i (dashed lines). All trajectories converge to the equilibrium $\bar{x} = -A^{-1}b$. Generated by `linear_systems.ipynb`.

To explore the solution further, we use the Eigenvalue Decomposition (ED) of A :

$$A = Q\Lambda Q^{-1},$$

where $Q \in \mathbb{R}^{n \times n}$ is an invertible matrix whose columns are the eigenvectors of A , and $\Lambda = \text{diag}(\lambda_1, \lambda_2, \dots, \lambda_n)$ is a diagonal matrix containing the eigenvalues of A .

For symmetric matrices, Q is orthogonal ($Q^{-1} = Q^\top$), and the eigenvalues are real but may be positive, negative, or zero. For non-symmetric matrices, Q is not necessarily orthogonal, and the eigenvalues may be complex.

Substituting the ED of A into $\exp(A)$, we find:

$$\begin{aligned}\exp(tA) &= \sum_{k=0}^{\infty} \frac{(tA)^k}{k!} = \sum_{k=0}^{\infty} \frac{t^k}{k!} \underbrace{Q\Lambda Q^{-1} \cdots Q\Lambda Q^{-1}}_{k \text{ times}} = Q \left(\sum_{k=0}^{\infty} \frac{(t\Lambda)^k}{k!} \right) Q^{-1} = Q \exp(t\Lambda) Q^{-1} \\ &= Q [\text{diag}(\exp(t\lambda_1), \dots, \exp(t\lambda_n))] Q^{-1},\end{aligned}$$

then arriving at the following simplification of Eq. (2.14)

$$x(t) = \exp(tA)x_0 + Q \left[\text{diag} \left(\frac{\exp(t\lambda_1) - 1}{\lambda_1}, \dots, \frac{\exp(t\lambda_n) - 1}{\lambda_n} \right) \right] Q^{-1} b.$$

What if A^{-1} Does Not Exist? If A is singular, meaning that one or more eigenvalues $\lambda_i = 0$, the above formula remains valid. The key lies in handling the term $(\exp(t\lambda_i) - 1)/\lambda_i$ for $\lambda_i = 0$. Using L'Hôpital's rule, we resolve the limit

$$\lim_{\lambda_i \rightarrow 0} \frac{\exp(t\lambda_i) - 1}{\lambda_i} = t.$$

This shows that the ED-based approach is robust and well-defined, even when A^{-1} does not exist.

Exercise 2.3.2. If we use SVD (and not ED as above) working with Eq. (2.14), does it lead to a simplification? To validate your answer consider, $A = \begin{bmatrix} 2 & 1 \\ 1 & -1 \end{bmatrix}$, derive ED and SVD for the matrix and see if either of the two allows to simplify $\exp(tA)$. (Hint: Pay attention to the sign of the eigen-values.)

2.3.3 Dynamics over Graph

For a symmetric matrix A with stochastic-like properties – e.g., a *graph Laplacian*, where *stochastic* does not imply randomness but rather refers to a structure in which all matrix elements are non-negative and each row sums to unity – the non-negativity of eigenvalues plays a crucial role in governing the system's behavior. Specifically, the non-negative spectrum of A ensures that the system's dynamics, described by the evolution matrix $\exp(-tA)$, asymptotically relaxes to an equilibrium state. This equilibrium corresponds to the eigenvector associated with the zero eigenvalue of A and it is reached in the limit $t \rightarrow \infty$.

This property is widely utilized in applications such as community detection and clustering algorithms, where the dynamics governed by the graph Laplacian can reveal the underlying structure of the data or network. The gradual relaxation highlights the separation of clusters, aiding in their identification.

Example 2.3.3 (Consensus Dynamics on a Ring Graph). Consider a simple undirected graph with $n = 5$ nodes arranged in a ring. The adjacency matrix A has entries $A_{ij} = 1$ if nodes i and j are neighbors on the ring and 0 otherwise. The degree matrix D is diagonal with $D_{ii} = \sum_j A_{ij}$, and the graph Laplacian is $L = D - A$.

We study the consensus dynamics

$$\dot{x}(t) = -Lx(t),$$

with an arbitrary initial condition $x(0) = x_0 \in \mathbb{R}^5$. Because L is positive semi-definite and has a single zero eigenvalue corresponding to the constant vector, the solution $x(t)$ converges to a consensus state where all components are equal:

$$x_i(t) \rightarrow \bar{x} = \frac{1}{n} \sum_{j=1}^n x_j(0), \quad t \rightarrow \infty.$$

The notebook `graph_dynamics.ipynb` simulates this ODE using a forward Euler scheme. Fig. 2.10 shows the node values $x_i(t)$ versus time, while Fig. 2.11 visualizes the initial and final states on the ring. The final configuration clearly illustrates convergence to a uniform consensus.

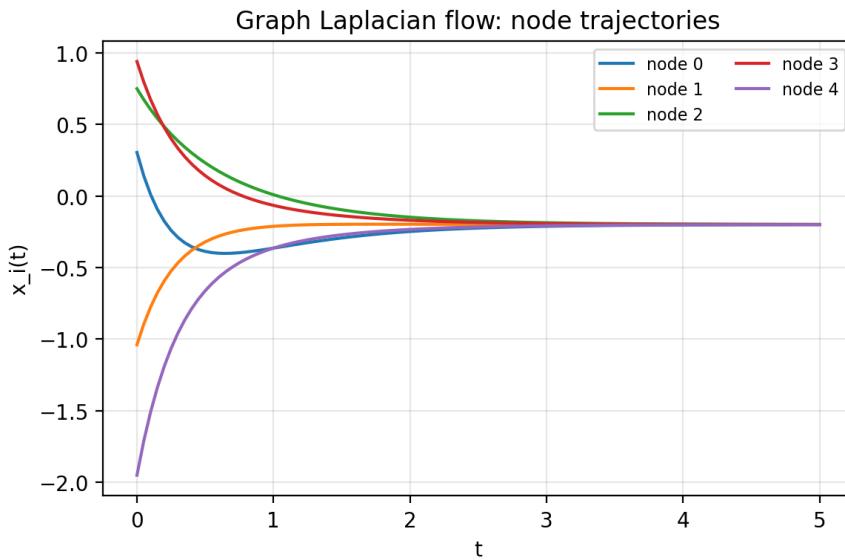


Figure 2.10: Consensus dynamics on a 5-node ring: trajectories of each node value $x_i(t)$ under $\dot{x} = -Lx$. All states converge to a common value (consensus). Generated by `graph_dynamics.ipynb`.

Exercise 2.3.3 (Analyzing Linear ODEs on Graphs). This exercise focuses on understanding and solving linear ODEs using graph-based dynamics.

1. **Graph-Laplacian Dynamics:** Consider a graph with n nodes and a symmetric weighted adjacency matrix A . Let the degree matrix D be diagonal with entries $D_{ii} = \sum_j A_{ij}$. The graph Laplacian is defined as $L = D - A$.

- (a) Write the ODE governing the evolution of node values $x(t)$:

$$\frac{dx(t)}{dt} = -Lx(t),$$

in components.

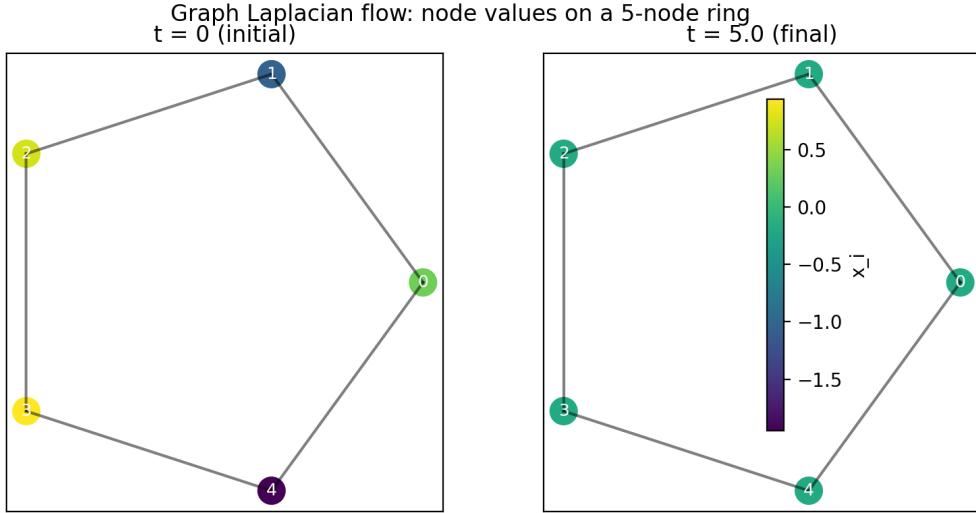


Figure 2.11: Graph representation of consensus dynamics on a 5-node ring. Left: node values at $t = 0$; right: node values at a large time t , after consensus is reached. Nodes are placed on a circle and colored according to their scalar state value. Generated by `graph_dynamics.ipynb`.

- (b) Solve the ODE analytically using eigen-decomposition for an initial condition $x(0)$.
 - (c) Discuss the long-term behavior of $x(t)$ in terms of the eigenvalues of L .
2. **Example with a Small Graph:** Consider a graph with 3 nodes and the adjacency matrix:
- $$A = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}.$$
- (a) Compute the degree matrix D and the graph Laplacian L .
 - (b) Solve the ODE $\frac{dx(t)}{dt} = -Lx(t)$ for $x(0) = [1, 0, 0]^\top$.
 - (c) Plot $x(t)$ for $t \in [0, 10]$ and interpret the results.

3. **Energy and Gradient Flow:** Consider the quadratic energy function:

$$f(x) = \frac{1}{2}x^\top Lx.$$

- (a) Derive the gradient flow ODE: $\frac{dx(t)}{dt} = -\nabla f(x)$.
- (b) Simulate the gradient flow numerically for the graph given above. Compare the numerical solution to the analytical solution obtained using ED.
- (c) Discuss the role of eigenvalues in determining the dynamics of $x(t)$.

2.3.4 Time-Ordered Exponential

Let us now discuss solution of $\dot{x} = Ax$, where $x \in \mathbb{R}^n$ and $A(t) \in \mathbb{R}^{n \times n}$, in the case where $A = A(t)$ is time-dependent. For $n = 1$, the solution is straightforward: the matrix exponential $\exp(tA)$ is replaced by: $\exp\left(\int_0^t A(t') dt'\right)$. However, it is also straightforward to check by substitution that for $n > 1$, this expression is incorrect in general, and specifically when the matrix function $A(t)$ evaluated at two different moments of time, t_1 and t_2 , do not commute. To account for this matrix non-commutativity, we replace the matrix exponential with the so-called *time-ordered exponential*. Unlike the case of time-independent A , this object cannot be expressed as a simple function of $A(t)$ and should be viewed as defined in terms of a Taylor series:

$$\begin{aligned}\mathcal{T} \exp\left(\int_0^t A(t') dt'\right) &= I + \int_0^t A(t_1) dt_1 + \int_0^t \int_0^{t_1} A(t_1)A(t_2) dt_2 dt_1 + \dots \\ &= \sum_{n=0}^{\infty} \int_0^t \dots \int_0^{t_{n-1}} A(t_1)A(t_2) \dots A(t_n) dt_n \dots dt_1,\end{aligned}\quad (2.15)$$

where \mathcal{T} indicates time-ordering, ensuring that matrices $A(t_1), A(t_2), \dots$ are multiplied in the correct chronological order, with earlier times appearing to the right.

Example 2.3.4 (Naive vs Time-Ordered Exponential for a Rotating System). *To visualize the difference between a naive exponential and the true time-ordered evolution, consider the 2D system*

$$\dot{x}(t) = A(t)x(t), \quad A(t) = \begin{bmatrix} 0 & -\omega(t) \\ \omega(t) & 0 \end{bmatrix}, \quad \omega(t) = 1 + 0.5 \sin t.$$

For each fixed t , the matrix $A(t)$ generates a rotation with instantaneous angular velocity $\omega(t)$. However, the matrices $A(t_1)$ and $A(t_2)$ do not commute at different times because $\omega(t)$ changes.

The notebook `timeordered.ipynb` compares:

- the true solution, obtained by time-stepping with small increments (approximating the time-ordered exponential), and
- a naive solution that treats the integral $\int_0^t \omega(\tau) d\tau$ as a single effective rotation angle and applies a simple rotation matrix of that angle.

Fig. 2.12 shows the components $x_1(t), x_2(t)$ over time for both solutions, while Fig. 2.13 compares the corresponding trajectories in phase space. The discrepancy between the two curves illustrates how ignoring time ordering can lead to noticeable errors when the generators $A(t)$ do not commute.

Exercise 2.3.4. Let $A(t) \in \mathbb{R}^{n \times n}$ be a continuous, time-dependent matrix defined for $t \in [0, T]$ and defined by Eq. (2.15).

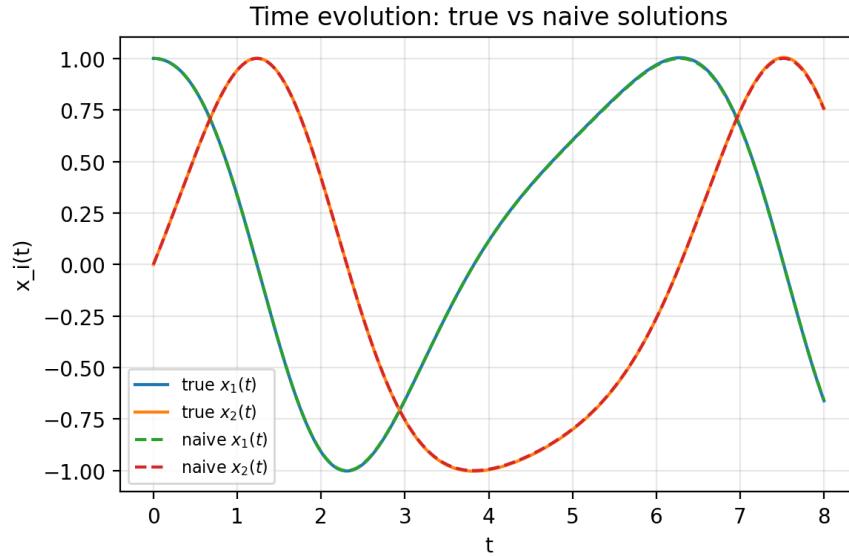


Figure 2.12: Time evolution of $x_1(t)$ and $x_2(t)$ for the true (time-stepped) solution and the naive exponential solution that ignores time ordering. The mismatch shows the effect of non-commuting $A(t)$. Generated by `timeordered.ipynb`.

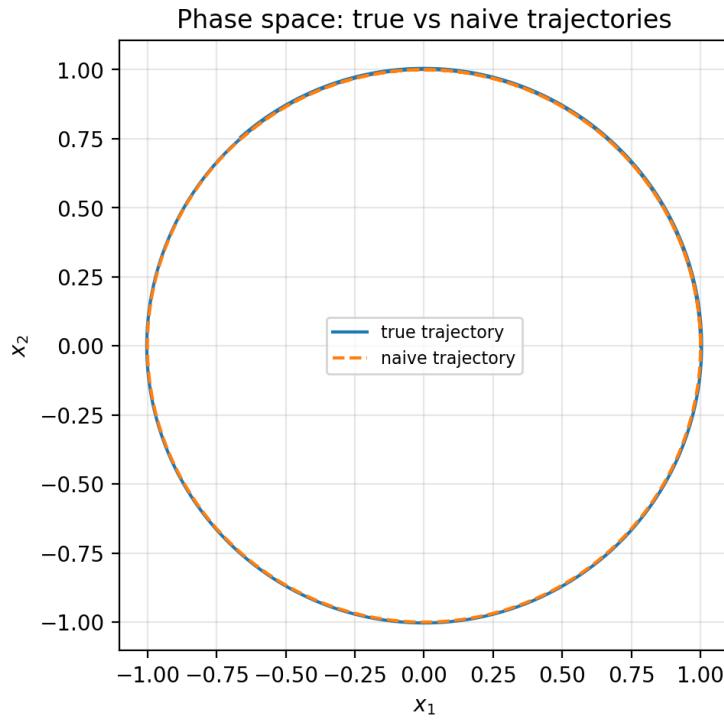


Figure 2.13: Phase-space trajectories $(x_1(t), x_2(t))$ for the true time-ordered dynamics and the naive approximation. Even though both are rotations in the plane, the paths diverge due to the time dependence of $\omega(t)$. Generated by `timeordered.ipynb`.

1. Prove that the Taylor expansion for the time-ordered exponential satisfies the differential equation:

$$\frac{d}{dt} \mathcal{T} \exp \left(\int_0^t A(t') dt' \right) = A(t) \mathcal{T} \exp \left(\int_0^t A(t') dt' \right),$$

with the initial condition:

$$\mathcal{T} \exp \left(\int_0^0 A(t') dt' \right) = I.$$

Hint: Expand the Taylor series term by term and verify the differential equation by differentiating under the integral sign. Use the fact that the limits of the integrals enforce time-ordering.

2. Consider the time-dependent matrix:

$$A(t) = \begin{bmatrix} 0 & -t \\ t & 0 \end{bmatrix}.$$

Compute the time-ordered exponential explicitly for $t \in [0, T]$ up to the second-order term of the Taylor series:

$$\mathcal{T} \exp \left(\int_0^T A(t') dt' \right) \approx I + \int_0^T A(t_1) dt_1 + \int_0^T \int_0^{t_1} A(t_1) A(t_2) dt_2 dt_1.$$

The Mathematics of Dynamics and the Path to Optimization

This chapter established the two core mathematical engines of modern AI: **Automatic Differentiation (AD)**, which provides the instantaneous gradient, and **Differential Equations**, which model the evolution of a system over continuous time.

The final section on linear systems, particularly the formula for the Time-Ordered Exponential, concludes our exploration of how high-dimensional dynamics are solved. Critically, these dynamics are directly linked to optimization: the core algorithm of **Gradient Descent** is simply a discrete approximation of the continuous **Gradient Flow** ODE, $\dot{x} = -\nabla f(x)$.

Moving to Optimization: We have the tool (the gradient, provided by AD) and the language (ODEs, for dynamics). The next step is to use the gradient iteratively to find minima. However, this is where the computational limits of high-dimensional AI become the primary design constraint. In the next chapter we will put these skills at work to analyze complex energy landscapes subject to optimization, in particular building the practical, first-order optimization algorithms (SGD, Adam) that dominate modern machine learning.

Chapter 3

Optimization (in AI)

Overview

Optimization underpins most advances in Artificial Intelligence (AI). Whether training a Neural Network (NN), fine-tuning a transformer, or building a generative diffusion model, optimization is at the heart of creating, training, and performing inference with these models. In this chapter, we focus on the principles and techniques of unconstrained optimization, a dominant paradigm in modern AI, where the complexity of the problem is encapsulated entirely in the energy (or loss) landscape.

Why Optimization in AI?

AI workflows involve three major stages:

1. **Model Creation:** Formulating the model structure and the optimization problem.
2. **Training:** Optimizing over parameters of the model using a fixed dataset.
3. **Inference:** Finding optimal states or predictions with parameters already fixed trained and with optimization over the model state space with the energy function (landscape) fixed or evolving in the (algorithmic) time.

Of these, the training stage demands the most computational resources, as it involves optimizing a super high-dimensional, continuous parameter space. Modern state-of-the-art AI models often contain billions or even trillions of parameters (of the underlying neural networks), making scalability and efficiency critical. This is why optimization methods in AI primarily focus on:

- **Continuous Spaces:** Most models have continuous parameters.
- **Single-Objective Optimization:** Only one objective (e.g., minimizing loss) is considered.
- **First-Order Methods:** Gradients (but not second derivatives) are used to ensure scalability (as evaluating Jacobians – second order derivatives - scales quadratically with the number of parameters, and thus not feasible).

- **Unconstrained Settings:** Constraints are avoided, accounted in the loss/cost function via the so-called Lagrange multiplier methodology to allow efficient use of the Automatic Differentiation (AD) - see Section 2.1, or used as guard rails within the first order methods.

Other optimization paradigms, such as discrete optimization or higher-order methods, are occasionally employed but primarily as supplements (for pre-processing or down-stream tasks) to improve scalability, robustness, or accuracy.

3.1 Starting Example — Logistic Regression

Optimization is at the core of modern AI. Virtually every supervised-learning task – from logistic regression to large-scale deep learning – requires adjusting model parameters to *minimize a loss function* computed over a dataset. For a Neural Network (NN) $f_\theta(x)$, the standard learning problem takes the form

$$\theta^* = \arg \min_{\theta} \frac{1}{N} \sum_{i=1}^N \mathcal{L}(f_\theta(x_i), y_i), \quad (3.1)$$

where $\{(x_i, y_i)\}_{i=1}^N$ is the dataset, \mathcal{L} is a loss (e.g., MSE or cross-entropy), and θ are the parameters to be optimized.

Neural Networks will be studied in depth in Chapter 4. Here, to build intuition for optimization in a simple yet realistic setting, we focus on a classical model: *Logistic Regression* (LR). Despite its simplicity, LR already illustrates many important ideas: convex loss functions, decision boundaries, linear separability, feature engineering, and the geometry of optimization landscapes.

3.1.1 The Logistic Regression Model

In its simplest 2D form, logistic regression has parameters $\omega = (b, \omega_1, \omega_2)$, and predicts the probability of class $y = 1$ via

$$\hat{y}_{LR}(x, \omega) = \sigma(b + \omega_1 x_1 + \omega_2 x_2), \quad \sigma(z) = \frac{1}{1 + e^{-z}}, \quad (3.2)$$

where $x = (x_1, x_2) \in \mathbb{R}^2$ and σ is the sigmoid activation function ¹.

¹The sigmoid function was originally introduced in the 19th century by the Belgian mathematician Pierre François Verhulst (1838) in the context of population growth modeling. He used it to describe how growth starts rapidly but slows as it approaches a natural limit due to constrained resources. In modern AI and machine learning, the sigmoid function is often called the logistic function, as it smoothly interpolates between 0 and 1, making it useful for probability estimation and binary classification. The term "logistic" comes from the Greek word *logistikos* (λογιστικός), meaning "skilled in calculation" or "rational reasoning." Though originally referring to mathematical logic, its adoption in this context likely reflects the function's role in making smooth, calculated transitions between discrete states – 0 and 1.

Given labeled samples $\{(x_i, y_i)\}_{i=1}^N$, the parameters are estimated by solving

$$\omega^* = \arg \min_{\omega} \frac{1}{N} \sum_{i=1}^N \mathcal{L}_{LR}(x_i, y_i, \omega), \quad (3.3)$$

with the *binary cross-entropy loss*

$$\mathcal{L}_{LR}(x_i, y_i, \omega) = - \left[y_i \log \hat{y}_{LR}(x_i, \omega) + (1 - y_i) \log(1 - \hat{y}_{LR}(x_i, \omega)) \right]. \quad (3.4)$$

Why cross-entropy? Cross-entropy measures the *information mismatch* between predicted probabilities and true labels:

- confident and correct predictions \rightarrow loss ≈ 0 ,
- confident and wrong predictions \rightarrow loss $\rightarrow \infty$
- uncertain predictions (≈ 0.5) \rightarrow moderate loss.

A full information-theoretic interpretation will appear later in Chapter 5.

3.1.2 Linear Logistic Regression and Its Limitations

Define the linear feature map

$$\phi_{LR}(x) = (1, x_1, x_2)^\top.$$

After training, the decision rule is

$$y_{LR}(x, \omega^*) = \begin{cases} 1, & (\omega^*)^\top \phi_{LR}(x) \geq 0, \\ 0, & (\omega^*)^\top \phi_{LR}(x) < 0. \end{cases} \quad (3.5)$$

Thus the decision boundary is the *hyperplane*

$$(\omega^*)^\top \phi_{LR}(x) = 0.$$

Conclusion: **Logistic regression is always a linear classifier.** Its separator in feature space is a straight line (in 2D), a plane (in 3D), or, more generally, a hyperplane.

3.1.3 Why Linear Logistic Regression Fails for Non-Linearly Separable Data

Many real-world datasets are not linearly separable. The XOR dataset provides the simplest illustration:

$$(0, 0) \rightarrow 0, \quad (0, 1) \rightarrow 1, \quad (1, 0) \rightarrow 1, \quad (1, 1) \rightarrow 0.$$

No single straight line can separate the classes. Thus *linear LR fails* because:

- it can only produce monotonic decision functions,
- its decision boundary is always linear.

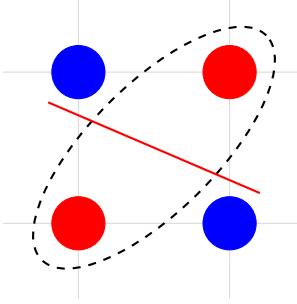


Figure 3.1: XOR dataset: the dashed nonlinear separator works; a linear separator (red) fails. Linear logistic regression cannot solve XOR.

3.1.4 Gradient Descent for Logistic Regression: The Vector Field

Optimization algorithms such as Gradient Descent (GD) operate by iteratively updating the parameter vector ω in the direction of steepest descent of the loss. For logistic regression, the loss landscape is smooth and convex, making it an ideal setting to visualize the *vector field* of the gradient.

Studying this vector field is especially useful because:

- it reveals how optimization “flows” toward the optimum;
- it highlights anisotropy in curvature (directions of flatness vs. sharpness);
- it provides a geometric viewpoint on why GD converges reliably for LR;
- it introduces the dynamical-systems perspective foundational for continuous-time optimization, Neural ODEs, and diffusion models.

Setup (2D visualization). To produce a 2D visualization, we restrict attention to a logistic regression model with two parameters, keeping the bias fixed:

$$\omega = (\omega_1, \omega_2).$$

Given data points (x_i, y_i) , the loss is

$$\mathcal{L}(\omega) = -\frac{1}{N} \sum_{i=1}^N \left[y_i \log \sigma(\omega^\top x_i) + (1 - y_i) \log (1 - \sigma(\omega^\top x_i)) \right].$$

The gradient has closed form:

$$\nabla_\omega \mathcal{L}(\omega) = \frac{1}{N} \sum_{i=1}^N (\sigma(\omega^\top x_i) - y_i) x_i.$$

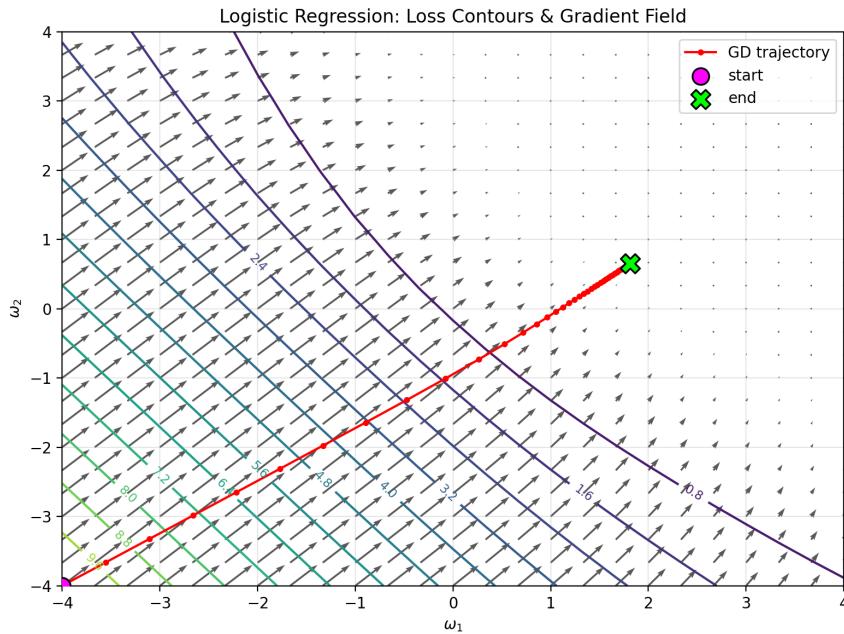
Gradient Flow Interpretation. Consider the continuous-time gradient flow ODE

$$\frac{d\omega}{dt} = -\nabla_{\omega}\mathcal{L}(\omega).$$

This ODE defines a vector field over the (ω_1, ω_2) -plane. Arrows point toward decreasing loss, and their magnitude encodes the steepness. For convex logistic regression, all trajectories converge to the unique optimum ω^* .

Example 3.1.1 (Linear Regression Gradient Field). *The accompanying notebook LR-gradient-field.ipynb constructs a simple 2D dataset, computes the gradient field on a grid, plots the loss contours, projects gradient flow trajectories, and performs discrete gradient descent for comparison with the continuous field.*

Fig. 3.2 shows a typical output: contour lines of the loss together with gradient arrows and a GD trajectory.



4. Add a bias parameter b (making the parameter space 3D) and visualize 2D slices. Discuss how the additional degree of freedom alters convergence.

3.1.5 Nonlinear Logistic Regression via Feature Engineering

A classical strategy – predating neural networks – is to enrich the feature map. Introduce a *quadratic feature map*

$$\phi_Q(x) = (x_1, x_2, x_1^2, x_2^2, x_1 x_2)^\top.$$

The nonlinear logistic regression model becomes

$$P(y = 1 | x, \omega) = \sigma((\phi_Q(x))^\top \omega),$$

which is trained by the same convex optimization problem as in Eq. (3.3), but with $\phi_{LR}(x)$ replaced by $\phi_Q(x)$.

The resulting decision boundary

$$(\phi_Q(x))^\top \omega^* = 0$$

is a *quadratic curve*: ellipse, parabola, hyperbola, etc.

Key advantages of nonlinear features:

- nonlinear decision boundaries,
- feature–feature interactions,
- ability to handle non-linearly separable datasets.

Example 3.1.2 (Linear vs. Quadratic Logistic Regression). We visualize these ideas using the notebook *LogReg+NN-supervised-2D.ipynb*. The dataset consists of two classes forming an elliptically shaped separation boundary.

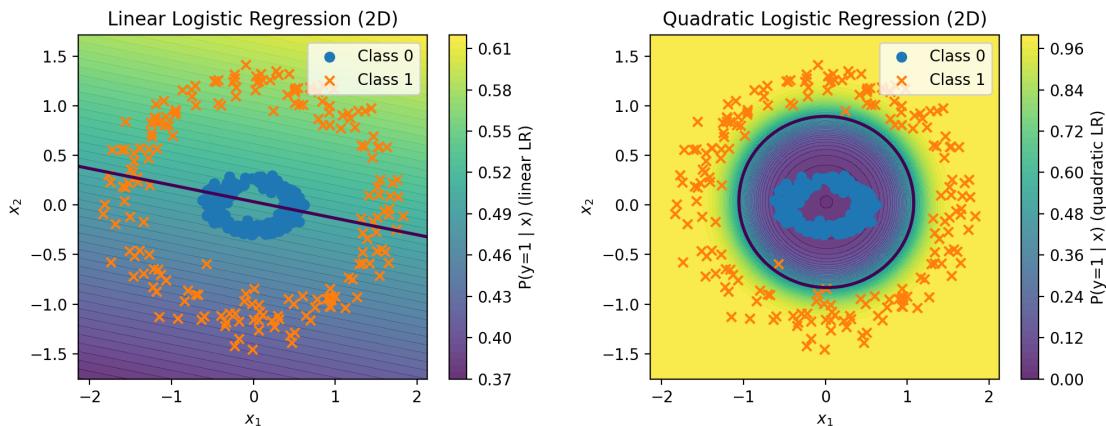


Figure 3.3: Left: linear logistic regression fails to separate classes with elliptic structure. Right: quadratic logistic regression succeeds by learning a curved boundary. Both plots generated by *LogReg2D.ipynb*.

The comparison clearly shows:

- The linear model learns the best straight-line separator – but it is fundamentally mismatched.
- The quadratic model learns a curved boundary that perfectly aligns with the data geometry.

This example demonstrates the power of nonlinear features:

Nonlinearity in the feature space enables linear models to solve nonlinear classification problems.

Exercise 3.1.2 (Exploring Logistic Regression with Polynomial Features). *Using the notebook LogReg2D.ipynb:*

1. Train both linear and quadratic logistic regression models. Compare classification accuracy, loss decay, and decision boundaries.
2. Perform a sensitivity analysis: perturb one entry of ω^* while holding others fixed, and plot how the loss changes.
3. Extend the feature map to include cubic terms. Visualize the resulting boundary and discuss overfitting tendencies.

3.2 Convex Optimization – Primer

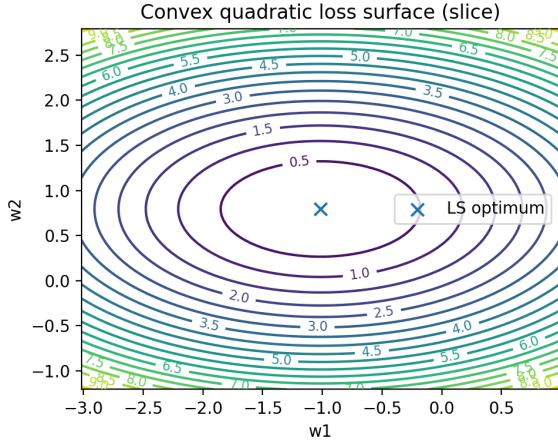
Optimization is at the heart of modern AI. Whether training a neural network, fitting a regression model, or solving a reinforcement learning problem, we repeatedly search for parameters that minimize a loss function. Yet the difficulty of this search strongly depends on the *shape* of the loss landscape, and that shape is governed by **convexity**.

Convex optimization problems enjoy beautiful mathematical guarantees: every local minimum is a global minimum, gradient descent converges reliably, and duality theory offers deep structural insights. In contrast, most AI models — in particular neural networks — lead to *non-convex* landscapes with many flat regions, narrow valleys, and saddle points. Understanding convexity therefore provides a clean baseline from which we can appreciate the complexity of non-convex optimization in AI.

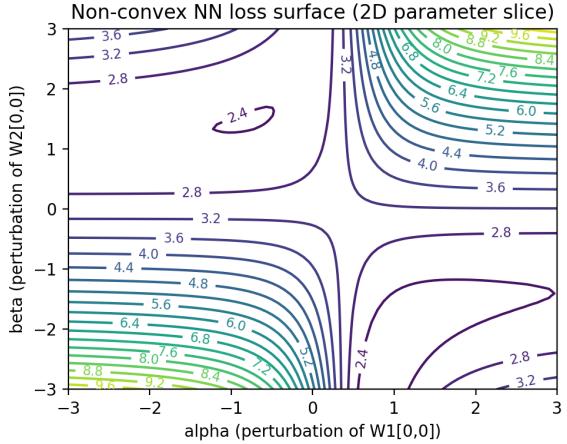
3.2.1 Variety of (Non-Convex) Landscapes

Optimization landscapes encountered in AI typically include:

- **Global minima:** Ideal convergence points where the loss is smallest.
- **Local minima:** Stationary points that are not global minima. In high dimensions, most local minima of large neural networks are “good enough” and often comparable to the global minimum.
- **Saddle points:** Points where curvature is positive in some directions and negative in others. These dominate high-dimensional non-convex landscapes.



(a) Convex quadratic loss landscape.



(b) Non-convex neural network loss landscape.

Figure 3.4: Convex vs non-convex loss landscapes. Both contour plots are produced by the notebook `Convex_vs_NonConvex_Landscapes.ipynb`. **Left:** The mean-squared error of a linear regression model, shown as a function of its two parameters (w_1, w_2), forms a strictly convex quadratic bowl with a unique global minimizer. **Right:** A two-parameter slice of the loss surface of a tiny one-hidden-layer neural network. The resulting landscape exhibits non-convex features such as distorted level sets, saddle-like regions, and multiple attraction basins. These differences illustrate why convex optimization behaves predictably, while neural-network training encounters the full complexity of non-convex geometry.

- **Flat regions (plateaus):** Regions where gradients vanish or are very small, causing slow optimization.
- **Narrow valleys:** Curvature varies dramatically across directions, forcing small learning rates and slowing gradient methods.

Even simple two-layer neural networks display many of these features. Convex optimization provides tools and intuition that help us navigate and mitigate such difficulties.

Example 3.2.1 (Convex vs non-convex loss landscapes). *To visualize the difference between convex and non-convex optimization problems, consider the Jupyter/Python notebook `Convex_vs_NonConvex_Landscapes.ipynb`. The notebook constructs and compares two loss landscapes:*

- *A convex quadratic loss corresponding to least-squares linear regression in 2D. We fit a linear model $\hat{y}(x; w) = w_1 x_1 + w_2 x_2$, to a small synthetic dataset. The resulting mean-squared error loss $L_{\text{lin}}(w_1, w_2)$ is a strictly convex quadratic function of the parameters. Its level sets are ellipses, and there is a unique global minimizer. The contour plot of this loss, as a function of (w_1, w_2) , is shown on the left panel of Fig. (3.4).*
- *A non-convex loss arising from a tiny one-hidden-layer neural network. We keep most parameters fixed and vary only two of them (for example, one weight in the hidden layer and one weight in the output layer). Evaluating the mean-squared error as a*

function of these two parameters defines a 2D slice of the full NN loss landscape. This slice exhibits multiple valleys, saddle-like regions, and non-elliptic level sets, illustrating non-convex behavior. The corresponding contour plot is shown on the right panel of Fig. (3.4).

Comparing the two figures highlights a key structural difference:

- In the convex quadratic case, the loss landscape is “bowl-shaped,” with smooth, nested ellipses and a single global minimum.
- In the neural network case, the same least-squares objective becomes non-convex in the parameters, showing distorted contours, flat regions, and directions of negative curvature.

Exercise 3.2.1 (Gradient Descent Trajectories in Convex and Non-Convex Landscapes). This exercise complements Example 3.2.1 and the Jupyter/Python notebook `Convex_vs_NonConvex_Landscapes.ipynb`. Extend the notebook as follows.

1. Gradient Descent on the Convex Quadratic.

- Implement vanilla gradient descent on the convex quadratic loss used in the linear-regression part of the notebook.
- Choose several distinct initial parameter vectors $(w_1^{(0)}, w_2^{(0)})$ and run gradient descent from each.
- On top of the convex contour plot `figs/convex_quadratic_loss_contours.png`, overlay the corresponding gradient-descent trajectories in the (w_1, w_2) -plane.
- Comment on your observations: do all trajectories converge smoothly to the same global minimizer? How does the step size (learning rate) influence the path and speed of convergence?

2. Gradient Descent on the Non-Convex Neural-Network Loss.

- Reuse the tiny one-hidden-layer neural network from the notebook and implement gradient descent (or basic SGD) on the full parameter vector.
- At each iteration, project the current parameter vector onto the 2D slice used in the contour plot (for example, by expressing the current values of the two perturbed parameters as (α, β) corresponding to $W_1[0, 0]$ and $W_2[0, 0]$).
- Overlay the resulting trajectory in the (α, β) -plane on top of the non-convex contour plot `figs/nonconvex_nn_loss_contours.png`.
- Compare the resulting paths for several different initializations of the full parameter vector. Do the trajectories converge to the same region or to different local minima?

3. Effect of Learning Rate and Noise.

- (a) Vary the learning rate in both the convex and non-convex experiments above. How do too-small and too-large learning rates manifest in the trajectories (e.g., very slow progress, overshooting, or divergence)?
- (b) Add small random perturbations (noise) to the gradient updates to mimic stochastic gradient descent. In the non-convex case, can you observe situations where noise helps the trajectory escape shallow minima or saddle-like regions?
- (c) Summarize your findings: contrast the behavior of gradient-based methods in the convex quadratic landscape versus the non-convex neural-network landscape. Relate your observations to the general discussion of convex vs. non-convex optimization in this section.

To summarize, the numerical experiments and visualizations in Example 3.2.1 and Exercise 3.2.1 illustrate a fundamental phenomenon: in convex landscapes, optimization behaves in a predictable and robust way, whereas in non-convex landscapes even simple models exhibit multiple valleys, plateaus, and dramatically different trajectories depending on initialization. These geometric differences explain why convex problems are mathematically tractable, while non-convex problems such as those arising in neural networks are considerably more challenging.

This motivates a deeper look at what convexity actually is and why it guarantees such favorable optimization behavior. We now formally introduce the mathematical notion of convexity and outline the key structural properties that make convex optimization the cornerstone of modern optimization theory.

3.2.2 Convexity: A Guiding Light in AI Optimization

Although AI optimization problems are generally non-convex, convexity is central to understanding how and why optimization works.

Convex Functions. A function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ is convex if

$$f(\alpha x + (1 - \alpha)y) \leq \alpha f(x) + (1 - \alpha)f(y), \quad \forall x, y, \alpha \in [0, 1]. \quad (3.6)$$

Convexity ensures that every local minimum is global.

Strict Convexity. If the inequality in Eq. (3.6) is strict for $x \neq y$, the minimizer is unique.

Second-Order Characterization. A twice differentiable function is convex iff its Hessian is positive semi-definite:

$$v^\top H_f(x)v \geq 0 \quad \forall v.$$

If the Hessian is positive definite, the function is strictly convex.

Convex Constraints. A feasible set \mathcal{C} is convex if

$$\alpha x + (1 - \alpha)y \in \mathcal{C}$$

for all $x, y \in \mathcal{C}$. Convex constraints preserve the favorable geometric structure of the problem.

Duality. Convex problems admit powerful dual formulations, relating the structure of constraints to the structure of the objective. Duality provides:

- computational advantages,
- sensitivity analysis,
- natural connections to Lagrangian methods.

First-Order Methods. Gradient descent and its variants (SGD, momentum, Adam, AdaGrad, RMSProp – all to be discussed below) form the backbone of modern ML optimization. In convex problems, gradient descent is guaranteed to converge to the global minimum; in strictly convex problems, convergence is linear.

3.2.3 Convexity of Logistic Regression

Logistic Regression (LR), introduced in Section 3.1, is a notable example of a convex optimization problem in machine learning, even when nonlinear feature maps are used.

Recall the objective:

$$\omega^* = \arg \min_{\omega} \sum_{i=1}^N \mathcal{L}_{\text{LR}}(y_i, \hat{y}(x_i, \omega)), \quad (3.7)$$

$$\hat{y}(x, \omega) = \frac{1}{1 + \exp(-\phi(x)^\top \omega)}, \quad \mathcal{L}_{\text{LR}}(y, \hat{y}) = -\log(\hat{y}^y (1 - \hat{y})^{1-y}). \quad (3.8)$$

Convexity follows from:

1. The logistic loss is convex in \hat{y} .
2. The mapping $\omega \mapsto \phi(x)^\top \omega$ is affine.
3. A nonnegative weighted sum of convex functions is convex.

From Linear Programming to Primal and Dual. It is instructive to transform logistic regression into a Linear Programming (LP) approximation. Introducing slack variables t_i enforcing

$$t_i \geq \mathcal{L}_{\text{LR}}(y_i, \hat{y}(x_i, \omega)),$$

and approximating the convex log-loss via a piecewise linear lower envelope yields the LP relaxation:

$$\min_{\omega, t} \sum_{i=1}^N t_i, \quad \text{s.t.} \quad t_i \geq a_k(y_i) \phi(x_i)^\top \omega + b_k(y_i), \quad \forall i, k.$$

Such LP formulations are effective for interpretability and robustness analysis, though not competitive with gradient-based methods for modern high-dimensional models.

3.2.4 Constrained Optimization and Lagrange Multipliers

Many AI problems involve constraints (fairness, robustness, regularization). The Lagrangian method provides a systematic approach to such problems:

$$\mathcal{L}(x, \lambda) = f(x) + \lambda^\top g(x),$$

where $g(x) \leq 0$ encodes constraints. Optimality is characterized by the Karush–Kuhn–Tucker (KKT) conditions.

Support Vector Machines (SVM) exemplify constrained convex optimization, forming an ideal entry point for primal–dual optimization. The soft margin SVM solves:

$$\min_{w,b,\xi \geq 0} \frac{1}{2} \|w\|^2 + C \sum_{i=1}^N \xi_i, \quad \text{s.t. } y_i(w^\top x_i + b) \geq 1 - \xi_i.$$

SVMs are closely related to logistic regression but optimize *hinge loss* instead of log-loss.

Example 3.2.2 (Primal–dual derivation of a two-point SVM). *For a toy dataset*

$$\{([1, 1], +1), ([−1, −1], −1)\},$$

one obtains the dual optimization problem:

$$\max_{0 \leq \lambda_i \leq C} \sum_{i=1}^2 \lambda_i - \frac{1}{2} \sum_{i,j} \lambda_i \lambda_j y_i y_j (x_i^\top x_j), \quad \sum_i \lambda_i y_i = 0.$$

Solving this by hand yields the optimal separating hyperplane.

Exercise 3.2.2. *Derive the dual formulation above using the Lagrangian and KKT conditions. Then:*

1. *Explicitly solve for the optimal multipliers λ .*
2. *Recover w and b .*
3. *Create a Jupyter notebook that visualizes the decision boundary for a larger synthetic 2D dataset.*
4. *Study the effect of varying C on the margin.*

From Optimization Landscapes to Optimization Algorithms

The preceding section emphasized the *geometry* of optimization problems: convexity, curvature, constraints, and the qualitative structure of loss landscapes. Before turning to specific algorithms, it is helpful to understand how geometry informs the design of practical methods.

- In a strongly convex landscape, plain gradient descent already enjoys global con-

vergence with a predictable rate, and algorithmic complexity is well understood.

- In non-convex problems—such as neural networks or transformers—the landscape contains flat regions, ridges, anisotropy, and poorly conditioned directions. Here, curvature varies dramatically across coordinates and layers.
- This mismatch between *local geometry* and a *uniform step size* motivates the introduction of momentum, coordinate-wise scaling, normalization layers, and adaptive learning rates.

Thus, gradient-based algorithms should not be viewed as abstract update rules but as tools designed to cope with the geometric distortions arising in modern AI models. The methods introduced next – GD, SGD, momentum, RMSProp, Adam, and their variants – can be interpreted as increasingly sophisticated attempts to adapt the update step to the underlying curvature.

3.3 Gradient Descent and Its Essential AI Variants

Gradient descent is the backbone of optimization in AI, enabling efficient model training by minimizing loss functions. However, AI optimization problems present unique challenges, including high-dimensionality, non-convex landscapes, and large datasets. To address these challenges, several gradient descent variants have been developed, each offering advantages in different settings.

This section introduces key first-order optimization methods, discussing their theoretical foundations in convex settings before transitioning to their practical application in AI, particularly in Deep Learning (DL).

3.3.1 Gradient Descent (GD)

Gradient Descent (GD) iteratively updates the vector of parameters θ_t in the discrete time, $t = 0, \dots$ based on the negative gradient of the loss function over θ_t :

$$\theta_{t+1} = \theta_t - \eta \nabla_{\theta_t} \sum_{i=1}^N \mathcal{L}(x_i, \theta_t), \quad (3.9)$$

where $(x_i | i = 1, \dots, N)$ is the Ground Truth (GT) data set with N samples. (Recall over-damped relaxation in a potential we have discussed in Chapter 2.) For a convex and L -smooth function, GD converges at a rate of $O(1/T)$ for general convex functions and $O(1/T^2)$ for strongly convex functions with an optimal step size η . Here the number of iterations T required to reach an error ϵ depends on the problem dimension n (in the context of AI training – on the number of parameters), with typical dependencies being $T = O(n \log(1/\epsilon))$ for well-conditioned convex problems. This dependence worsens with high-dimensional ill-conditioned problems.

GD was first introduced by Augustin-Louis Cauchy (1847) and later formalized for convex optimization by Yurii Nesterov (1983) [2, 3].

3.3.2 Stochastic Gradient Descent (SGD)

SGD approximates GD by computing noisy gradient estimates over randomly chosen batches:

$$\theta_{t+1} = \theta_t - \eta \nabla_{\theta_t} \sum_{i \in \text{batch}} \mathcal{L}(x_i, \theta_t),$$

where batch is updated frequently (possibly at each iteration step) at random, that is stochastically – thus the name. SGD reduces per-iteration computational cost and enables large-scale learning but converges at a slower rate of $O(1/\sqrt{T})$ in expectation. A key consideration in SGD is batch size. Worst case analysis suggests that an optimal batch size, balancing computational efficiency and variance reduction, scales as $O(N)$. However, in practice the batch size is much smaller than the number of GT samples, therefore resulting in gain in efficiency. The method was pioneered by Robbins and Monro (1951) and refined for ML by Léon Bottou (1998) [4].

3.3.3 Momentum-Based Methods

Momentum methods incorporate past increments to accelerate convergence by simulating the behavior of a dynamical system with inertia. The classical Polyak Heavy-Ball Method (Polyak, 1964) follows the update rule:

$$\theta_{t+1} = \theta_t + \beta(\theta_t - \theta_{t-1}) - \eta \nabla_{\theta_t} \sum_i \mathcal{L}(x_i, \theta_t), \quad (3.10)$$

where β is the momentum coefficient, controlling how much of the past increment is retained; η is the step size; $\mathcal{L}(x_i, \theta)$ is the loss function evaluated at data point x_i . The heavy-ball method can be interpreted as a damped second-order system in continuous time, where momentum reduces oscillations and accelerates convergence.

Nesterov (1983) introduced an accelerated momentum-based method:

$$\theta_{t+1} = \theta_t + \beta(\theta_t - \theta_{t-1}) - \eta \nabla_{\theta_t} \sum_i \mathcal{L}(x_i, \theta_t + \beta(\theta_t - \theta_{t-1})), \quad (3.11)$$

where the key difference is that the gradient is evaluated at an anticipated point $\theta_t + \beta(\theta_t - \theta_{t-1})$, rather than at θ_t .

For a smooth and strongly convex objective the heavy-ball Polyak's method achieves $O(1/T)$ convergence in T iterations under well-tuned parameters, which is however not as fast typically as the optimal $O(1/T^2)$ rate achieved by Nesterov's acceleration.

The main distinction between heavy-ball and standard gradient descent lies in their behavior for ill-conditioned problems: the heavy-ball method can exhibit faster practical convergence despite its theoretical rate.

Exercise 3.3.1. Mechanical Interpretation of Momentum Methods: Consider the equations of motion for a damped oscillator discussed in the previous chapter. Provide a mechanical interpretation for the Polyak heavy-ball and the Nesterov Accelerated Method. Further, consider the following extensions:

- *Pen-and-paper derivation:* Derive the continuous-time ODE limit of Polyak’s and Nesterov’s methods and discuss their implications for optimization.
- *Phase space analysis:* Sketch the phase portraits (velocity vs. position) for both methods and explain the impact of momentum on the trajectory.
- *Parameter tuning:* Given a convex quadratic function, analyze how choosing different values of β affects convergence and stability.

3.3.4 Projected Gradient Descent (PGD)

In constrained optimization, a key challenge is ensuring that iterates remain within a feasible set \mathcal{C} . One widely used approach is Projected Gradient Descent (PGD), which enforces constraints by projecting the updated iterate back onto the feasible set after each gradient step. The PGD update rule is:

$$\theta_{t+1} = \text{Proj}_{\mathcal{C}}(\theta_t - \eta \nabla_{\theta_t} \sum_i \mathcal{L}(x_i, \theta_t)).$$

This ensures that each iterate θ_t remains within \mathcal{C} while descending along the gradient of the objective function. The projection operator $\text{Proj}_{\mathcal{C}}(\cdot)$ finds the closest point in \mathcal{C} to the updated parameter, maintaining feasibility.

PGD is guaranteed to converge under mild assumptions. When the loss function is convex and Lipschitz-smooth, and the constraint set \mathcal{C} is convex, PGD converges at a rate of $O(1/T)$ for general convex functions and $O(\exp(-\mu T))$ for strongly convex functions with $\mu > 0$. These guarantees ensure that PGD remains an effective tool for constrained optimization. PGD has its roots in classical projection methods for constrained convex optimization, with early developments appearing in the work of *Bertsekas (1976)* and later refined in studies such as *Nesterov (2004)* and *Beck and Teboulle (2009)*. PGD has since been widely used in applications ranging from machine learning to signal processing and robust optimization.

3.3.5 Adaptive Learning Rate Methods

Optimization methods with *adaptive learning rates* dynamically adjust step sizes based on past gradient information. Such approaches have become central in modern deep learning, where high dimensionality, heterogeneous curvature, and sparsity patterns necessitate learning-rate schedules more flexible than the fixed-step schemes used in classical gradient descent.

Below we review three foundational adaptive methods – **AdaGrad**, **RMSProp**, and **Adam** – highlighting their mathematical structure, historical origins, and practical implications.

AdaGrad (Adaptive Gradient Algorithm). AdaGrad [5] modifies standard gradient descent, Eq. (3.9), by assigning each parameter θ_α its own time-dependent learning rate:

$$\eta_\alpha^{(t)} = \frac{\eta}{\sqrt{G_\alpha^{(t)} + \epsilon}}, \quad G_\alpha^{(t)} = \sum_{t'=1}^t \left(\partial_{\theta_\alpha^{(t')}} \sum_i L(x_i, \theta^{(t')}) \right)^2.$$

Since $G_\alpha^{(t)}$ accumulates the squared gradients over time, directions with frequent large gradients receive progressively smaller step sizes. AdaGrad is particularly effective for sparse or highly anisotropic problems, but its monotonically growing accumulator can cause learning rates to shrink excessively, slowing convergence at later stages.

RMSProp (Root Mean Square Propagation). RMSProp, introduced by Tieleman and Hinton in a lecture note [6], mitigates AdaGrad's diminishing-step-size problem by replacing the cumulative sum with an exponentially decaying moving average:

$$\begin{aligned} G_\alpha^{(t+1)} &= \beta G_\alpha^{(t)} + (1 - \beta) \left(\partial_{\theta_\alpha^{(t)}} \sum_i L(x_i, \theta^{(t)}) \right)^2, \\ \theta_\alpha^{(t+1)} &= \theta_\alpha^{(t)} - \eta \frac{\partial_{\theta_\alpha^{(t)}} \sum_i L(x_i, \theta^{(t)})}{\sqrt{G_\alpha^{(t+1)} + \epsilon}}. \end{aligned}$$

The exponential decay prevents $G_\alpha^{(t)}$ from diverging, making RMSProp a stable and widely used optimizer for non-stationary or noisy gradient regimes, especially in deep learning.

Adam (Adaptive Moment Estimation). Adam [7] combines the ideas of AdaGrad and RMSProp with momentum by tracking both first and second moments of the gradient:

$$\begin{aligned} m_\alpha^{(t+1)} &= \beta_1 m_\alpha^{(t)} + (1 - \beta_1) \partial_{\theta_\alpha^{(t)}} \sum_i L(x_i, \theta^{(t)}), \\ G_\alpha^{(t+1)} &= \beta_2 G_\alpha^{(t)} + (1 - \beta_2) \left(\partial_{\theta_\alpha^{(t)}} \sum_i L(x_i, \theta^{(t)}) \right)^2, \\ \hat{m}_\alpha^{(t+1)} &= \frac{m_\alpha^{(t+1)}}{1 - \beta_1^t}, \quad \hat{G}_\alpha^{(t+1)} = \frac{G_\alpha^{(t+1)}}{1 - \beta_2^t}, \\ \theta_\alpha^{(t+1)} &= \theta_\alpha^{(t)} - \eta \frac{\hat{m}_\alpha^{(t+1)}}{\sqrt{\hat{G}_\alpha^{(t+1)} + \epsilon}}. \end{aligned}$$

Adam has become the de-facto standard optimizer in deep learning due to its fast empirical convergence, robustness to hyperparameter choices, and efficiency in stochastic settings. Despite subtle issues with theoretical convergence (now well understood), its practical performance has made it ubiquitous across modern neural network applications.

Example 3.3.1 (Adaptive Optimizers on a Non-Convex ReLU Landscape). *To illustrate the qualitative differences between AdaGrad, RMSProp, and Adam on a genuinely non-convex optimization problem, consider the Jupyter/Python notebook `adaptive_relu_2D.ipynb`. The dataset consists of two classes in \mathbb{R}^2 :*

- points inside a disk of radius R (label 1),
- points outside the disk (label 0).

This makes the optimization landscape of the classification problem strongly non-convex even in low dimension. A tiny neural network

$$x \mapsto f_\theta(x) = W_2 \sigma(W_1 x + b_1) + b_2, \quad \sigma(z) = \max(z, 0),$$

is trained using binary cross-entropy loss. Because the ReLU network with a small hidden layer cannot represent a true circular boundary, the model learns a piecewise linear polygonal approximation to the circle.

Optimizer Comparison at Hidden Dimension $H = 2$. Figs. 3.5–3.6 show the performance of the three optimizers when the hidden layer width is $H = 2$.

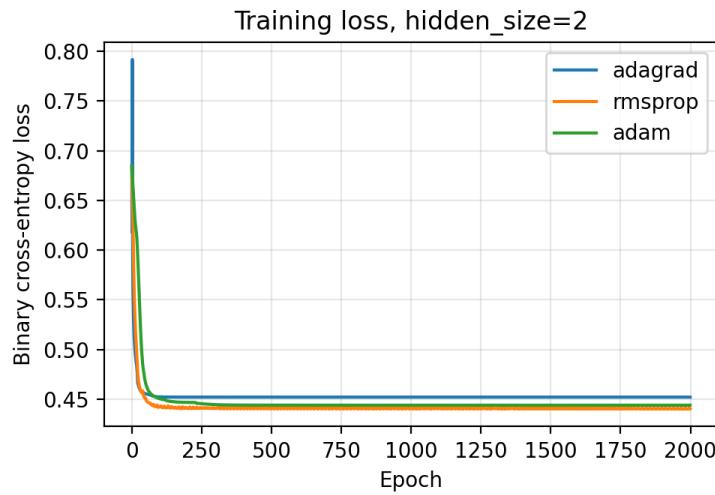


Figure 3.5: Training loss for AdaGrad, RMSProp, and Adam on the non-convex ReLU ring-classification task with $H = 2$. AdaGrad descends very quickly at first but plateaus higher. Adam exhibits (presumably due to oscillations) a slower progress. RMSProp shows the most stable and lowest long-run loss.

Observations.

- **AdaGrad:** Converges quickly initially due to large effective learning rates on under-utilized parameters, but its learning rate decays rapidly and becomes too conservative. As a result, optimization slows dramatically and the model underfits.
- **Adam:** Exhibits oscillatory behavior and a somewhat noisy trajectory in this small-scale setting; this is consistent with known issues where Adam may overshoot due to accumulated first moments when curvature is highly anisotropic.
- **RMSProp:** Performs the best. It balances adaptivity and stability, avoids AdaGrad's vanishing steps, and avoids Adam's oscillatory behavior.

Width Sweep: $H = 2, 4, 8, 32$ with RMSProp.

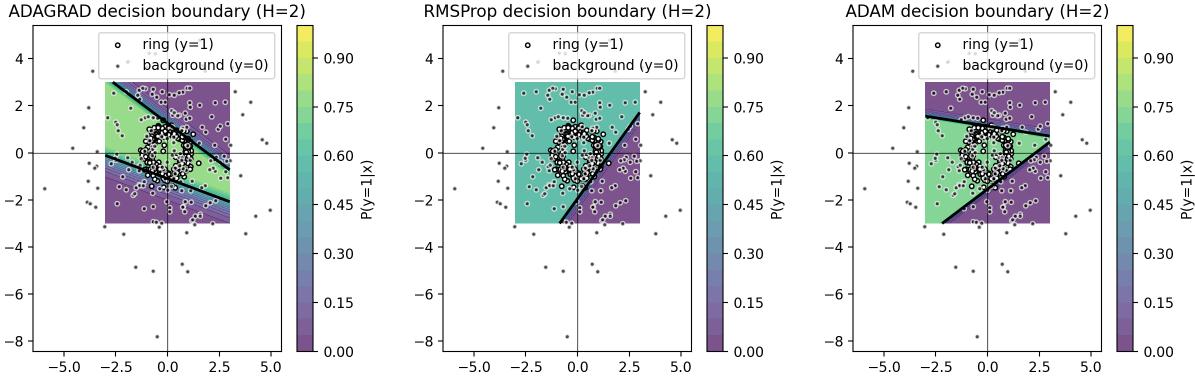


Figure 3.6: Decision boundaries (white curves) for AdaGrad (left), RMSProp (center), and Adam (right). All boundaries are triangular, reflecting that $H = 2$ ReLU units can produce only three linear segments. This demonstrates the expressivity bottleneck at small width: optimizer choice cannot overcome representational limits of the network. RMSProp produces the most symmetric and stable triangle.

To understand the geometry induced by width, the notebook increases the number of hidden ReLU units. A network with H ReLUs can represent decision boundaries as a polygon with at most H linear segments.

Figures 3.7–3.8 visualize this transition from triangle → polygon → nearly circular boundary.

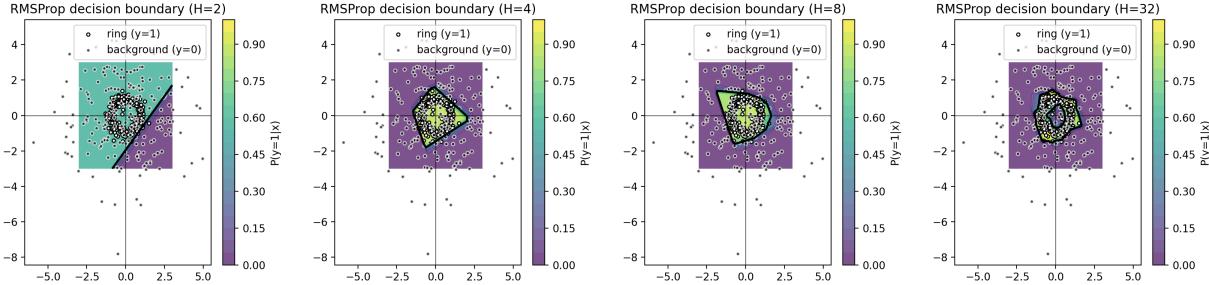


Figure 3.7: Decision boundaries for RMSProp as hidden width increases. **Left to right:** $H = 2$ (triangle), $H = 4$ (quadrilateral), $H = 8$ (octagon-like polygon), $H = 32$ (nearly circular). Increased width increases the number of linear regions, approximating the circle with growing fidelity.

Key Takeaways.

- *Width limits representational capacity: small H forces polygonal boundaries; large H approximates smooth shapes.*
- *Optimizers behave differently in low-dimensional non-convex problems: RMSProp is stable, AdaGrad collapses to tiny steps, Adam oscillates.*
- *The notebook provides a clean sandbox for observing expressivity vs. optimization effects in a controlled low-dimensional model.*

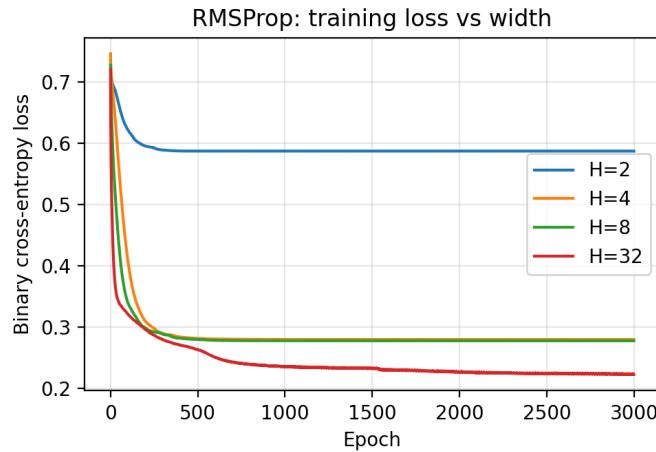


Figure 3.8: Training loss for RMSProp as a function of iteration for different hidden widths. Larger width models achieve lower final loss and more stable convergence.

Exercise 3.3.2 (Optimizer Behavior in a Non-Convex ReLU Model). *Using the notebook `adaptive_relu_2D.ipynb`, extend the analysis from Example 3.3.1 through the following tasks.*

1. **Optimizer Sensitivity Study.** For the three optimizers (AdaGrad, RMSProp, Adam):

- (a) explore learning rates $\eta \in \{10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}\}$,
- (b) run multiple random initializations,
- (c) plot mean and variance of the loss curves.

Discuss which optimizer is most robust to hyperparameters and initialization.

2. **Expressivity and Geometry.** For widths $H \in \{2, 4, 8, 16, 32, 64\}$:

- (a) train with RMSProp,
- (b) extract the decision boundary,
- (c) compute a measure of “circularity” (e.g., variance of boundary radius).

Verify empirically that the boundary converges from triangle \rightarrow polygon \rightarrow smooth circle.

3. **Optimizer–Expressivity Interaction.** Pick $H = 8$ and compare how AdaGrad, RMSProp, and Adam differ in:

- boundary smoothness,
- convergence rate,
- stability across random seeds.

Explain why optimizers interact differently with the same network architecture.

4. ***Challenging Regimes.*** Increase noise by adding label flips or Gaussian perturbations to inputs. Which optimizer is most robust? Which boundary shape distorts the least?

Include all plots and provide a short written report summarizing insights about:

- optimizer dynamics,
- expressivity-limited non-convexity,
- geometry of learned decision boundaries.

The Indispensability of First-Order Methods

The algorithms we have discussed so far in this section are fundamentally *first-order methods* because their search directions rely primarily on the computation of the first derivative (the gradient) of the optimization objective. While simple, these techniques are the indispensable workhorses that allow us to navigate the complex, high-dimensional loss landscapes of Generative AI.

However, their pervasive use raises a critical question: If second-order methods offer theoretically superior convergence speed by incorporating curvature information, why do we rely solely on the simple gradient? The answer lies in the computational limitations and scale of the Hessian matrix, which we address in the following subsection.

3.3.6 Why the Second-Order Methods are no go in AI?

The pervasive use of first-order methods like Gradient Descent and Adam in large-scale AI is not a matter of choice, but necessity. While Automatic Differentiation (AD) handles first-order derivatives (gradients) with remarkable efficiency, extending the same logic to second-order derivatives in high-dimensional spaces introduces a fundamental, insurmountable bottleneck. This bottleneck explains why virtually all state-of-the-art Generative AI models are trained using first-order methods and ignore the theoretically superior second-order methods (like Newton's method).

The second-order derivative information is captured by the *Hessian matrix*, H . For an optimization objective $L(W)$ (like a loss function in neural network training) depending on a parameter vector W with P elements, the Hessian is a matrix of all second partial derivatives:

$$H_{ij} = \frac{\partial^2 L}{\partial W_i \partial W_j}, \quad \text{where } H \in \mathbb{R}^{P \times P}.$$

The Hessian is crucial because it provides local curvature information about the loss landscape. This curvature information is what allows second-order optimization methods (like Newton's method) to converge quadratically (extremely fast) near a minimum by adapting the search direction to the shape of the function.

However, the cost of utilizing the Hessian becomes prohibitive for modern AI uses. The core issue is scaling:

- **Storage Complexity:** For a generative model with P parameters, the Hessian matrix H has P^2 elements. Since P can reach 10^{11} (billions), storing the P^2 values requires an infeasible amount of memory (scaling as $\mathcal{O}(P^2)$).

- **Inversion Complexity:** The full Newton step requires computing the inverse of the Hessian, H^{-1} , which scales as $\mathcal{O}(P^3)$ and is computationally impossible for realistic values of P .

While methods exist to compute the Hessian-Vector Product (HVP) efficiently ($\mathcal{O}(P)$) without explicitly forming H , the full Hessian required for global optimization remains the *Second-Order Cost Trap*. This hard limit forces us to rely on the simpler, less informed gradient direction, which necessitates more sophisticated techniques (momentum, adaptive learning rates) to navigate the complex, high-dimensional loss landscapes of modern AI.

Exercise 3.3.3 (Computational Cost and Properties of the Hessian). *This exercise explores the computational limitations and optimization insights associated with the Hessian matrix, $H = \nabla^2 L(W)$.*

1. **Complexity Analysis.** If a typical large language model has $P \approx 10^{11}$ parameters (e.g., 100 billion):
 - (a) Calculate the memory (in bytes, assuming 4-byte floats) required just to store the Hessian matrix $H \in \mathbb{R}^{P \times P}$. Why does this immediately rule out its use?
 - (b) What is the computational complexity (in terms of P) required to invert H (i.e., to implement the full Newton update)? State the complexity using Big O notation.
2. **Optimization Insight.** In low dimensions, Newton's method (which uses H^{-1}) finds optimal solutions faster than Gradient Descent. Why? Specifically, what information about the loss landscape, provided by the Hessian, allows Newton's method to choose a more effective direction and step size than the simple negative gradient?
3. **The Problem of Saddle Points.**
 - (a) The critical points of the loss function $L(W)$ are defined by $\nabla L(W) = 0$. Describe the necessary and sufficient conditions on the eigenvalues of the Hessian, $\lambda_i(H)$, that define a saddle point.
 - (b) Why do saddle points pose a significantly greater obstacle to training large neural networks than local minima, particularly in high-dimensional space? (Hint: Think about the number of dimensions/directions that lead away from the critical point.)

3.4 Regularization & Sparsity

In the previous sections of the chapter, we analyzed optimization methods without assuming much about the structure of the underlying optimization problem. The worst-case analysis often dominates theoretical discussions, leaving significant room for practical improvements when special structural properties exist in the problem. One such property is sparsity, which arises naturally in various AI applications and can be exploited to improve both training and inference efficiency.

This section introduces sparsity as an essential concept in AI optimization, covering:

- The motivation for sparsity: Why it arises and how it can be leveraged.
- Compressed sensing: A theoretical (and also practical) framework for exploiting sparsity.
- The link between ℓ_0 and ℓ_1 regularization, convex relaxation, and Linear Programming (LP).
- The role of sparsity in AI models, including training DNNs and accelerating inference.
- Practical implementations, including a computational exercise on sparsity-based regularization in NNs.

3.4.1 Compressed Sensing and Sparse Optimization

Compressed sensing (CS) is a paradigm that exploits the fact that many high-dimensional signals and datasets have an underlying sparse structure. The fundamental insight is that *under-determined linear systems, which appear to be ill-posed, can still be solved uniquely when the true solution is sparse*.

Historical Context and AI Relevance: The theory of compressed sensing was pioneered by Candes, Romberg, and Tao (2006) and Donoho (2006), who showed that a sparse signal can be recovered from far fewer observations than traditionally required by the Nyquist-Shannon sampling theorem. The connection to AI arises in several contexts:

- In **deep learning**, sparsity is used in weight pruning and feature selection.
- In **optimization**, sparsity reduces the effective problem dimensionality, improving efficiency.
- In **inference acceleration**, sparse representations allow for faster computations at runtime.

Mathematical Formulation: Consider an underdetermined system:

$$Ax = b, \quad A \in \mathbb{R}^{m \times n}, \quad m \ll n. \quad (3.12)$$

The system has infinitely many solutions unless additional constraints are imposed. If we know that x is sparse, meaning that only a few of its components are nonzero, we can attempt to recover it by solving:

$$\min_x \|x\|_0 \quad \text{s.t.} \quad Ax = b, \quad (3.13)$$

where $\|x\|_0$ is the shortcut for the l_0 "norm" (which is not actually a norm, as it does not satisfy the "homogeneity" property $\|ax\| = |a|\|x\|$ required for norms) of the vector x , that is

$$\|x\|_0 = \sum_{i=1,\dots,n} \begin{cases} 1, & x_i = 0 \\ 0, & \text{otherwise.} \end{cases}$$

However, this ℓ_0 minimization problem is combinatorial and computationally intractable (that is number of steps required is exponential in n). A major breakthrough in compressed sensing is that relaxing ℓ_0 to ℓ_1 norm, which consists of summing up the absolute values of each component of x , leads to a convex optimization problem, which can be solved efficiently:

$$\min_x \sum_{i=1}^n |x_i| \quad \text{s.t.} \quad Ax = b. \quad (3.14)$$

This reformulation can be efficiently solved using **Linear Programming** (LP), linking sparsity to convex optimization techniques discussed earlier.

Why ℓ_1 Regularization Works?

A fundamental insight behind the success of ℓ_1 minimization is that the ℓ_1 norm promotes sparsity while remaining convex. Intuitively:

- Unlike the ℓ_2 norm, which penalizes large deviations evenly, the ℓ_1 norm favors solutions where many elements are exactly zero.
- Under suitable conditions (such as the Restricted Isometry Property, RIP), the ℓ_1 minimization recovers the same sparse solution as the ℓ_0 problem with high probability. (And this is where the asymptotic, $n \rightarrow \infty$, theory by Candes, Romberg, and Tao (2006) [8] plays a crucial role.)

A Geometric Perspective: The difference between ℓ_1 and ℓ_2 regularization can be understood through the geometry of their constraint regions and how they interact with the level sets of the loss function.

Let us illustrate this with a simple two-dimensional constrained optimization problem:

$$\begin{aligned} \ell_2 : \quad & \arg \min_x (x_1^2 + x_2^2) \quad \text{s.t.} \quad x_1 + 2x_2 = 1 \\ \ell_1 : \quad & \arg \min_x (|x_1| + |x_2|) \quad \text{s.t.} \quad x_1 + 2x_2 = 1. \end{aligned}$$

This setup captures a fundamental contrast: when minimizing the ℓ_2 norm, the optimal solution has small but nonzero values for both x_1 and x_2 , while minimizing the ℓ_1 norm promotes sparsity by forcing one component to be exactly zero.

As shown in Fig. (3.9):

- The **red line** represents the constraint $x_1 + 2x_2 = 1$, restricting the feasible solutions.
- The **blue dashed circles** represent level sets of the ℓ_2 loss function. The optimal solution occurs where the smallest blue circle touches the constraint, leading to small but nonzero values for both x_1 and x_2 .
- The **green dashed squares (rotated)** represent level sets of the ℓ_1 loss function. The optimal solution is at a corner, where either x_1 or x_2 is exactly zero. This illustrates why ℓ_1 regularization induces sparsity.

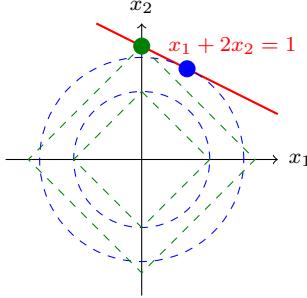


Figure 3.9: **Geometric interpretation of ℓ_1 vs. ℓ_2 regularization.** The red line represents the constraint $x_1 + 2x_2 = 1$, which restricts feasible solutions. The blue dashed circles are level sets of the ℓ_2 loss, with the **largest circle touching the constraint** at $(1/5, 2/5)$, leading to **small but nonzero values** for both x_1 and x_2 . The green dashed squares represent the level sets of the ℓ_1 loss, with the **largest square touching the constraint** at $(0, 1/2)$, enforcing a **sparse solution** where one coordinate is exactly zero. Smaller dashed contours are shown to illustrate suboptimal solutions. For a live demonstration of this geometric interpretation, please see

<https://www.desmos.com/calculator/1unxjbr3bm>.

3.4.2 Regularization and Its Importance in AI

Sparse Regularization in Machine Learning: Many modern AI models incorporate sparsity-inducing regularization. A common approach is **Lasso regression**, where the loss function includes an ℓ_1 penalty:

$$\min_w \sum_{i=1}^N \mathcal{L}(y_i, f_w(x_i)) + \lambda \|w\|_1. \quad (3.15)$$

This enforces sparsity in w , effectively selecting a subset of relevant features.

In **Neural Networks**, sparsity constraints are used to prune weights, reducing model size without significant loss of accuracy.

Example 3.4.1 (Image Denoising via Total Variation). *Total Variation (TV) regularization is one of the classical and most widely used approaches for image denoising. Given a noisy observation*

$$w = v + \xi,$$

where v is the clean image and ξ is noise, the TV restoration problem seeks

$$u^* = \arg \min_u \left[\frac{1}{2} \|u - w\|_2^2 + \lambda \sum_{(i,j) \sim (i',j')} \sqrt{\varepsilon^2 + (u_{ij} - u_{i'j'})^2} \right].$$

The first term enforces fidelity to the observed image, and the second term enforces spatial smoothness, penalizing local differences between neighboring pixels. This corresponds to the isotropic TV norm with a small smoothing parameter ε used to make the functional differentiable.

Gradient Flow Interpretation. The minimizer u^* can be found by gradient descent on the energy

$$E(u) = \frac{1}{2} \|u - w\|_2^2 + \lambda \sum_{(i,j) \sim (i',j')} \sqrt{\varepsilon^2 + (u_{ij} - u_{i'j'})^2}.$$

For each pixel,

$$\frac{du_{ij}}{dt} = -\frac{\partial E}{\partial u_{ij}} = w_{ij} - u_{ij} - \lambda \sum_{(i',j') \sim (i,j)} \frac{u_{ij} - u_{i'j'}}{\sqrt{\varepsilon^2 + (u_{ij} - u_{i'j'})^2}}.$$

This PDE is discretized and integrated numerically. A small step size is required for stability when λ is large, reflecting the stiff nature of the TV term, as discussed previously in Section 3.3.1.

Worked Example. We apply isotropic TV denoising to a 64×64 binary image of the letter “U”. Three levels of Gaussian noise with standard deviations $\sigma \in \{0.05, 0.15, 0.30\}$ are added, and TV denoising is performed for several values of the regularization parameter $\lambda \in \{0.0, 0.5, 1.5, 4.0\}$.

Figure 3.10 shows the resulting multi-panel comparison:

- top row: original and noisy images,
- subsequent rows: restored images for increasing λ .

As expected:

- small λ produces mild denoising while preserving edges,
- moderate λ removes noise effectively but begins to round corners,
- large λ oversmooths, yielding a blurry but very clean image.

Energy Decay During Optimization. The energy $E(u(t))$ decreases monotonically along the gradient flow. This behavior is illustrated in Figure 3.11, which shows the decay of the objective for several values of λ . Larger regularization produces a stiffer system and therefore a slower (despite stable) decay under explicit gradient descent.

The complete Python implementation generating both figures is available in the accompanying Jupyter notebook `ImageRestoration_TV.ipynb`, and students are encouraged to experiment with different values of λ , noise levels, and numerical solvers (explicit Euler, midpoint, implicit schemes).

3.4.3 Sparsity for Inference Acceleration

Motivation: Once a model is trained, inference should be fast. One approach to accelerating inference is to reduce the number of active parameters used in the forward pass.

Sparsity-Based Compression: A natural idea is to store only the most informative directions, identified during training. Specifically:

- Track gradients during training and identify key weight directions.
- Construct a low-dimensional subspace for inference.
- Project new inputs onto this space for efficient computation.

This reduces computational cost while maintaining accuracy.

Exercise 3.4.1. (*) Sparse Projection for Faster Inference²

In this exercise, we explore a dimensionality reduction strategy for trained Neural Networks (NN) to enhance inference efficiency:

- Train a NN on the MNIST dataset.
- Identify the most significant weight vectors utilized during training.
- Construct a compressed subspace based on these important weights and project test inputs onto this subspace.
- Evaluate the trade-off between speedup and accuracy, analyzing the impact of the compression.

This approach investigates how targeted sparsification can maintain performance while reducing computational overhead, contributing to efficient model deployment.

In conclusion (for this section): Sparsity is a powerful principle in AI optimization. Whether through compressed sensing, ℓ_1 -based regularization, or inference acceleration, sparsity enables efficient computation and improved generalization. By leveraging structured sparsity, we can develop AI models that are both accurate and computationally efficient.

3.5 (*) Optimization of Transformers – GenAI Example

³ In Section 1.1.5, we introduced the multi-head transformer model to illustrate key concepts of tensor operations, such as multiplications and convolutions, combined with nonlinear transformations. Here, we conclude the chapter on optimization by focusing on how transformers are trained – framing this as a non-convex optimization problem in state-of-the-art AI.

Training a transformer involves optimizing its weight matrices $\{W_Q, W_K, W_V\}$ in the attention layers, as well as the parameters in the feed-forward layers. We will use θ for an aggregated notation for all the adjustable parameters used to train a transormer. Given the large-scale nature of transformer models and their highly non-convex optimization landscape, sophisticated optimization techniques are essential to ensure convergence.

²The asterisk (*) indicates that this is a bonus exercise – challenging and optional, potentially requiring significant effort and creativity. Completing this exercise may even contribute to an original publication. The topic of inference acceleration is crucial, as it relies on innovation rather than massive computational resources. See [9, 10, 11, 12] and references therein.

³Sections marked with an asterisk (*) are optional and can be skipped on a first reading without loss of continuity.

3.5.1 Loss Function and Optimization Objective

The training objective for **transformers** typically involves minimizing the **cross-entropy** loss — \mathcal{L} , which we define later in the text — accumulated (sum over) all the input data. This optimization problem is non-convex due to:

- The hierarchical composition of multiple nonlinear transformations (e.g., self-attention, softmax, feedforward layers).
- The presence of highly non-linear residual connections and layer normalization.

Notably the high-dimensional parameter space makes global optimization intractable, thus motivating researchers to look for efficient heuristics, in particular based on Stochastic Gradient Based iterative algorithms.

3.5.2 Optimization Process: From Backpropagation to SGD Variants

Training transformers relies on **Back-Propagation** and **Stochastic Gradient**-based optimization methods. As discussed in Section 2.1, **Automatic Differentiation** (AD) plays a crucial role in computing gradients efficiently. There, we introduced both Forward and **Reverse Mode AD** and noted that Reverse Mode AD is preferable for minimizing the loss function in machine learning models. This preference arises because we need to compute the gradient of a scalar-valued loss function with respect to a high-dimensional vector of parameters (weight matrices), making Reverse Mode AD computationally efficient.

The gradient of the loss function with respect to each component of θ . Given the high-dimensional nature of transformers, simple gradient descent is inadequate due to slow convergence and instability in optimization. Instead, adaptive optimization techniques such as Adam are typically employed. By leveraging Reverse Mode AD and adaptive gradient-based methods, transformer models efficiently navigate the complex, non-convex loss landscape to achieve effective training.

3.5.3 How Does the Loss Function Handle Growing Input Sequences?

The training of transformers for sequence prediction involves handling dynamically growing input sequences. At each training step, the model processes progressively larger input segments while predicting the next token. The loss function — typically cross-entropy loss — is computed over multiple overlapping sub-sequences.

Consider a sequence of tokens:

$$[t_1, t_2, t_3, t_4, t_5].$$

During training, the model is presented with progressively increasing input sequences:

$$\begin{aligned} \text{Input: } [t_1] &\rightarrow \text{Target: } [t_2], \\ \text{Input: } [t_1, t_2] &\rightarrow \text{Target: } [t_3], \\ \text{Input: } [t_1, t_2, t_3] &\rightarrow \text{Target: } [t_4], \\ \text{Input: } [t_1, t_2, t_3, t_4] &\rightarrow \text{Target: } [t_5]. \end{aligned}$$

At each step, the model makes a prediction for the next token, and the loss function evaluates how well the predicted distribution aligns with the true target.

Summing Over Different Input Lengths: The total loss for a sequence of length T is computed as:

$$\mathcal{L}(\theta) = - \sum_{i=1}^T \log p_\theta(t_{i+1}|t_1, \dots, t_i), \quad (3.16)$$

where $p_\theta(t_{i+1}|t_1, \dots, t_i)$ is the predicted probability of the next token, and θ is a collective notation for all the parameters we adjust to train the transformer (see Section 1.1.5).

The model is trained to minimize this sum over all input subsequences with respect to θ , reinforcing its ability to predict tokens based on growing contexts. This dynamic approach allows transformers to efficiently learn dependencies in sequences and generalize effectively to unseen data.

Connector: From Optimization Theory to Transformers

The adaptive optimizers introduced above (momentum methods, RMSProp, Adam, and their variants) were not designed with transformers in mind – yet they became the backbone of modern large-scale transformer training. Why?

Transformers present a highly *anisotropic*, *layer-coupled*, and *poorly conditioned* optimization landscape. Gradients vary in scale across attention heads, feed-forward blocks, embeddings, and normalization layers. As a consequence:

- coordinate-wise scaling (as in RMSProp/Adam) helps navigate directions of extremely uneven curvature;
- momentum smooths out high-variance gradients that arise even in full-batch regimes due to architectural depth;
- the geometry of attention introduces directions in which naive gradient descent makes excessively large steps.

The tiny transformer experiment that follows illustrates these effects on a much smaller scale: although the model is toy-sized, the same optimization pathologies and sensitivities already appear. In this way, the example serves as a bridge from classical optimizer theory to the practical realities of training modern neural architectures.

Example 3.5.1 (Tiny Transformer Optimization). *In this example we make the abstract discussion of transformer optimization concrete by training a very small character-level transformer on the toy string*

"hello world. this is a simple transformer example."

using the companion Jupyter/PyTorch notebook `tiny-transformer.ipynb`.

We tokenize at the character level, build overlapping windows of fixed length $L = 5$, and train the transformer to predict the next character in each window. More precisely, if x_1, \dots, x_T

denotes the encoded character sequence, we form input–target pairs

$$(x_i, \dots, x_{i+L-1}) \mapsto (x_{i+1}, \dots, x_{i+L}), \quad i = 1, \dots, T - L.$$

The model is a minimal encoder–decoder transformer with an embedding layer, a few self–attention and feed–forward blocks, and a final linear layer projecting to logits over the vocabulary. We compare three optimizers:

- Stochastic Gradient Descent (SGD),
- Adam,
- RMSProp.

Fig. 3.12 shows the evolution of the mean per–token training loss for all three methods. A striking observation is that RMSProp does not perform well in this setting: the loss initially increases and then drifts back toward its starting value, showing little meaningful progress. Adam, on the other hand, converges quickly and achieves the lowest loss, while SGD improves steadily but slowly.

This behavior highlights a pedagogically important point: even on a tiny problem, transformer training is highly non–convex and sensitive to hyperparameters and initialization. For the particular model and learning rates used here, RMSProp can effectively “freeze” after a few large early gradients, producing little further learning. Adam’s two–moment normalization makes it more robust to such early steps.

Beyond the global loss, we can inspect how difficult each position in the length–5 window is to predict. Figure 3.13 shows the mean per–position loss at the final epoch. Some positions are systematically easier or harder, reflecting both the geometry of the data (e.g., spaces and punctuation) and the internal inductive biases of the transformer. The optimizers induce different profiles, again demonstrating that optimization and data structure are intertwined. Finally, Fig. 3.14 displays the confusion matrix of the best–performing optimizer (Adam). The strong diagonal indicates accurate predictions, while off–diagonal structure reveals consistent confusions (e.g., between space, period, and some letters). Even this tiny transformer has learned a meaningful next–character model, and the confusion structure provides a useful qualitative diagnostic.

The notebook `tiny-transformer.ipynb` contains the full implementation and produces all three figures. Students are encouraged to run and modify it.

Exercise 3.5.1 (Exploring Optimizers and Data Geometry in Tiny Transformers). *Using the companion notebook `tiny-transformer.ipynb` as a starting point, explore how optimization, model architecture, and data geometry interact in this tiny transformer.*

1. **Change the training text.** Replace the toy string “hello world. this is a simple transformer example.” with a different short text (natural language or a structured synthetic string).
 - Retrain with SGD, Adam, and RMSProp using the same hyperparameters and regenerate Fig. 3.12.

- Compare the new loss curves to the original ones. Does RMSProp behave differently on a different dataset? When does Adam retain its advantage?
2. **Vary the window length.** Modify `sequence_length` (e.g., $L = 3$ and $L = 8$) and repeat the experiment.
- Regenerate the per-position loss plot (analogous to Fig. 3.13). How does the difficulty profile change with L ?
 - Interpret how the receptive field and context length interact with optimizer behavior.
3. **Hyperparameter sensitivity of RMSProp.** RMSProp performed poorly in the example above.
- Reduce its learning rate (e.g., to `lr = 0.001` or `0.0005`) and repeat the training.
 - Compare the resulting curves to Adam and SGD. When (if ever) does RMSProp become competitive?
 - Discuss why RMSProp can “freeze” after large early gradients, and how this differs from Adam’s two-moment updates.
4. **Analyze confusion matrices.** For each modified setup, compute the confusion matrix of the best-performing optimizer.
- Compare how structure, spaces, and punctuation influence confusion.
 - Relate changes in confusion structure to both optimization behavior and the data distribution.

Summarize your findings, including regenerated versions of the three figures and a discussion of how small transformers expose the sensitivity of adaptive optimizers in non-convex settings.

Connector: Optimization as a Moving Target

The behavior observed in the tiny-transformer example highlights a fundamental truth about modern deep learning: *optimization is not a solved problem*. Performance depends delicately on architectural choices, initialization, normalization, learning-rate scales, and the interaction between model geometry and optimizer dynamics.

This sensitivity is amplified in large-scale transformers, where depth, attention mechanisms, long-range dependencies, and mixture-of-experts routing all interact to produce highly structured but challenging loss surfaces. As a result, optimizer design has become an active research frontier, tightly coupled to systems considerations (memory, throughput), modeling innovations (compression, structure), and new inference paradigms (multi-token prediction, speculative decoding).

The final subsection outlines several representative directions in which this frontier is evolving, illustrating how optimization, architecture, and hardware co-design now advance together.

3.5.4 Ongoing Advances in Transformer Optimization

Research on transformer optimization continues to evolve at a remarkable pace, driven by the need for faster training, lower memory footprint, and more efficient inference. Rather than a single dominant direction, progress now unfolds simultaneously across multiple fronts—including architectural innovations, algorithmic refinements, and new approaches to compression and sampling. The diversity of methods reflects a growing recognition that scalable transformer training is fundamentally a systems, architecture, and optimization problem at once.

A few representative examples (among many) illustrate these active trends:

- **DeepSeek-V3** [13]: Introduces low-rank join compression of attention keys and values, a refined mixture-of-experts design, and multi-token prediction to enable partial parallelization in generation. These ideas exemplify the broader movement toward *compression-aware training* and improved utilization of hardware bandwidth.
- **Titans** [14]: Proposes a long-short-term memory reinterpretation of attention, where attention handles short-term dependencies and recurrent structure preserves long-term information. This reflects a wider effort to bring *Recurrent Neural Network (RNN)-style recurrence and transformer-style attention* into closer alignment, thereby reducing memory costs while retaining expressivity.
- **Transformer²** [15]: Uses Singular-Value Decomposition (SVD) and Adaptive Importance Sampling (AIS) to accelerate optimization. This contributes to a growing class of *factorization- and sampling-based* approaches that seek to reduce training-time overhead by acting directly at the level of weight matrices or gradient estimation.

Many related themes are developing in parallel: structured sparsity, dynamic token routing, long-context approximations, improved optimizers tailored to attention geometry, mixture-of-experts scaling laws, and multi-token or speculative decoding strategies aimed at reducing inference latency. The field is highly dynamic, and no single idea has yet emerged as definitive. Instead, transformer optimization has become an active, rapidly shifting research frontier in which architectural changes, algorithmic insights, and systems-level constraints continually reshape one another.

TV-regularized image denoising: clean, noisy, and denoised images

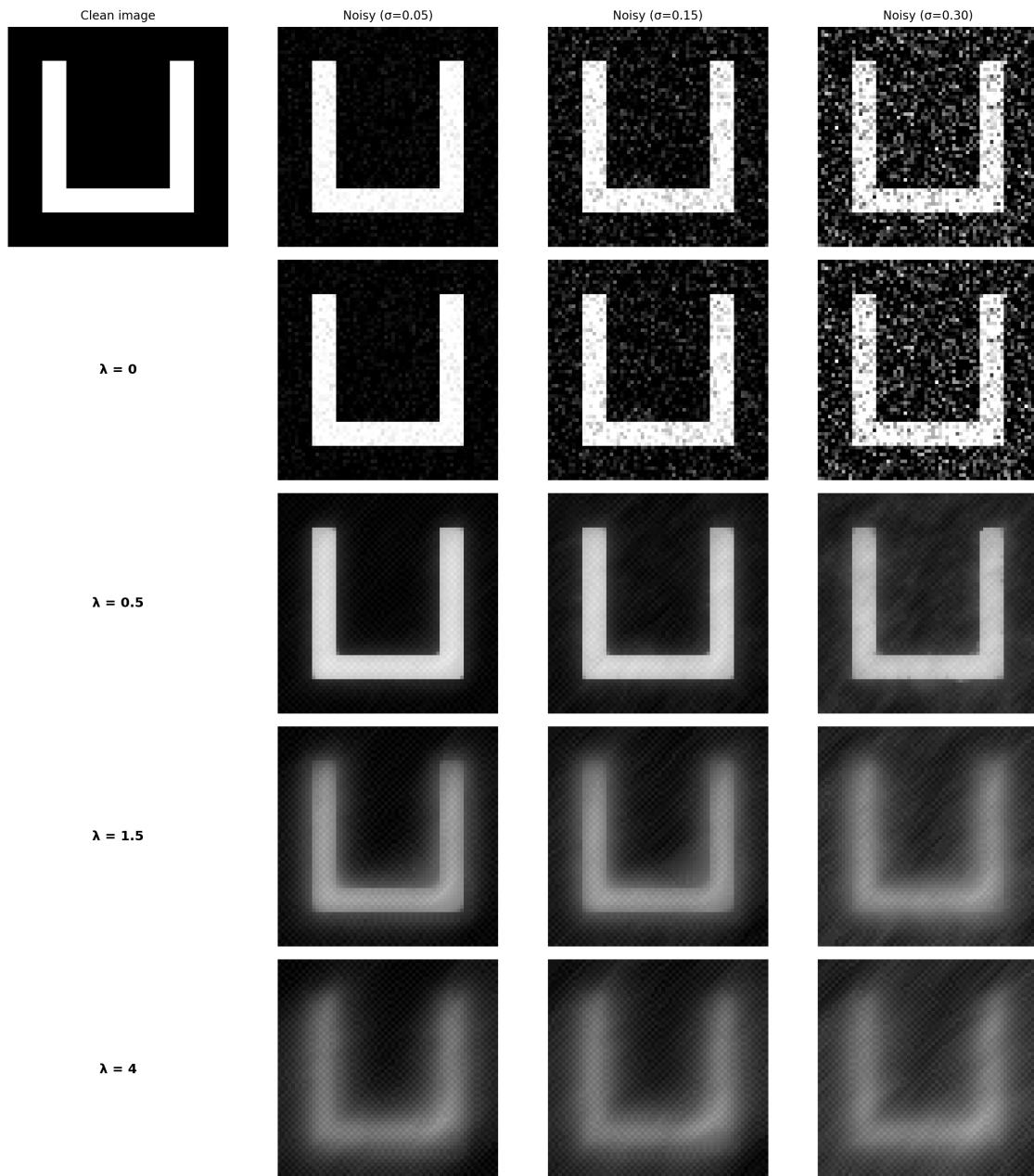


Figure 3.10: Total Variation denoising on a 64×64 image of the letter “U” for multiple noise levels (columns) and regularization parameters λ (rows). Larger values of λ produce stronger smoothing.

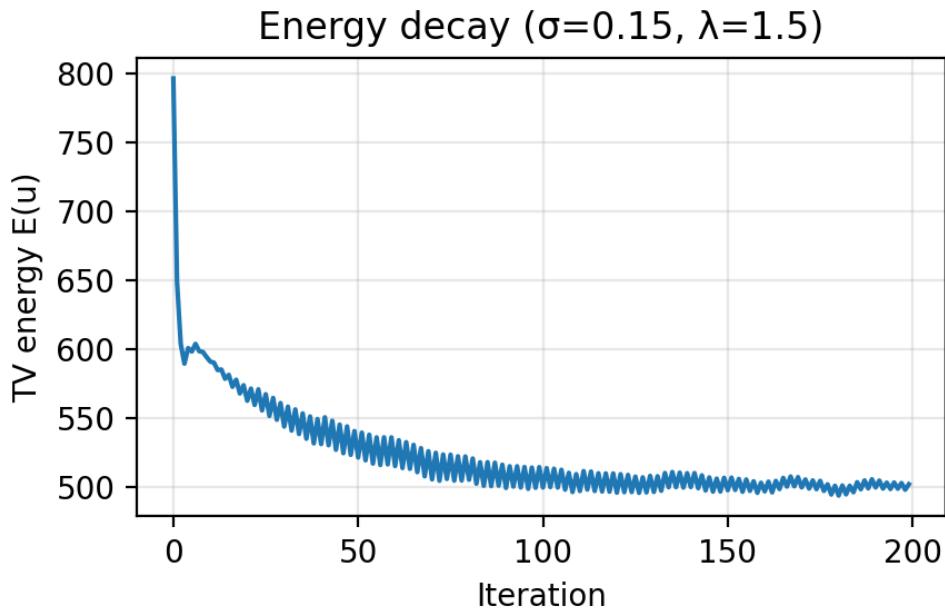


Figure 3.11: Energy $E(u(t))$ as a function of iteration during TV denoising. Each curve corresponds to a different λ . Larger λ results in a stiffer gradient flow and slower decay.

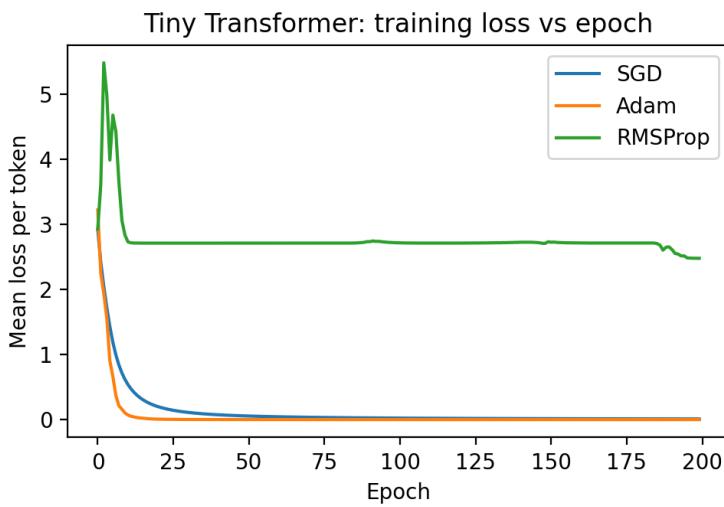


Figure 3.12: Tiny transformer: mean per-token training loss versus epoch for three optimizers (SGD, Adam, RMSProp) on the toy character-level sequence "hello world. this is a simple transformer example." Adam and RMSProp converge faster and reach a lower loss than vanilla SGD.

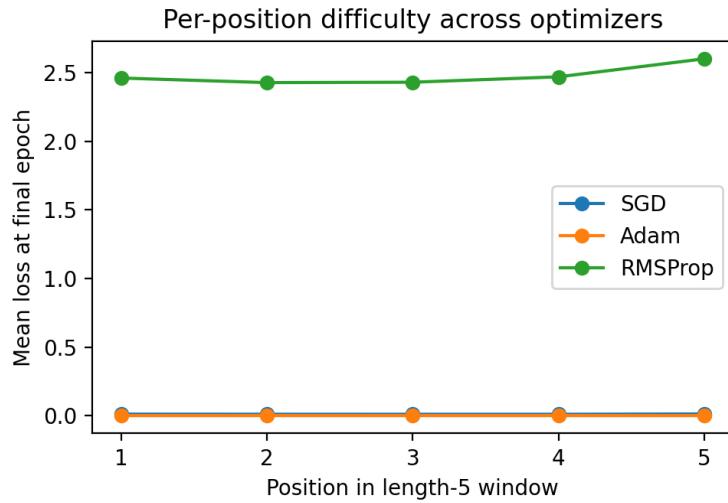


Figure 3.13: Mean per-position loss at the final epoch for each optimizer, as a function of position in the length-5 input window. Some positions are consistently easier or harder to predict, reflecting both data geometry and the optimization dynamics.

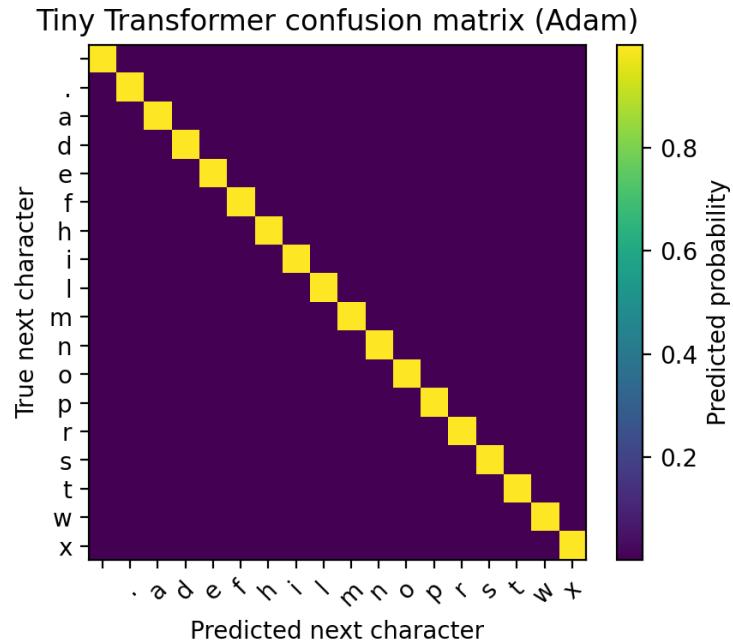


Figure 3.14: Confusion matrix for the best-performing optimizer (lowest final mean loss). Rows correspond to true next characters and columns to predicted characters; entries show the average predicted probability. The strong diagonal indicates accurate predictions, while off-diagonal structure reveals systematic confusions (e.g., between spaces, punctuation, and certain letters).

Chapter 4

Neural Networks & Deep Learning

We begin this chapter by situating neural networks within the mathematical framework developed in Chapters 1,2,3. The aim is to make explicit how the core ingredients of deep learning – representation of data, dynamical evolution through layers, and optimization of parameters – connect to earlier material:

1. **Data as vectors, matrices, and tensors (Chapter 1).** Neural networks process inputs that are represented as vectors or higher-order tensors. The linear components of each layer (affine maps, convolutions, embeddings) are direct applications of the matrix and tensor transformations introduced in Section 1.1.
2. **Layers as discrete-time dynamical systems (Chapter 2).** Each neural-network layer applies a transformation of the form

$$x_{k+1} = \sigma(W_k x_k + b_k),$$

and thus defines a discrete-time dynamical system. Residual and skip connections relate this directly to the ODE viewpoint of Section 2.2, where

$$x_{k+1} \approx x_k + h f_\theta(x_k)$$

resembles an explicit Euler step. This dynamical interpretation later serves as a bridge to Neural ODE models and, ultimately, to diffusion-based generative models.

3. **Training as optimization (Chapter 3).** Neural-network training is formulated as an empirical-risk minimization problem, where one seeks parameters θ minimizing a *loss function*:

$$\min_{\theta} \frac{1}{N} \sum_{i=1}^N \mathcal{L}(f_\theta(x_i), y_i).$$

This viewpoint extends the optimization framework of Sections 2.2.2–3.1 and incorporates *regularization* techniques such as sparsity (Section 3.4) that control model complexity and improve generalization.

4. **Gradient-based training algorithms.** The parameter updates used in deep learning—stochastic gradient descent, momentum, and adaptive methods—are direct descendants of the gradient-descent variants introduced in Section 3.3. Thus, the evolution of parameters during training can itself be viewed as a dynamical system in parameter space, influenced by noise, step size, and curvature of the loss landscape.
5. **Neural networks as nonlinear extensions of classical supervised-learning models.** Logistic regression (Section 3.1) is the simplest example of a neural network: a single affine layer followed by a nonlinearity. In contrast, deep networks compose many such transformations, enabling vastly richer decision boundaries and more complex feature hierarchies. The notebook associated with Exercise 3.1.2 illustrated how multilayer networks can represent intricate 2D boundaries and illuminated the geometry of the loss surface.

Purpose and scope of this chapter. Neural networks are among the most expressive and computationally effective function approximators available today. In this chapter, we introduce their mathematical structure and interpret them as layered compositions of linear transformations and nonlinear activations. Our emphasis is on supervised learning, with a focus on classification, where the goal is to learn a mapping $x \mapsto y$ from high-dimensional inputs to discrete class labels. Training proceeds by minimizing an appropriate loss function — typically the cross-entropy loss — over labeled data.

Along the way, we highlight architectural innovations (convolutions, residual connections, Neural ODEs) and structural principles (representation learning, energy landscapes) that set the stage for later chapters on probabilistic modeling, information theory, graphical models, and generative diffusion processes.

4.1 Neural Network Mechanics

In this section, we move from the abstract optimization and dynamical-systems viewpoints of Chapters 1,2,3 to the concrete mechanics of Neural Networks (NNs). Our focus is on how relatively simple and compact NNs can be used as flexible function approximators for supervised learning, and on how their parameters are trained by gradient-based methods. We proceed in three steps:

1. Start from the historical single-layer *perceptron*, which can be viewed as both a primitive model of a neuron and a special case of logistic regression.
2. Introduce a neural network with a single fully connected hidden layer and interpret it as an automatically learned nonlinear feature map.
3. Compare this NN to logistic regression with manually engineered nonlinear features on a two-dimensional classification task, using the accompanying Jupyter notebook.

4.1.1 Perceptron – Historical Remark

We begin with the simplest single-layer architecture (with no hidden layers) and one of the earliest artificial NN models — the **Perceptron**, introduced by Frank Rosenblatt in

1958 [16]. The name “Perceptron” reflects its **biologically inspired** design: Rosenblatt aimed to develop a mathematical model that mimicked, in highly simplified form, how neurons in the brain perceive and process information.

A perceptron takes an input vector $x \in \mathbb{R}^d$, computes a linear score $w^\top x + b$, and then applies a threshold nonlinearity to decide between two classes:

$$\hat{y} = \text{sign}(w^\top x + b).$$

Training consists of iteratively adjusting w and b based on misclassified examples, which can be interpreted as a primitive gradient-like update (compare with Chapter 3.3).

Despite its historical importance, the perceptron has a major limitation: it can only learn **linearly separable** problems. In their seminal work, Minsky and Papert (1969) [17] showed that a single-layer perceptron **cannot solve linearly non-separable problems**, such as the **XOR problem** (already discussed in the context of logistic regression with linear feature vectors in Section 3.1).

This limitation served as a turning point. It highlighted the need for more expressive architectures — in particular, **multi-layer** neural networks with hidden layers and nonlinear activations — and for efficient training algorithms, culminating in the **backpropagation** method. These developments opened the way to modern deep learning, where many layers are stacked to form highly expressive nonlinear maps.

4.1.2 NN with a Single Fully Connected Hidden Layer

We now revisit the two-dimensional classification problem introduced in Exercise 3.1.2. There, we examined logistic regression and saw that with a *linear* feature vector it cannot separate data sets of XOR type or other nonlinearly separable patterns. Introducing **nonlinear feature vectors** (e.g., polynomial features) can remedy this limitation, but at the price of explicit feature engineering.

In this subsection, we demonstrate how a neural network can address the same problem **at least as efficiently**, while learning its own nonlinear features. The simplest architecture that already exhibits this behavior is a **single-hidden-layer, fully connected neural network**.

Logistic Regression with Nonlinear Features vs. NN with a Hidden Layer

Before turning to a concrete example, it is useful to summarize the conceptual similarities and differences between these two approaches:

- **Similarity:** Both methods introduce nonlinearity to go beyond linear decision boundaries. Logistic regression achieves this via nonlinear feature maps $\phi(x)$, while an NN introduces nonlinearity through activation functions in the hidden layer.
- **Difference:** Logistic regression with nonlinear features requires **explicit feature engineering** (e.g., manually adding polynomial or radial basis transformations). By contrast, an NN with a hidden layer **learns** an internal feature transformation through its hidden neurons and parameters.

From the perspective of Chapters 1–3, both can be viewed as:

- compositions of linear maps (matrices) and nonlinearities (activation functions),
- trained by gradient-based optimization of a loss function (typically cross-entropy),
- but differing in whether the feature map is hand-designed or learned.

Example 4.1.1. *2D Classifier with a Fully Connected Single-Hidden-Layer NN.* In the Jupyter/PyTorch notebook `LogReg+NN-supervised-2D.ipynb`, we study a two-dimensional classification task in which the ground-truth labels are determined by membership in the union of two overlapping circles in \mathbb{R}^2 .

Data generation. Points $x = (x_1, x_2) \in \mathbb{R}^2$ are sampled uniformly in a square domain. Two circles of radius r are centered at $(-\text{offset}, 0)$ and $(+\text{offset}, 0)$. A point is labeled $y = 1$ if it lies inside at least one of the circles, and $y = 0$ otherwise.

Neural-network architecture. The implemented NN consists of:

- a **2D input layer** for (x_1, x_2) ,
- a **hidden layer with 10 neurons**, using the hyperbolic tangent activation function \tanh ,
- a **single output neuron** producing a scalar logit; applying the sigmoid $\sigma(z) = 1/(1 + e^{-z})$ yields the estimated probability of class $y = 1$.

Training. The network parameters are trained using the binary cross-entropy loss and the Adam optimizer (see Section 3.3) over several thousand epochs. The training loss as a function of epochs shows a good convergence behavior (decay, temporary-saturation and then decay).

Decision boundary. After training, the learned decision boundary of the NN is visualized in the left panel of Fig. 4.1, which shows the estimated probability of class 1 across the input domain. For comparison, the logistic regression model with polynomial features is visualized in the right panel of Fig. 4.1. Both are overlaid on the training data.

Observation. Both models are capable of approximating the nonlinearly separable boundary. The key difference is conceptual:

- Logistic regression relies on an explicit polynomial feature map chosen by the model designer.
- The NN learns its own internal representation through the hidden layer, with the feature map encoded in its weights and activations.

This illustrates the shift from manual feature engineering to learned representations.

Exercise 4.1.1 (Comparing Logistic Regression and a Single-Hidden-Layer NN). Use the notebook `LogReg+NN-supervised-2D.ipynb` to perform the following steps.

1. **Implement and train both models.**

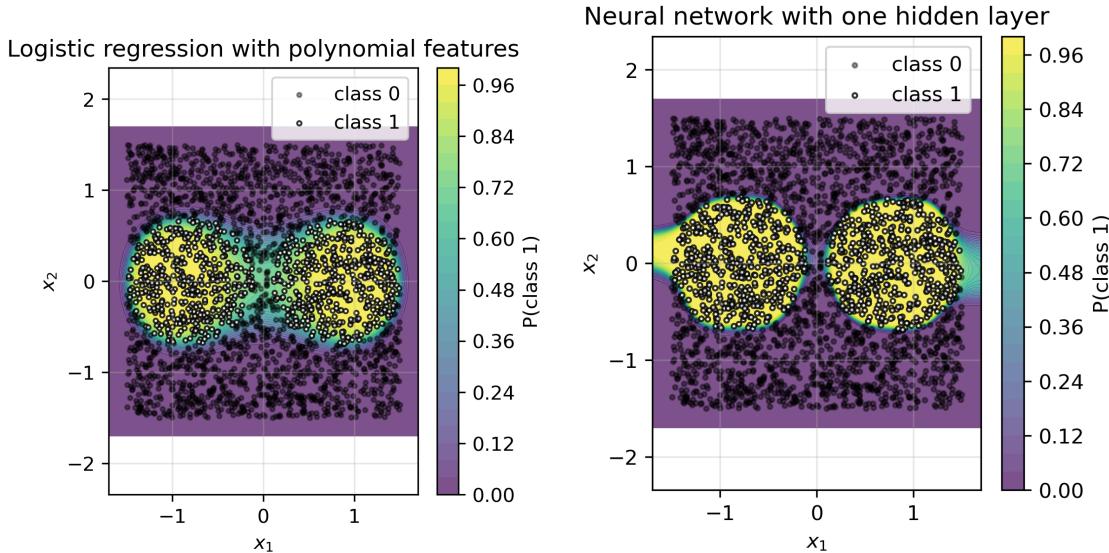


Figure 4.1: Left: Decision boundary learned by logistic regression with degree-4 polynomial features. The color encodes the predicted probability of class 1. Right: Decision boundary learned by the neural network with one hidden layer. The network learns a flexible nonlinear separation that closely matches the ground-truth two-circle structure.

- (a) Train logistic regression with a polynomial feature expansion of degree d (e.g., $d = 2, 3, 4$).
- (b) Train the single-hidden-layer NN with 10 hidden units and tanh activation, using binary cross-entropy loss and the Adam optimizer.

2. Quantitative comparison.

- Record training and test accuracies for both models.
- Compare how the results change as you vary the polynomial degree in logistic regression and the number of hidden units in the NN.

3. Qualitative comparison of decision boundaries.

- Reproduce and inspect figures similar to Figs. 4.1.
- Comment on regions where the models differ significantly from the ground truth.

4. Discussion.

- Under what conditions does logistic regression with polynomial features perform comparably to the NN?
- When does the NN provide a clear advantage in terms of approximation quality or robustness to hyperparameter choices?
- Relate your observations to the role of learned representations discussed later in this chapter.

From Hand-Crafted Features to Learned Representations

A key transition from classical machine learning to modern neural networks lies in **where the nonlinearity comes from**. Logistic Regression with polynomial or radial features relies on an *explicitly engineered* feature map $x \mapsto \phi(x)$, chosen by the model designer. Its expressive power is determined entirely by this choice.

In contrast, a neural network introduces a *learned* nonlinear feature map through its hidden layers:

$$x \longmapsto h(x) = \sigma(W_1 x + b_1),$$

where the weights and biases are optimized jointly with the classifier. Thus, the representation itself adapts to the data. This shift has profound consequences:

- Feature engineering is replaced by **representation learning**.
- Model capacity scales by adding layers or neurons rather than manually expanding the feature set.
- Optimization (Chapter 3) becomes central – not only for fitting a classifier, but for shaping the internal geometry of learned features.

In the simple 2D example above, both approaches succeed, but the neural network **learns its own features**, illustrating on a small scale the principle that underlies modern deep architectures – ResNets, Transformers, and diffusion-model backbones explored later in this book.

4.1.3 Interpolation vs. Extrapolation: Polynomial Regression vs Neural Networks

In this subsection we analyze **overfitting** (or its surprising absence) in the *interpolation regime* by studying a simple one-dimensional regression problem $f : \mathbb{R} \rightarrow \mathbb{R}$. We compare two families of models:

- **Polynomial regression** of increasing degree.
- A **single-hidden-layer fully connected neural network**, as introduced in Section 4.1.2.

Both are trained on a small number of data points in a bounded interval, and then evaluated both inside this interval (interpolation) and outside (extrapolation). The experiments are implemented in the Jupyter/PyTorch notebook `InterPoly-vs-NN.ipynb`.

Example 4.1.2. *Interpolation vs Extrapolation in 1D: Polynomials and a Neural Network*
We consider a smooth non-polynomial target function

$$f(x) = e^{-x^2} \cos(3x), \quad x \in \mathbb{R}.$$

A small training set of $n_{\text{train}} = 5$ points $\{(x_i, y_i)\}_{i=1}^{n_{\text{train}}}$ is sampled uniformly in the interval $[-1, 1]$, with $y_i = f(x_i)$. This interval is the **interpolation region**. We then examine model

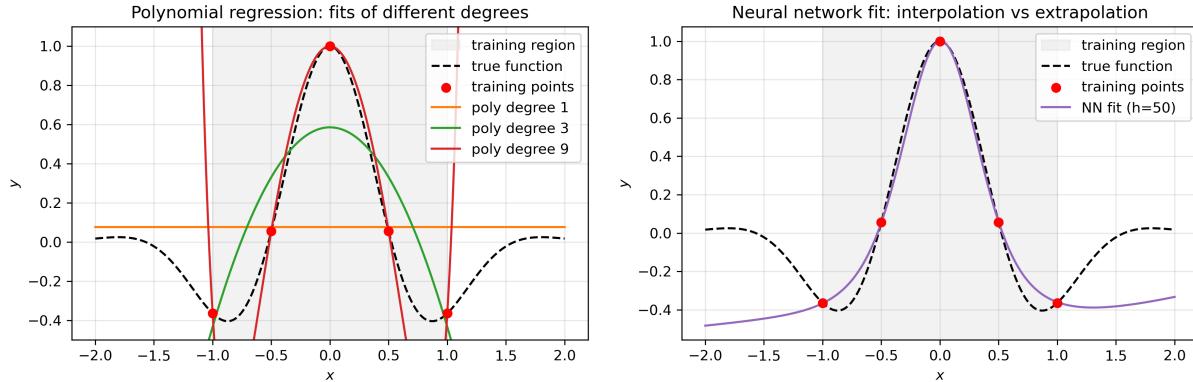


Figure 4.2: Left: Polynomial regression for degrees $d = 1, 3, 9$ trained on the same 5 data points (red) in the interval $[-1, 1]$ (shaded). For $d = 9 > n_{\text{train}} - 1$, the polynomial interpolates all training points but develops strong oscillations, especially near the edges and outside the training region. Right: Neural-network fit (solid curve) versus the true function (dashed) and training points (red). Inside the training region $[-1, 1]$ (shaded), the NN provides a smooth interpolation without oscillations, even though the number of parameters ($2h + 2$) greatly exceeds the number of data points. Outside $[-1, 1]$, however, the NN also fails to follow the true function, illustrating limited extrapolation.

behavior on a wider domain $[-2, 2]$, which includes both interpolation and extrapolation regions.

Polynomial regression. Given the training data $\{(x_i, y_i)\}$, we fit a polynomial of degree d ,

$$P_d(x; w) = w_0 + w_1 x + \cdots + w_d x^d,$$

by minimizing the mean squared error (MSE)

$$\mathcal{L}(w) = \frac{1}{n_{\text{train}}} \sum_{i=1}^{n_{\text{train}}} (P_d(x_i; w) - y_i)^2.$$

Left panel in Fig. 4.2 shows polynomial fits of degrees $d \in \{1, 3, 9\}$ over the wide interval $[-2, 2]$.

Overfitting in the interpolation regime. From the left panel of Fig. 4.2 we observe:

- For **low degree** ($d = 1$), P_d is too rigid to approximate f well; it underfits even inside the training interval.
- For **moderate degree** ($d = 3$), P_d fits the data reasonably in the interpolation region and behaves smoothly.
- For **high degree** ($d = 9 > n_{\text{train}} - 1$), the polynomial can pass exactly through all training points but exhibits sharp oscillations between them and near the boundary of $[-1, 1]$. This is a classical form of overfitting or memorization.

Neural network with one hidden layer. We now consider a simple fully connected neural network with one hidden layer:

$$\hat{y}(x) = W_{out} \tanh(W_{hidden}x + b_{hidden}) + b_{out}.$$

Here $W_{hidden} \in \mathbb{R}^{h \times 1}$, $W_{out} \in \mathbb{R}^{1 \times h}$, and h is the number of hidden neurons (we take, e.g., $h = 50$). The total number of trainable parameters is $2h + 2$, which can be much larger than n_{train} .

The network is trained using the MSE loss and the Adam optimizer, as in Section 3.3. The learned fit on $[-2, 2]$ is shown in the right panel of Fig. 4.2.

Key observations.

- In the **interpolation regime** (inside $[-1, 1]$), high-degree polynomials tend to overfit and oscillate, while the neural network produces a much smoother fit, despite being heavily overparameterized.
- In the **extrapolation regime** (outside $[-1, 1]$), both models deviate substantially from the true function; increasing polynomial degree or the number of hidden units does not automatically improve extrapolation.

This illustrates that neural networks can generalize smoothly in the interpolation regime, even with far more parameters than data points, yet still share the fundamental limitation that neither model extrapolates reliably without additional structure or prior knowledge.

Exercise 4.1.2 (Interpolation, Overfitting, and Extrapolation). Use the notebook *InterPoly-vs-NN.ipynb* to explore the following questions.

1. Varying polynomial degree.

- Fix $n_{train} = 5$ and vary the degree d (e.g., $d = 1, 3, 5, 9, 15$).
- For each d , plot $P_d(x)$ together with $f(x)$ and the training points on $[-2, 2]$.
- Identify the smallest d for which you observe pronounced oscillations in the interpolation region.

2. Varying network width.

- Fix the number of training points and vary the number of hidden neurons h (e.g., $h = 5, 20, 50, 100$).
- For each h , record the final training loss and plot the NN fit on $[-2, 2]$.
- Compare how the smoothness of the NN fit depends on h .

3. Interpolation vs extrapolation.

- For a representative polynomial degree (e.g., $d = 9$) and network width (e.g., $h = 50$), compare the approximation error inside $[-1, 1]$ and outside.
- Discuss why both models fail to capture $f(x)$ reliably outside the training region, despite excellent performance inside.

4. Inductive bias.

- Relate your observations to the notion of inductive bias: what kinds of functions are favored by high-degree polynomials vs one-hidden-layer NNs?
- How might these biases change when we move to deeper architectures (multi-layer NNs, ResNets) discussed later in this chapter?

4.1.4 Simple Convolutional Neural Network

The fully connected NN layer, which was the building block in the preceding subsections, is deliberately *structure-agnostic*: it treats all input coordinates as unrelated. In many practical settings, however, we *do* know something about the structure of the data. For images, for example, it is natural to assume that *nearby pixels are more strongly related than distant ones*, and that the same local pattern (such as an edge or corner) may appear anywhere in the image.

Convolutional Neural Networks (CNNs) exploit exactly this kind of prior structure:

- They connect each neuron only to a *local receptive field* in the previous layer, enforcing **locality**.
- They *reuse the same filter weights* across all locations, implementing **weight sharing** and a form of translation equivariance.

These design choices dramatically reduce the number of parameters compared to fully connected layers and lead to architectures that are particularly effective for images. We introduced discrete convolutions already in Section 1.1.3; here we see how they appear inside a concrete NN architecture.

Network construction (architecture). Consider an input image $x \in \mathbb{R}^{H \times W \times C}$ with height H , width W , and C channels (for MNIST, $H = W = 28$, $C = 1$). A single convolutional layer with K filters $F_k \in \mathbb{R}^{f \times f \times C}$ produces K feature maps $z_k \in \mathbb{R}^{H \times W}$ via

$$z_k(i, j) = \sigma \left(\sum_{m=0}^{f-1} \sum_{n=0}^{f-1} \sum_{c=1}^C x_c(i+m, j+n) F_k(m, n, c) + b_k \right), \quad k = 1, \dots, K,$$

where $\sigma(\cdot)$ is a point-wise nonlinearity (e.g., ReLU) and b_k is a bias term. In practice, we often include padding and striding to control the spatial resolution.

A typical CNN block alternates:

- **Convolution + nonlinearity**: local feature extraction with shared parameters.
- **Pooling** (e.g., max pooling): local down-sampling that reduces spatial resolution while keeping the number of channels.

After several such blocks, the feature maps are flattened and fed into one or more *fully connected* layers that produce class logits.

In our running example we use the following small CNN:

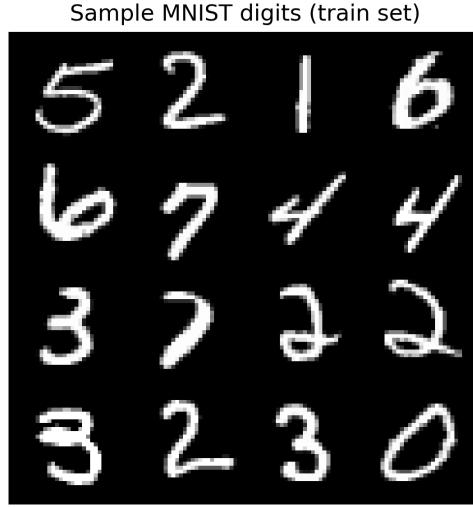


Figure 4.3: Sample MNIST digits from the training set (grayscale 28×28 images). Each image is labeled with a digit 0–9. Figure generated by the notebook `cnn-simple-MNIST.ipynb`.

- **Conv layer 1:** $1 \rightarrow 16$ channels, 3×3 kernels, padding 1 (keeps 28×28).
- **MaxPool 1:** 2×2 pooling $\Rightarrow 14 \times 14 \times 16$.
- **Conv layer 2:** $16 \rightarrow 32$ channels, 3×3 kernels, padding 1 (keeps 14×14).
- **MaxPool 2:** 2×2 pooling $\Rightarrow 7 \times 7 \times 32$.
- **Fully connected head:** flatten to a vector of size $7 \cdot 7 \cdot 32 = 1568$, apply a dense layer with 128 hidden units and ReLU, then a final dense layer with 10 outputs (class logits).

Fig. 4.3 shows example MNIST digits that serve as inputs to this architecture.

Optimization: training with cross-entropy and mini-batch SGD. The CNN is trained as a multi-class classifier. For a given input x and label $y \in \{0, \dots, 9\}$, the network produces logits $\ell_y(x; \theta)$ and class probabilities via the softmax

$$p(y | x, \theta) = \frac{\exp(\ell_y(x; \theta))}{\sum_{y'} \exp(\ell_{y'}(x; \theta))}.$$

The standard **cross-entropy loss** for one sample is

$$L(\theta; x, y) = -\log p(y | x, \theta).$$

For a mini-batch \mathcal{B} , the batch loss is

$$L_{\text{batch}}(\theta) = \frac{1}{|\mathcal{B}|} \sum_{(x_i, y_i) \in \mathcal{B}} L(\theta; x_i, y_i).$$

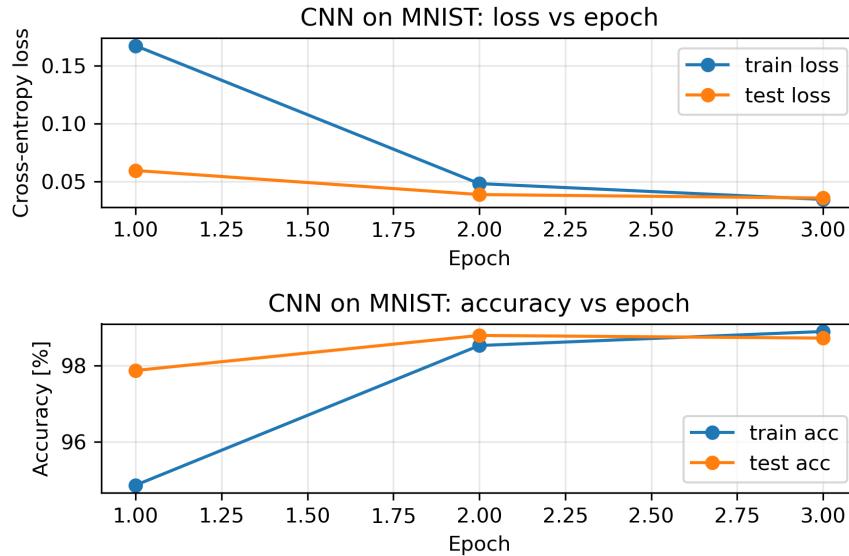


Figure 4.4: Training and test loss (top) and accuracy (bottom) versus epoch for the simple CNN on MNIST. The model quickly reaches high accuracy, illustrating the effectiveness of convolutional architectures on structured image data.

Training consists of solving

$$\min_{\theta} L_{\text{batch}}(\theta)$$

approximately using a gradient-based optimizer. In practice, we update θ iteratively using mini-batch SGD or one of its variants (e.g., Adam) as in Section 3.3:

$$\theta \leftarrow \theta - \eta \nabla_{\theta} L_{\text{batch}}(\theta).$$

Gradients are computed efficiently via reverse-mode automatic differentiation (AD), introduced in Section 2.1.

In the notebook `cnn-simple-MNIST.ipynb` we train the above CNN using Adam and record both training and test loss/accuracy across epochs. The resulting learning curves are shown in Fig. 4.4.

Inference. Once trained, the CNN defines a mapping $x \mapsto p(y | x, \theta)$ from images to class probabilities. For a new test image x , the network performs a forward pass and outputs $p(y | x, \theta)$; the predicted class is

$$\hat{y} = \arg \max_{y \in \{0, \dots, 9\}} p(y | x, \theta).$$

Even our small CNN achieves strong accuracy on the MNIST test set; however, it still makes mistakes. Fig. 4.5 shows representative misclassified digits, which are useful for diagnosing model limitations and dataset ambiguities. We will study MNIST misclassifications again in Chapter 6, using information-theoretic diagnostics to understand how and why different models diverge in their uncertainty and predictions.

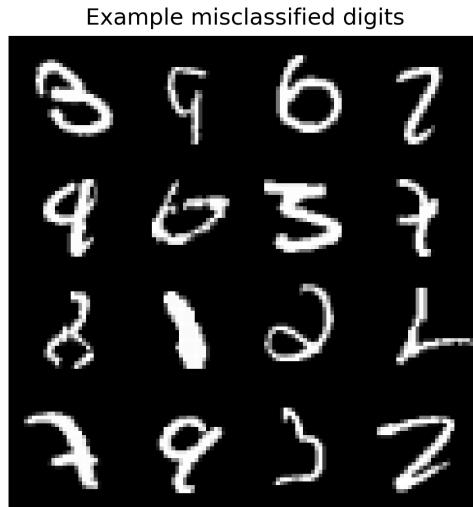


Figure 4.5: Example misclassified MNIST digits (true labels vs predicted labels are shown in the notebook output). Such examples highlight the limits of the learned representation and the importance of architecture, data quality, and optimization.

Principles for choosing the number of channels. The number of channels (filters) in a CNN controls how many distinct feature types the network can represent at each layer. There is no closed-form rule for the optimal number of channels; instead, several empirical principles are used:

1. **Shallow layers: low-level features.** Early layers detect simple patterns (edges, corners, textures). A moderate number of channels (e.g., 16–64) is typically sufficient to capture such local structures.
2. **Deeper layers: abstract features.** As depth increases and spatial resolution decreases (due to pooling or striding), layers need more channels to represent a richer variety of high-level patterns (object parts, whole digits, faces, . . .). Many successful architectures increase channel counts with depth (e.g., $32 \rightarrow 64 \rightarrow 128 \rightarrow 256$).
3. **Computational budget.** More channels mean more parameters and higher computational cost. Choices must balance accuracy with runtime and memory, especially in resource-constrained environments.
4. **Dataset complexity.** Simple datasets like MNIST can be handled with relatively few channels (e.g., 16 and 32 in this example). Complex datasets (e.g., ImageNet) benefit from deeper networks with hundreds of channels per layer.
5. **Empirical tuning and architectural templates.** Popular CNN architectures (AlexNet, VGG, ResNet, . . .) provide practical templates. A common heuristic is to roughly *double* the number of channels whenever the spatial resolution is halved.

These rules of thumb are widely used in practice but still lack full theoretical justification, making them a natural target for future theory.

Exercise 4.1.3 (Exploring the Impact of Filter Count and Size in a CNN). *Using the Jupyter/PyTorch notebook `cnn-simple-MNIST.ipynb`, investigate how varying the number of filters and the filter size affects CNN performance on MNIST.*

1. Vary the number of filters.

- *Modify the first and second convolutional layers to use different channel counts (e.g., (8, 16), (16, 32), (32, 64)).*
- *For each configuration, train the network for the same number of epochs and record training and test accuracy.*
- *Discuss the trade-off between model capacity and overfitting/underfitting: larger models fit the training data better but may generalize worse if the dataset is small.*

2. Vary the filter size.

- *Change the kernel size from 3×3 to 5×5 or 7×7 , adjusting padding as needed to preserve spatial dimensions.*
- *Compare classification accuracy and training time for different kernel sizes.*
- *Analyze how larger kernels affect the effective receptive field and whether they improve or degrade performance on MNIST.*

3. Convergence behavior.

- *For each architectural variant, plot training and test loss across epochs (as in Fig. 4.4).*
- *Compare convergence speed and final accuracy as a function of channel count and kernel size.*

4. Summary.

- *Summarize your findings about how filter number and size influence expressiveness, overfitting, and computational cost.*
- *Discuss how these lessons might change when moving from MNIST to more complex datasets or to deeper CNNs such as ResNets.*

4.2 Neural Architectures

We have already experimented with the MNIST database – a collection of small handwritten digit images that has become one of the most widely used benchmarks for testing ideas in Convolutional Neural Networks (CNNs). Indeed, MNIST was a central example in one of the earliest and most influential works that helped shape the modern Deep Learning (DL) era: LeCun et al. (1998) [18]. That work demonstrated, in a convincing and scalable way, how CNNs can exploit the **spatial structure** of image data through three key architectural principles:

- **Local receptive fields** (nearby pixels are more correlated than distant ones),
- **Weight sharing** (translational equivariance),
- **Hierarchical feature extraction** (from edges to shapes to semantic content).

These ideas established the basic architectural template that would guide deep-learning research for two decades.

The AlexNet turning point. Although CNNs were known earlier, the breakthrough that triggered the explosive growth of deep learning came with AlexNet [19]. Its impact derived not only from its convolutional components but from the demonstration that *deep* and *heterogeneous* architectures – consisting of multiple convolutional layers, interleaved pooling, and fully connected layers – could be trained reliably at scale using **GPU acceleration**. AlexNet effectively launched the *scaling era of Deep Learning*, showing that performance improves dramatically when models, data, and compute are scaled in tandem. This insight underlies virtually all contemporary state-of-the-art systems.

Beyond AlexNet: major architectural milestones. Several architectural innovations following AlexNet have shaped modern DL and influenced the development of contemporary generative AI. Each is rooted in a distinct inductive bias and contributes new mathematical insights or optimization strategies.

- **Autoencoders [20]:** Autoencoders provide a foundational framework for nonlinear **compression** and **representation learning**. In contrast to the linear PCA–SVD compression methods discussed in Section 1.2.4, autoencoders learn *nonlinear* low-dimensional latent spaces, enabling tasks such as denoising, generative modeling, and unsupervised feature discovery.
- **U-Net [21]:** Originally developed for medical image segmentation, U-Net introduced a novel **encoder–decoder** geometry enhanced by **skip connections** that preserve fine-scale spatial information. This architecture remains a backbone for many modern high-resolution image-to-image models and serves as a precursor to the U-shaped architectures used in diffusion models.
- **ResNet [22]:** ResNet introduced **residual (skip) connections**, addressing the vanishing-gradient bottleneck and enabling the stable training of networks with hundreds or thousands of layers. This development is essential for understanding the continuous-depth limit of neural networks, which leads naturally to **Neural ODEs** and diffusion-model architectures discussed later.
- **Transformers [1]:** Transformers replace convolutional locality with a flexible, global **self-attention mechanism**. Initially a revolution in Natural Language Processing, Transformers have since migrated to computer vision (e.g., Vision Transformers, ViTs), where they challenge the dominance of CNNs by removing fixed spatial inductive biases and enabling dynamic, data-driven receptive fields.

Each of these architectures is tied to a core mathematical theme: nonlinear approximation, hierarchical representation learning, compression, and optimization under increasingly large parameter budgets. Autoencoders and U-Nets reappear in Section 6.3.4, where we connect them to information-theoretic views of compression. Transformers were already introduced in Chapters 1–2 and will be revisited in later chapters devoted to the synthesis of modern generative AI.

Focus of this section. In the remainder of this section we will examine **ResNet** and its continuous-depth interpretation, leading naturally into the mathematics of **Neural ODEs**. These architectures introduce new perspectives on depth, stability, optimization, and expressivity—concepts that form the backbone of contemporary generative modeling frameworks, including diffusion models and flow-based models, which we encounter in later chapters. Before diving into ResNet, it is worth emphasizing a broader practical point: *architectural design is an exercise in balancing model capacity, computational constraints, data availability, and inductive bias*. Modern state-of-the-art neural architectures often contain **billions of parameters**, making choices of depth, width, locality, normalization, and parameter sharing essential both for model generalization and for computational feasibility.

4.2.1 From CNN to ResNet – the Power of Skip Connections

The step from a plain CNN to a ResNet may look minor architecturally, but it has had a *major* impact on practice. The key idea is to make each layer (or block) learn a *residual* correction on top of an identity map, rather than a full transformation from scratch.

A basic **Residual Block** (RB) in a ResNet consists, schematically, of two convolutional layers plus an identity (skip) connection:

$$\text{RB}(x) = \sigma\left(\text{BN}\left(F_2\left(\sigma(\text{BN}(F_1(x)))\right)\right) + x\right),$$

where

- F_1 and F_2 are convolutional layers,
- $\text{BN}(\cdot)$ is batch normalization,
- $\sigma(\cdot)$ is a pointwise nonlinearity (e.g. ReLU),
- the *skip connection* is the additive $+x$ term.

Without the skip connection, we would simply have $\text{RB}(x) \approx g(x)$ for some nonlinear g ; with the skip, the block instead learns $x \mapsto x + f(x)$, where f is typically “small” in a suitable sense.

Continuous-depth perspective and ODE analogy. To interpret this more formally, recall from Chapter 2 that a standard deep network without skip connections can be viewed as the composition

$$h_L = F_L \circ F_{L-1} \circ \cdots \circ F_1(h_0),$$

where h_0 is the input and F_ℓ are layer maps. If we imagine L large and the changes per layer small, this composition suggests a continuous-depth limit

$$\frac{dh(t)}{dt} = f(h(t), \theta(t)), \quad (4.1)$$

where t plays the role of (continuous) depth, $h(t)$ is the feature representation at depth t , and $\theta(t)$ collects layer parameters.

ResNet makes this analogy explicit: a residual block applies an update of the form

$$h_{t+1} = h_t + f(h_t, \theta_t), \quad (4.2)$$

which is precisely the forward Euler discretization of the ODE

$$\frac{dh(t)}{dt} = f(h(t), \theta(t)).$$

In other words, each block performs a *small* step in feature space, starting from h_t and moving to $h_t + f(h_t, \theta_t)$. Empirically, such residual updates

- ease optimization in very deep networks,
- help mitigate vanishing/exploding gradients (since Jacobians are closer to the identity, rather than arbitrary products of nonlinear maps),
- make the connection to continuous-time dynamics and Neural ODEs natural.

We will return to this ODE-based viewpoint in Section 4.2.2.

Batch Normalization. Most modern ResNet implementations also use **Batch Normalization** (BN) inside each block to stabilize training. For an intermediate pre-activation z , BN computes

$$\text{BN}(z) = \gamma \frac{z - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta,$$

where μ and σ^2 are the mini-batch mean and variance; γ, β are learnable scale and shift parameters; and $\epsilon > 0$ is a small constant. BN keeps activation statistics more stable across layers and epochs, which in turn supports deeper architectures and larger learning rates.

ResNet-9 as a small residual architecture. Full-scale ResNets used in computer vision (ResNet-18, -34, -50, ...) are deep CNNs built from many residual blocks. For didactic purposes and toy problems, smaller variants such as *ResNet-9* are often used. These retain the essential pattern

$$(\text{Conv} + \text{BN} + \text{ReLU}) \rightarrow (\text{Residual blocks}) \rightarrow (\text{Global pooling} + \text{classifier}),$$

while drastically reducing depth and parameter count. In image applications, ResNet-9 uses convolutional residual blocks; in our 2D spiral example below we adopt the same skip-connection pattern but in a fully connected feature space.

Example 4.2.1 (Residual Network on a 2D Spiral Dataset). *To illustrate the effect of skip connections on optimization and decision geometry, we consider a synthetic 2D classification task where three classes form spiral arms in \mathbb{R}^2 . The corresponding Jupyter/PyTorch notebook is `ResNet9-Spiral.ipynb`.*

- **Data.** We generate $N = 3n$ points (x_i, y_i) with $x_i \in \mathbb{R}^2$ and $y_i \in \{0, 1, 2\}$, where each class follows a noisy spiral arm. The resulting dataset is not linearly separable and features highly curved decision boundaries; see the left panel of Fig. 4.6.
- **Architecture.** The model is a small fully connected ResNet-style network:

1. an input linear layer mapping $\mathbb{R}^2 \rightarrow \mathbb{R}^{64}$, followed by ReLU;
2. four residual blocks acting in \mathbb{R}^{64} , each of the form

$$h \mapsto \sigma(h + g(h)),$$

- where g is a two-layer MLP with optional batch normalization;
3. a final linear layer mapping \mathbb{R}^{64} to 3 class logits.

This mirrors the ResNet pattern in a low-dimensional feature space.

- **Training.** We train the network with cross-entropy loss and the Adam optimizer over a moderate number of epochs, recording the training loss and accuracy curves; see Fig. 4.7. The network quickly attains high classification accuracy on the spirals.
- **Decision regions.** After training, we evaluate the classifier on a dense grid in the (x_1, x_2) -plane and color each point by the predicted class. The right panel of Fig. 4.6 shows that the residual network learns a smooth but nontrivial partition of the plane that wraps around the spiral arms.

Fig. 4.7 also shows a snapshot of gradient norms across layers for a single mini-batch, which we use as a baseline in the exercise below.

Exercise 4.2.1 (Effect of Skip Connections in a ResNet-style Network). *Building on Example 4.2.1 and the notebook `ResNet9-Spiral.ipynb`, investigate the role of skip connections in training the spiral classifier.*

1. **Remove skip connections.** Modify the model so that residual blocks become plain two-layer Multi-Layer-Perceptron (MLP) blocks, i.e., replace

$$h \mapsto \sigma(h + g(h))$$

by

$$h \mapsto \sigma(g(h)).$$

Train this non-residual network under the same conditions (optimizer, learning rate, number of epochs) and compare its training loss and accuracy curves to those in Fig. 4.7.

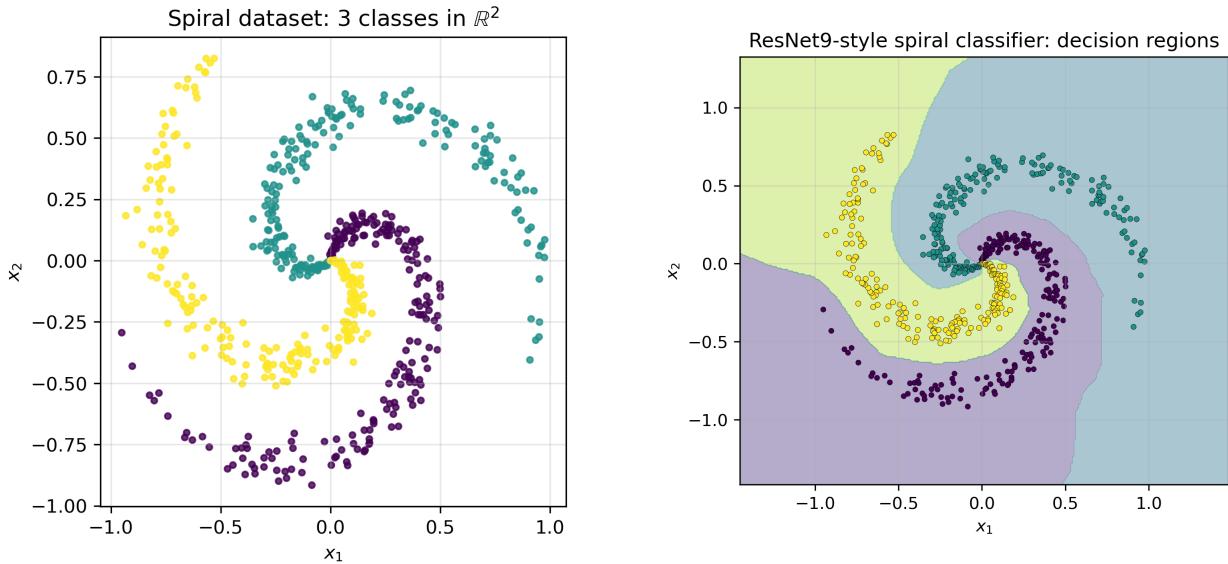


Figure 4.6: **Left:** Spiral dataset in \mathbb{R}^2 (three classes). **Right:** Decision regions learned by the ResNet-style model from Example 4.2.1. Figures generated by `ResNet9-Spiral.ipynb`.

2. **Decision boundaries.** For the non-residual network, recompute and plot the decision regions in the (x_1, x_2) -plane, as in Fig. 4.6. Compare the shapes of the learned decision boundaries with and without skip connections.
3. **Gradient flow.** For both models (with and without skip connections), compute gradient norms for each layer using a single mini-batch (as in the notebook) and plot them on a log scale. Discuss how skip connections influence the distribution of gradient norms across depth and how this relates to the vanishing-gradient phenomenon.
4. **Summary.** Summarize your observations:
 - How does removing skip connections affect training stability and convergence?
 - How do skip connections impact gradient flow quantitatively (via gradient norms) and qualitatively (via optimization behavior)?
 - How do these findings support the ODE-based interpretation $h_{t+1} = h_t + f(h_t, \theta_t)$ as a sequence of small, stable updates?

From Discrete Layers to Continuous Depth

Residual Networks (ResNets) introduce *skip connections* that reinterpret each layer as a small *incremental update* rather than a full transformation. This idea allows a deep network to evolve its representation gradually, layer by layer. When we view the layer index as a discretized “time” variable, the residual update

$$h_{k+1} = h_k + f_\theta(h_k) \Delta t$$

resembles an explicit Euler step for an ordinary differential equation (ODE).

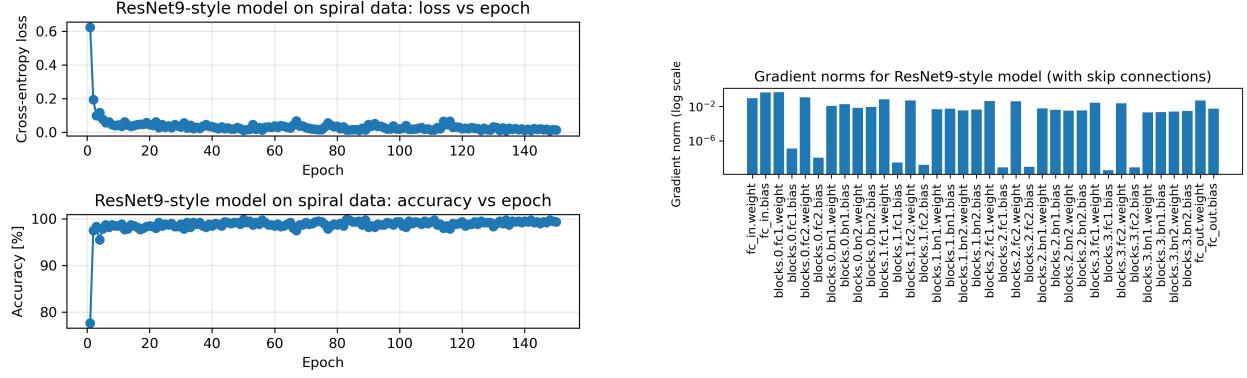


Figure 4.7: **Left:** Training loss and accuracy versus epoch for the ResNet-style model on the spiral dataset. **Right:** Gradient norms of individual layers (log scale) for a single mini-batch, illustrating how gradients propagate through the residual architecture.

This observation motivates a conceptual shift:

- Rather than stacking finitely many discrete layers, we may imagine a **continuous-depth model** governed by an ODE $\dot{h}(t) = f_\theta(h(t))$.
- The architecture of the network is now encoded in the choice of vector field f_θ and the numerical solver used to integrate it.
- Training requires differentiating *through the ODE solver*, giving rise to the **Neural ODE** framework.

Thus the small modification introduced by skip connections in ResNets opens a direct path to treating deep networks as continuous-time dynamical systems. The next subsection develops this perspective and illustrates it on a simple but instructive example: learning the dynamics of a 2D spiral.

4.2.2 From Residual Networks to Neural ODEs: A 2D Spiral Example

Residual networks can be viewed as an explicit Euler discretization of a continuous-depth evolution. Recall the residual update

$$h_{k+1} = h_k + \Delta t f_\theta(h_k), \quad k = 0, \dots, K-1, \quad (4.3)$$

which, in the limit $\Delta t \rightarrow 0$ and $K \rightarrow \infty$ at fixed final time T , suggests an underlying ordinary differential equation (ODE)

$$\frac{dh(t)}{dt} = f_\theta(h(t)), \quad h(0) = h_0, \quad (4.4)$$

with $h(t)$ playing the role of a “continuous layer index”. The Neural ODE viewpoint replaces the discrete stack of residual blocks by such a continuous-depth evolution, and delegates the

role of the “network architecture” to the choice of vector field f_θ and the numerical ODE solver used to approximate (4.4).

In practice, one chooses a parameterized vector field $f_\theta : \mathbb{R}^d \rightarrow \mathbb{R}^d$ and solves (4.4) with a standard ODE integrator. The simplest choice is explicit Euler,

$$h_{k+1} = h_k + \Delta t f_\theta(h_k),$$

but higher-order schemes such as Runge–Kutta methods, or adaptive solvers such as Dormand–Prince, are equally natural. The key algorithmic idea of Neural ODEs [23] is to *differentiate through the solver* in order to back-propagate from a loss defined at the terminal time T (or along the entire trajectory) back to the parameters θ .

2D spiral setup. To illustrate these ideas in a tangible, low-dimensional setting, we consider a decaying spiral trajectory in \mathbb{R}^2 given by

$$x_{\text{true}}(t) = e^{-\alpha t} \begin{bmatrix} \cos(\omega t) \\ \sin(\omega t) \end{bmatrix}, \quad t \in [0, T], \quad (4.5)$$

with $\alpha > 0$ and $\omega > 0$. We sample (4.5) on a uniform grid $0 = t_0 < t_1 < \dots < t_N = T$ and corrupt the samples with small Gaussian noise, obtaining $\{x_k^{\text{noisy}}\}_{k=0}^N$. The task is then to learn a Neural ODE whose solution trajectory $x_\theta(t)$ matches these noisy observations.

The accompanying notebook `NeuralODE-Spiral.ipynb` implements this example using a fixed-step fourth-order Runge–Kutta (RK4) solver and a small neural network $f_\theta : \mathbb{R}^2 \rightarrow \mathbb{R}^2$. The initial condition is taken from the first noisy observation, $x_\theta(0) = x_0^{\text{noisy}}$, and the loss is the mean squared error over the entire time grid:

$$\mathcal{L}(\theta) = \frac{1}{N+1} \sum_{k=0}^N \|x_\theta(t_k) - x_k^{\text{noisy}}\|_2^2. \quad (4.6)$$

The network parameters θ are trained with Adam, while gradients are obtained by automatic differentiation through the RK4 integrator. Fig. 4.8 shows the underlying noiseless spiral, the noisy data, and the fitted Neural ODE trajectory; the right panel reports the training loss as a function of iteration.

A particular advantage of the dynamical viewpoint is that we can easily *extrapolate* beyond the training horizon $[0, T]$ by integrating (4.4) further in time. This reveals the inductive bias imposed by the learned vector field f_θ : in some parameter regimes the learned dynamics continues to spiral inward in a physically plausible way, while in others it may deviate or develop spurious oscillations. The extrapolation behavior is illustrated in Fig. 4.9.

Example 4.2.2 (Neural ODE fit of a 2D spiral). *The notebook `NeuralODE-Spiral.ipynb` constructs a 2D decaying spiral in \mathbb{R}^2 , corrupts it with Gaussian noise, and then learns a Neural ODE drift f_θ so that the RK4 solution $x_\theta(t)$ matches the noisy data. The drift is represented by a two-layer MLP with tanh nonlinearities. Training is carried out with Adam by backpropagating through the RK4 solver. The example exposes three important aspects of Neural ODEs:*

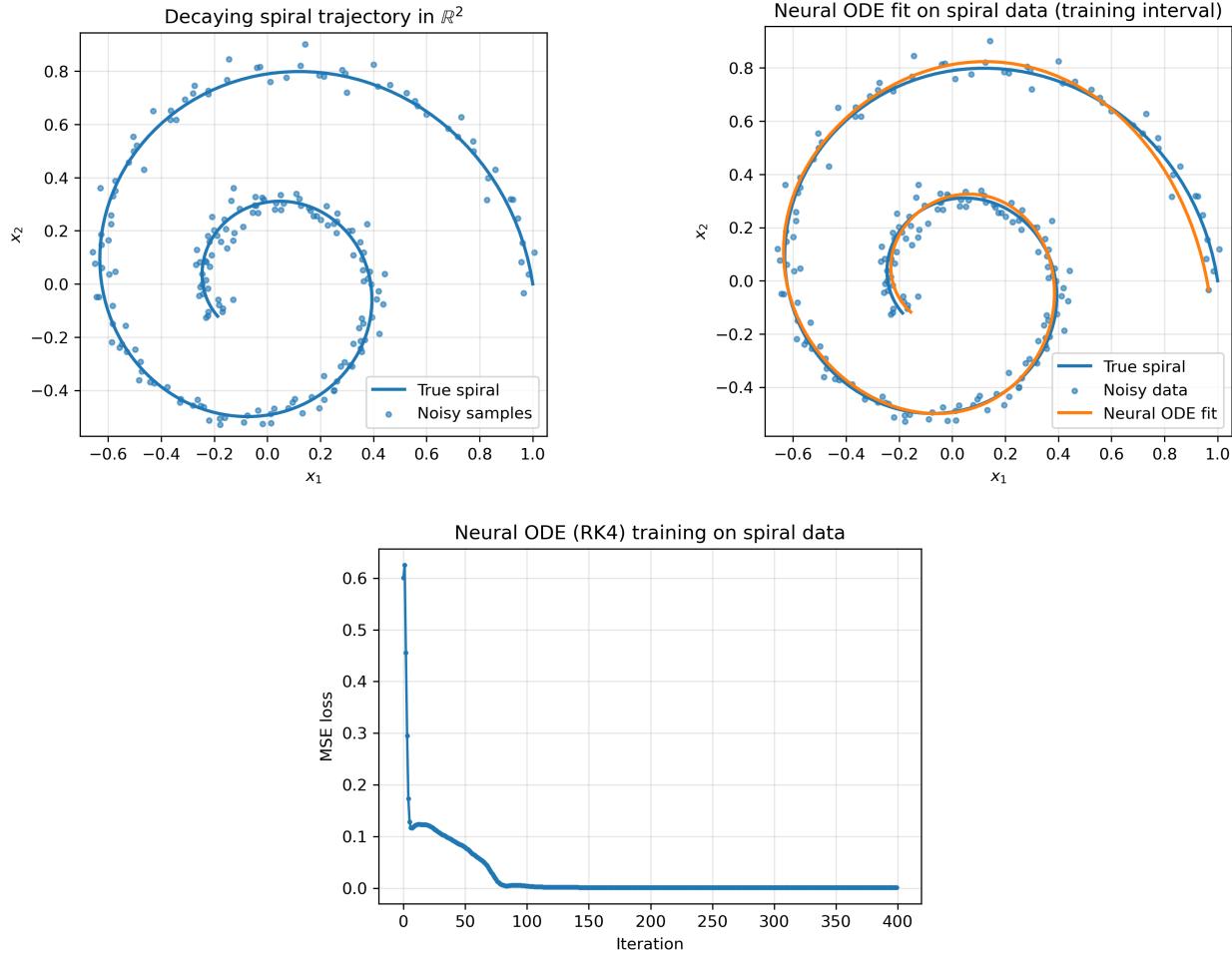


Figure 4.8: Top left: ground truth spiral trajectory (4.5) (solid curve) and noisy observations (dots). Top right: Neural ODE trajectory $x_\theta(t)$ fitted using a fixed-step RK4 solver, superimposed on the data. Bottom: training loss (4.6) versus optimization iteration. Generated by the notebook `NeuralODE-Spiral.ipynb`.

1. *The continuous-depth viewpoint:* the “depth” variable k of a residual network is replaced by a continuous time variable t , and the model is specified by a vector field f_θ .
2. *The role of the numerical solver:* changing the ODE integrator (Euler vs. RK4 vs. adaptive methods) changes the effective model class.
3. *Trajectory-level supervision:* the loss is defined over the entire time series $\{x_\theta(t_k)\}$, not just at a single terminal state.

Exercise 4.2.2 (Discrete vs. continuous depth on the spiral). *Using the same spiral dataset and time grid as in `NeuralODE-Spiral.ipynb`, construct a discrete-time model in the spirit of a residual network, e.g. by learning a map $g_\phi : \mathbb{R}^2 \rightarrow \mathbb{R}^2$ and iterating*

$$x_{k+1} = x_k + g_\phi(x_k), \quad k = 0, \dots, N-1.$$

Train ϕ to minimize the same trajectory loss as in (4.6), and compare:

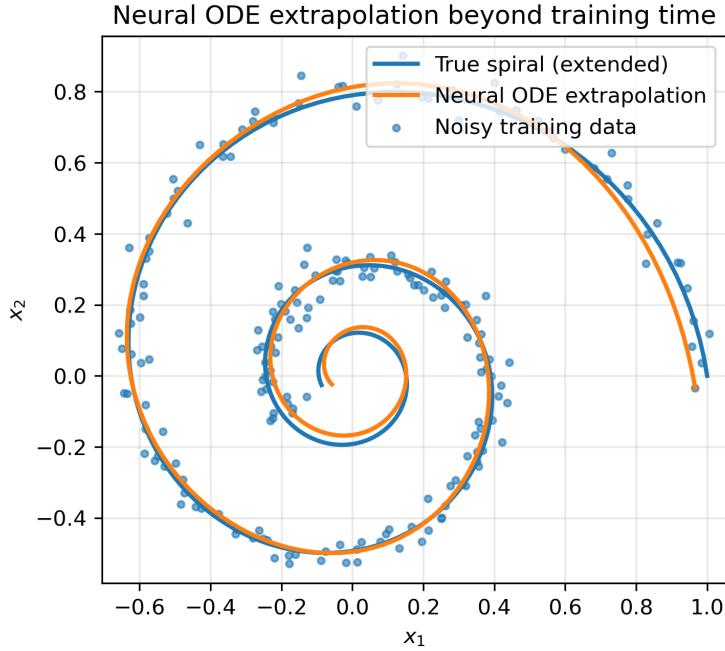


Figure 4.9: Extrapolation of the learned Neural ODE beyond the training window. Solid curve: ground truth spiral extended to a longer time interval; dashed curve: extrapolated Neural ODE trajectory. Noisy training data are shown as dots near the origin. Generated by `NeuralODE-Spiral.ipynb`.

1. *Quality of fit on the training interval $[0, T]$.*
2. *Extrapolation behavior beyond T .*
3. *Sensitivity of the learned dynamics to changes in the number of steps N (while keeping T fixed), for both the discrete and the continuous models.*

Discuss in which regimes the discrete residual model behaves similarly to the continuous Neural ODE, and where they differ qualitatively, both in terms of trajectory geometry and stability.

Adaptive solvers and the adjoint equation. In practice, Neural ODE implementations almost always rely on *adaptive* time-stepping, so that the solver chooses its own internal step sizes in order to control a local error estimate. This is illustrated in the companion notebook `NeuralODE-Adjoint-Spiral.ipynb`, which replaces the fixed-step RK4 integrator with a simple adaptive RK scheme based on step-doubling. The resulting model $x_\theta(t)$ still fits the noisy spiral (see Fig. 4.10), but now the number and size of solver steps depend on both the data and the current parameters θ .

From the variational perspective developed earlier in this chapter, gradients with respect to the initial condition and the parameters can be expressed in terms of an *adjoint state* $\lambda(t)$ solving a companion ODE backward in time. For a terminal loss

$$L = \frac{1}{2} \|x(T) - x_{tar}\|_2^2 \quad (4.7)$$

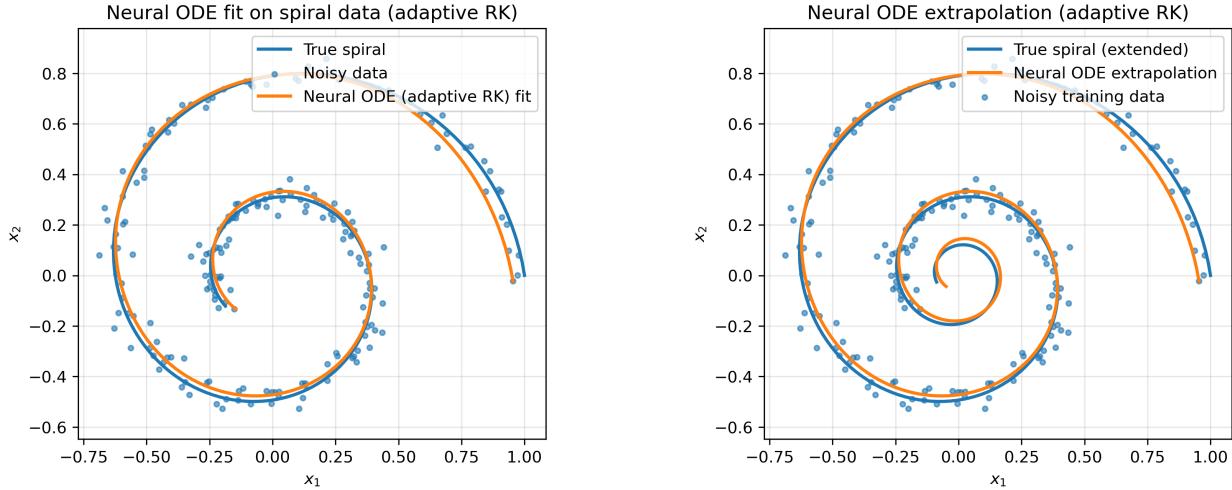


Figure 4.10: Left: Neural ODE fit of the spiral using an adaptive-step RK solver. Right: extrapolation of the adaptive solver beyond the training window. Generated by `NeuralODE-Adjoint-Spiral.ipynb`.

with x_{tar} fixed, the adjoint $\lambda(t)$ obeys

$$\dot{\lambda}(t) = -(\partial_x f_\theta(x(t)))^\top \lambda(t), \quad \lambda(T) = x(T) - x_{tar}. \quad (4.8)$$

The notebook `NeuralODE-Adjoint-Spiral.ipynb` computes $\lambda(t)$ for the trained spiral model and verifies numerically that $\lambda(0)$ coincides with the gradient of L with respect to the initial state $x(0)$:

$$\lambda(0) \approx \nabla_{x(0)} L, \quad (4.9)$$

in agreement with the general adjoint theory for ODE-constrained optimization. Fig. 4.11 shows the time evolution of the adjoint components $\lambda_1(t)$ and $\lambda_2(t)$ along the spiral.

Exercise 4.2.3 (From backpropagation to adjoint dynamics). *Starting from the implementation in `NeuralODE-Adjoint-Spiral.ipynb`, extend the code so that the adjoint state $\lambda(t)$ is used not only to recover $\nabla_{x(0)} L$ but also to assemble an approximation of the gradient $\nabla_\theta L$ via the continuous-time formula*

$$\nabla_\theta L = \int_0^T \lambda(t)^\top \partial_\theta f_\theta(x(t)) dt.$$

Compare the resulting gradient vector to the one obtained by automatic differentiation through the adaptive solver, both in norm and direction. Discuss numerical issues that arise (such as the need to recompute the forward trajectory or to store it in memory) and relate them to the trade-offs in full-fledged Neural ODE implementations.

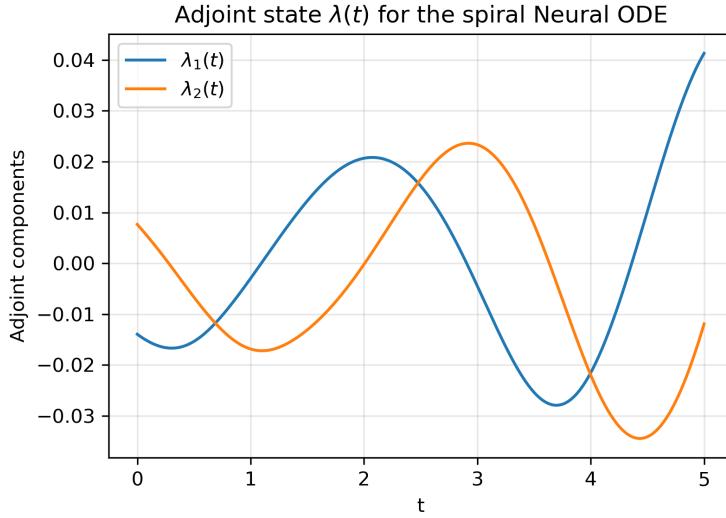


Figure 4.11: Adjoint state $\lambda(t)$ for the spiral Neural ODE, obtained by integrating the adjoint ODE (4.8) backward in time from the terminal condition $\lambda(T) = x(T) - x_{tar}$. The value $\lambda(0)$ matches the gradient of the terminal loss with respect to the initial condition. Generated by `NeuralODE-Adjoint-Spiral.ipynb`.

From architectures to universal phenomena in deep learning

The progression *CNN* → *ResNet* → *Neural ODE* revealed how architectural innovations increasingly impose structure on the transformations performed by a network: locality and equivariance in CNNs, stability and incremental updates in ResNets, and continuous-depth flows in Neural ODEs.

However, many striking behaviors of modern neural networks *do not depend on architecture at all*. Across CNNs, MLPs, Transformers, autoencoders, and ODE-based models, we repeatedly observe:

- optimization trajectories that preferentially settle in **flat**, high-volume minima;
- internal representations that collapse onto **low-dimensional manifolds**;
- implicit regularization induced by SGD, normalization layers, and network width.

In the following Section we analyze these effects through simple but revealing experiments which move us from *how networks are built* to *why trained networks behave the way they do*.

4.3 Universal Geometric Principles of Deep Learning

Having traced the evolution of architectures from CNNs to ResNets and finally to Neural ODEs, we now turn to a different – but equally fundamental—perspective: the *architecture-independent* geometric principles that govern how neural networks learn. Despite operating

in extremely high-dimensional parameter and activation spaces, trained networks consistently exhibit two remarkable phenomena. First, the dynamics of stochastic gradient descent bias solutions toward *flat, high-volume minima* in the loss landscape, a property closely tied to stability and generalization. Second, the internal representations of deep networks collapse onto *low-dimensional manifolds*, revealing an implicit form of structured compression.

The two subsections that follow examine these emergent behaviors in detail—first through the lens of SGD-induced flat minima, and then through PCA- and activation-based analyses of the intrinsic dimensionality of learned representations.

4.3.1 Discovery of Flat Regions in the Energy Landscape

The geometry of the loss landscape plays a critical role in determining a model’s generalization ability [24]. Stochastic Gradient Descent (SGD) tends to converge to **flat regions** in the energy landscape, which correspond to solutions that are robust to small perturbations in the data or model parameters. **Flat minima** are associated with better generalization, while **sharp minima** often lead to overfitting. The emergence of flat minima is influenced by the interaction between the **learning rate**, **batch size**, and the **controlled stochasticity** of SGD.

Example 4.3.1 (Neural-network decoding, SGD noise, and the geometry of minima). *To explore the highlighted feature of SGD in a setting that is both analytically simple and conceptually rich, we consider a three-bit **error-correcting code**. Let $x = (x_1, x_2, x_3)$ be a binary vector constrained by a single parity check*

$$x_1 + x_2 + x_3 \equiv 0 \pmod{2}.$$

The valid codewords are

$$(0, 0, 0), (1, 1, 0), (1, 0, 1), (0, 1, 1).$$

One of these four codewords is transmitted through an additive noisy channel, yielding a corrupted observation

$$y = x + \text{noise},$$

where $y \in \mathbb{R}^3$ and the noise is Gaussian. A small fully connected neural network is then trained to decode y and recover the most probable original codeword.

Setup. We consider a feedforward neural-network decoder

$$\phi_\theta : \mathbb{R}^3 \rightarrow [0, 1]^3$$

(with one hidden layer and three output logits) is trained to recover x from y using a bit-wise binary cross-entropy loss.

SGD noise via batch size. To study the influence of stochasticity, we train the same network using three batch sizes:

$$\text{batch} \in \{4, 32, \text{full batch}\},$$

all with the same learning rate. The accompanying notebook `NN-Decoding.ipynb` records:

- **Training loss trajectories.** Small batches produce noisy, spike-filled loss curves, whereas large batches have smoother, more monotone behavior.
- **Approximate Hessian eigenvalues** near the final solution. The Hessian is computed on a small subset of the training data via automatic differentiation.
- **Local two-dimensional slices** of the loss landscape obtained by probing the loss on a random 2D affine subspace through the solution.

What the results show. Fig. 4.12 summarizes representative outcomes. The most striking feature is the behavior of the training loss trajectories. Small batches produce visibly noisy dynamics: the loss fluctuates substantially from one step to the next and exhibits characteristic “spikes”—temporary increases in loss caused by high-variance gradient estimates. Batch size 32 displays a milder version of this phenomenon, with smaller oscillations and fewer spikes. In contrast, full-batch gradient descent follows a smooth, nearly monotone path, reflecting its low-variance updates. These differences illustrate a core principle of SGD: **optimization noise is controlled by batch size**, and this noise directly shapes the geometry of the optimization trajectory long before convergence.

By comparison, the curvature profiles near convergence show much subtler differences. Although small-batch SGD is often associated with flatter minima in large modern networks, the Hessian spectra for this tiny model—with only 59 parameters and a highly constrained decoding task—are remarkably similar across batch sizes. This is not surprising: in low-dimensional settings with strong inductive structure, the optimizer is effectively guided into the same broad basin regardless of the stochasticity level. Thus, while the trajectory-level effects of batch size are clearly visible, the curvature-level effects may remain muted in such small toy examples.

Local loss slices. A two-dimensional slice of the loss around the solution (Fig. 4.13) reveals a gently curved, broad basin. Such slices are helpful for visualizing qualitative geometry, even when Hessian spectra are not dramatically different.

Key observations. Even this small example illustrates essential conceptual points:

- Smaller batches induce noisy, irregular optimization trajectories with characteristic spikes.
- Flat minima do not always show sharply separated curvature signatures—especially in small models.
- Hessian spectra, loss slices, and trajectories together give a more complete picture of the loss landscape than any single metric.

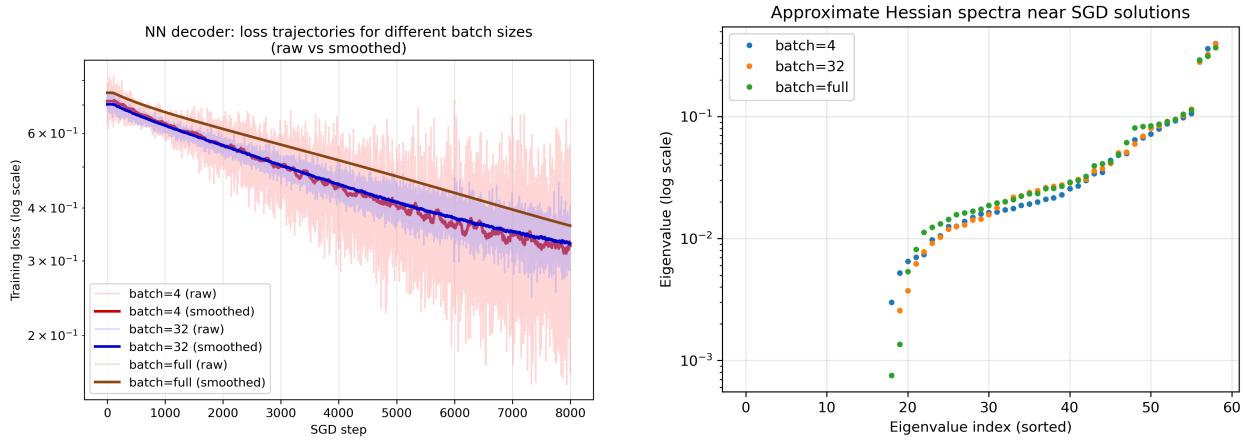


Figure 4.12: **Left:** Training loss (log scale) for different batch sizes. Small batches exhibit high-variance trajectories with distinct spikes; large batches produce smoother curves. **Right:** Approximate Hessian spectra near the trained solutions. In this small decoding task, all three runs converge to minima with remarkably similar curvature, illustrating that batch-size effects on flatness are problem-dependent and may be subtle in low-dimensional settings. Generated by `NN-Decoding.ipynb`.

Trajectory Noise vs. Curvature Effects

Small-batch SGD leaves a strong signature on the *optimization trajectory* – its noisy, spike-ridden path reflects high-variance gradient estimates. However, in small neural networks with strong inductive bias, this stochasticity does not necessarily translate into visibly different *curvature profiles* near the final solution: the optimizer is drawn into the same broad basin regardless of batch size.

This illustrates a general principle: **SGD's noise strongly shapes the path by which a solution is reached, but its effect on the local geometry of that solution depends on model scale and problem complexity.** Larger models and more flexible loss landscapes exhibit much richer curvature differences across batch sizes.

Exercise 4.3.1 (Exploring SGD dynamics and curvature in a toy decoding problem). *Extend the notebook `NN-Decoding.ipynb` to deepen your understanding of how SGD interacts with the loss landscape.*

1. **Batch size, learning rate, and flatness.** Compute the Hessian eigenvalue spectra for several combinations of batch size and learning rate. Compare:
 - the number of small eigenvalues (e.g., $\lambda < 10^{-2}$);
 - the largest eigenvalues.

Is there any detectable trend? Discuss why flatness differences may or may not appear in such a small model.

2. **Convergence behavior.** Plot loss trajectories for each configuration. Compare:

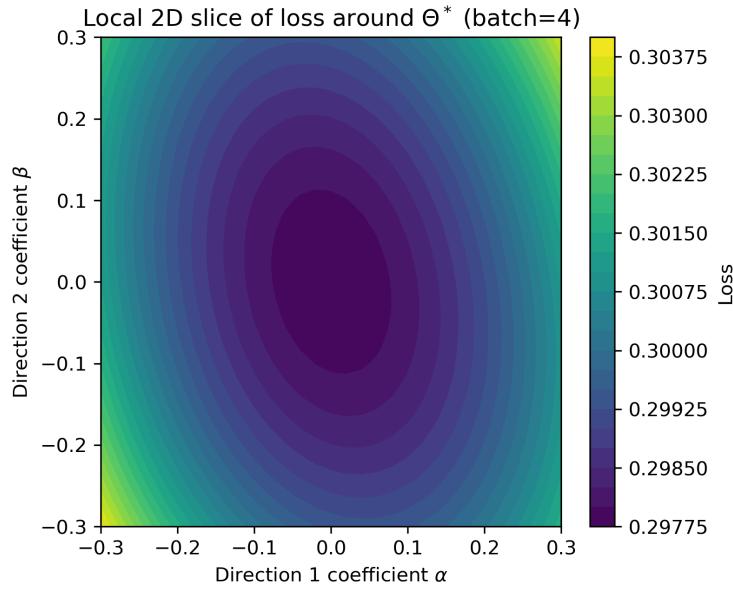


Figure 4.13: Two-dimensional slice of the loss landscape around a solution obtained with batch size 4. The basin is broad and smooth, characteristic of a flat minimum in this low-dimensional example. Despite different SGD noise levels, all batch sizes converge to geometrically similar regions in this task.

- *convergence speed,*
- *variance of the trajectory,*
- *frequency and magnitude of spikes.*

How does SGD noise influence the qualitative shape of the optimization path?

3. **Local geometry via contour slices.** For several trained solutions (different batch sizes or learning rates), generate 2D loss slices. Compare their shapes and relate them to Hessian spectra.
4. **Effect of model size.** Increase the number of hidden units. Does a higher-dimensional parameter space reveal clearer curvature differences between batch sizes? At what model size do you begin to see distinct spectral signatures?

Support your conclusions with:

- *Hessian eigenvalue plots,*
- *loss trajectories,*
- *2D loss contour plots.*

4.3.2 Dynamic Selection of Low-Dimensional Manifolds in Deep Networks

A recurring theme in modern deep learning is that, despite operating in extremely high-dimensional parameter and activation spaces, neural networks *implicitly restrict their computations to low-dimensional manifolds*. Empirical and theoretical studies [25] suggest that during training, networks organize data into structured, low-rank representations, effectively using only a small subset of the available degrees of freedom. This emergent reduction of intrinsic dimensionality is deeply related to how neural networks generalize.

These ideas connect naturally to concepts introduced earlier in the book:

- Chapter 1 introduced PCA and SVD as tools for identifying dominant directions of variation.
- Section 4.1.4 showed how CNNs extract increasingly abstract hierarchical features.
- Section 4.2.1 on ResNets and Section 4.2.2 on Neural ODEs emphasized smoothness, stability, and low-complexity transformations.

In this subsection, we use PCA to directly visualize how a trained CNN organizes MNIST digits into a compact, low-dimensional manifold inside its intermediate layers. This provides geometric intuition for hierarchical feature extraction and generalization.

Example 4.3.2 (Low-dimensional structure in CNN activation manifolds). *A small CNN trained on MNIST produces high-dimensional activations:*

- *the first convolutional layer has $16 \times 28 \times 28$ features,*
- *the second convolutional layer has $32 \times 14 \times 14$ features,*
- *the first fully connected layer has 128 features.*

These spaces are far too high-dimensional to visualize directly. However, PCA (defined via SVD in Chapter 1) allows us to extract the dominant variance directions and project activations into two or three dimensions.

Using the notebook `CNN-MNIST-PCA.ipynb`, we:

- *train a CNN for several epochs,*
- *collect activations from multiple layers for a subset of test images,*
- *flatten each activation tensor into a feature vector,*
- *perform PCA, compute explained-variance curves, and visualize PC1–PC2 scatter plots.*

*Left panel of Fig. 4.14 shows the PCA projection for the **first convolutional layer**. Even at this early stage, the network partially separates digits into clusters, indicating that the CNN has already started constructing a task-dependent embedding.*

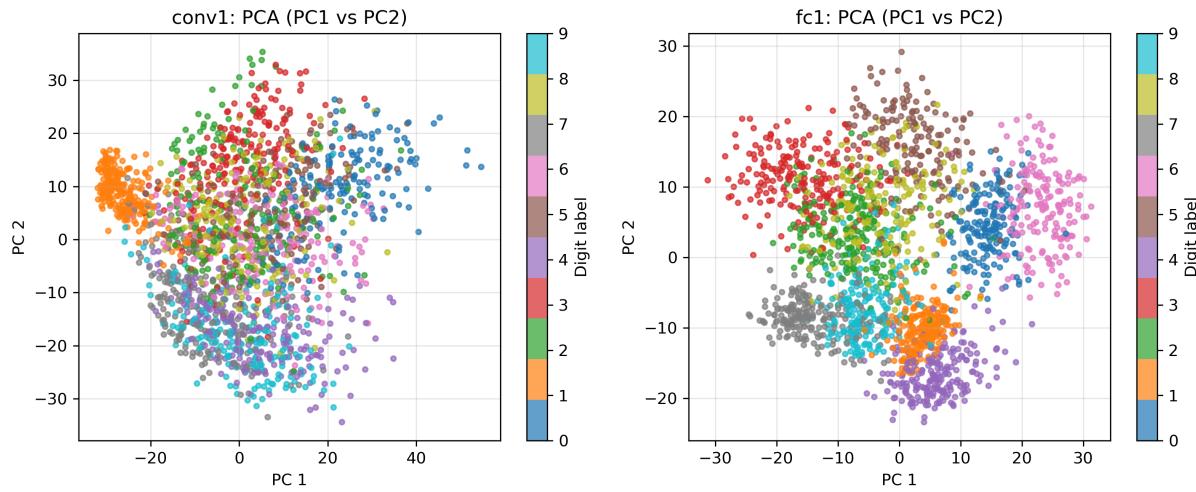


Figure 4.14: **PCA of activations in the first convolution (left) and last hidden (right) layers.** Left – first layer: Even early in the network, the CNN organizes digits into a moderately structured manifold. Points represent MNIST test images projected onto the first two principal components and are colored by digit label. Right – last (hidden) layer: Digit classes now form tight, well-separated geometric clusters in the top two principal components. This illustrates the emergence of a low-dimensional manifold encoding digit identity. Figure generated by `CNN-MNIST-PCA.ipynb`.

Fig. 4.15 shows explained-variance curves for three layers. The number of principal components needed to capture most of the variance decreases with depth—empirical evidence of representation compression.

*Finally, right panel of Fig. 4.14 shows PCA for the **fully connected layer**. The clusters become significantly tighter and more separable, reflecting the network’s progression from pixel-level detail to abstract digit identity.*

Why PCA exposes hidden geometry in neural networks

Deep networks tend to compress information into a small number of dominant directions in activation space. PCA reveals this compression: if a handful of components explain most of the variance, then activations lie near a **low-dimensional manifold**. Crucially, this structure is not hand-designed—it emerges dynamically during training as part of deep learning’s implicit regularization.

Exercise 4.3.2 (Exploring learned low-dimensional manifolds). *Using `CNN-MNIST-PCA.ipynb`, investigate how deep networks compress and structure information.*

1. **Layer-wise PCA.** Compute PCA for each of the layers shown above. How does the intrinsic dimensionality evolve with depth?
2. **Effect of nonlinearities.** Compare PCA before and after ReLU in the convolutional layers. How does the activation function reshape the manifold?

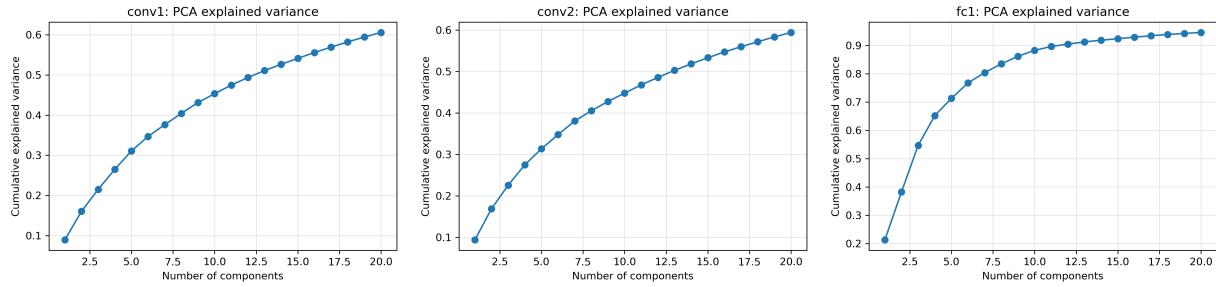


Figure 4.15: **Explained-variance curves for PCA across layers.** Deeper layers exhibit earlier saturation of variance with the number of components, indicating strong dimensionality compression. Only a small number of principal components are needed to capture most of the structure in the activations.

3. **Class geometry.** Which digits separate cleanly in the PCA plane? Which overlap? Relate this to difficulty of classification.
4. **Compression vs. memorization.** Train on a reduced dataset (e.g., 1 000 images). Does the PCA spectrum become flatter, indicating less compression?
5. **Depth and abstraction.** Compare PCA scatter plots of the first convolutional layer, second convolutional layer, and fully connected layer. How does the manifold evolve from pixel space to abstract concept space?

Support your findings with:

- PCA scatter plots,
- explained-variance curves,
- layer-wise comparisons of intrinsic dimensionality.

Computational Companion Notebooks

To complement the mathematical exposition of this Section (and more generally Chapter), three optional Jupyter notebooks are provided in the Companion Notebook Collection accompanying this book. They allow the reader to explore empirically the transition from discrete architectures (CNNs, ResNets) to continuous-depth models (Neural ODEs), as well as the geometric principles described in this Section.

`ResNet9-MNIST.ipynb` and `ResNet18-MNIST.ipynb` illustrate how skip-connections modify optimization dynamics and progressively approximate continuous-time flows.

`Sharp-vs-Flat.ipynb` provides an implementation-level view of loss-landscape geometry, complementing the conceptual discussion of flat regions and low-dimensional manifolds discussed above.

These notebooks are optional and exploratory: they are designed to reinforce intuition.

4.4 Further Reading and Roadmap to the Rest of the Book

Further Reading

We conclude this chapter by noting that learning from ODE-based or more generally *operator-based* data structures has become a major research direction that extends far beyond the Neural ODE framework. Many of the ideas predate the modern deep learning era, yet they continue to shape state-of-the-art methods through the integration of domain knowledge, physics, and mathematical structure.

The earliest use of **artificial neural networks for solving differential equations** appeared in [26], where NN-based trial solutions were used to satisfy boundary conditions and approximate ODE/PDE solutions. This line of work was revitalized two decades later by Raissi, Perdikaris, and Karniadakis [27] under the name **Physics-Informed Neural Networks (PINNs)**, which incorporate differential operators directly into the loss function. PINNs exploit the governing equations to regularize learning, often improving sample efficiency and enabling limited extrapolation.

Another early direction involves **equation-free modeling** and data-driven coarse-graining [28], in which high-fidelity simulators are used only to estimate local dynamical behavior. This allowed researchers to learn reduced-order dynamics without ever writing down the full equations.

Symbolic approaches have also contributed to data-driven discovery of dynamics. **Symbolic regression** methods [29] search for compact analytical expressions explaining observed trajectories, while the **Sparse Identification of Nonlinear Dynamics (SINDy)** framework [30] uses sparse regression to recover interpretable differential equations from time-series data. These approaches demonstrate how incorporating sparsity and interpretability priors can significantly reduce the hypothesis space.

A more recent family of methods — **neural operators** — aims to learn *mappings between infinite-dimensional function spaces*, such as

$$\mathcal{G} : f(x) \mapsto u(x),$$

where u solves a PDE with input field f . Unlike PINNs, which approximate *solutions*, neural operators approximate *solution operators*, enabling rapid evaluation for many different inputs. Two widely used architectures are:

- **DeepONet** [31], based on the universal approximation theorem for operators, which decomposes the mapping into “branch” and “trunk” networks and can learn nonlinear operators from data.
- **Fourier Neural Operators (FNO)** [32], which learn integral operators using convolution kernels represented in the Fourier domain. FNOs scale extremely well and have become a foundation for scientific machine learning in CFD, climate modeling, materials science, and more.

Neural operators represent a natural extension of Neural ODEs: they learn not just trajectories, but entire *families* of dynamical solutions. They provide a powerful interface between

scientific computing and generative AI, while retaining mathematical structure such as translation symmetries and convolutional integral kernels.

Together, these approaches illustrate a broader principle: **learning improves significantly when augmented with physics, structure, sparsity, or operator priors**. Standard neural networks excel at interpolation within the observed domain but often fail to extrapolate meaningfully. Methods such as Neural ODEs, PINNs, SINDy, symbolic regression, and neural operators help overcome this limitation by embedding additional structure into the optimization problem, providing bridges between black-box machine learning and interpretable, equations-aware modeling.

Software Tools

For state-of-the-art, research-grade software that leverages neural networks for ODEs, PDEs, and optimization problems, we recommend:

- **Julia:** The SciML (Scientific Machine Learning) ecosystem, led by Chris Rackauckas. It includes `DifferentialEquations.jl`, `DiffEqFlux.jl`, neural operators, adjoint sensitivity tools, and highly optimized PDE solvers.
- **Python:** The NeuroMANCER library developed at PNNL (Jan Drgona et al.), which provides a unified differentiable programming framework for control, optimization, PINNs, and operator learning.

From Geometry and Dynamics to Learning Systems

Chapter 4 traced a unifying geometric and dynamical perspective on modern deep learning. We saw that SGD behaves not merely as an optimization routine but as a *stochastic dynamical system* whose trajectories tend toward wide, stable valleys of the loss landscape; that architectures such as CNNs, ResNets, and Neural ODEs implicitly *steer computations onto low-dimensional manifolds*; and that deep networks perform a kind of *structured dimensionality reduction* as information flows across layers.

Across all examples—loss-landscape exploration, skip connections as discretized ODE solvers, Neural ODEs as continuous-depth models, and PCA analyses of CNN activations—the core message is that **deep learning works because learning dynamics and architecture jointly impose powerful geometric biases**. These biases guide solutions toward smooth, low-complexity representations that generalize beyond the training set.

This geometric–dynamical lens prepares us for the more advanced themes of the upcoming chapters, where differential geometry, variational principles, optimal transport, and stochastic processes will become the organizing framework for understanding generative models and the mathematics of modern AI.

Roadmap: How Chapter 4 Leads into Chapters 5,6,7,8,9

The ideas developed in Chapter 4 serve as conceptual foundations for the remainder of the book. Here we briefly summarize how the main themes of this chapter — representation

learning, optimization dynamics, and continuous-depth architectures — naturally flow into the topics that follow.

From deterministic feature maps to probabilistic models (Chapter 5). In Chapter 4 we treated neural networks as deterministic maps $x \mapsto f_\theta(x)$ learned from data. To reason about *uncertainty* in predictions, random initialization, SGD noise, and generalization, we must place these constructions into a probabilistic framework. Chapter 5 develops this framework: probability spaces, random variables, multivariate Gaussians, empirical distributions, and change-of-variables. These tools underlie:

- viewing neural outputs as random variables, not just point estimates;
- interpreting stochastic gradient methods as operating on random losses;
- understanding normalizing flows as invertible neural maps between probability measures.

From low-dimensional manifolds to information and compression (Chapter 6). Section 4.3.2 demonstrated empirically that deep networks concentrate data on low-dimensional manifolds and perform strong nonlinear compression. Chapter 6 builds a quantitative language for this behavior: entropy, mutual information, KL divergence, and cross-entropy. These notions are then used to:

- formalize training losses (cross-entropy) already used for classification in Chapter 4;
- reinterpret autoencoders and U-Net as information-theoretic encoders and decoders;
- introduce the information bottleneck viewpoint on deep representations and compression.

From Neural ODEs and SGD noise to stochastic processes (Chapter 7). The ResNet and Neural ODE sections linked deep networks to time-discretized and continuous ODEs, and the SGD discussion highlighted the role of noise and flat minima. Chapter 7 turns these intuitions into stochastic-process language: Brownian motion, diffusion, Markov chains, and MCMC. This allows us to:

- treat noisy gradient dynamics as stochastic processes in parameter space;
- view score-based diffusion and denoising as time-reversed stochastic dynamics;
- connect auto-regressive models and transformers to Markov and Markov-like chains.

From feedforward networks to energy-based and graphical models (Chapter 8). Chapter 4 focused on feedforward discriminative networks trained by backpropagation. Chapter 8 broadens the picture to *energy-based* models and graphical models, where the central object is a probability distribution

$$p_\theta(x) \propto e^{-E_\theta(x)}.$$

This chapter revisits themes from Chapter 4 — representation, optimization, and decoding — in new guises:

- neural decoding of error-correcting codes extends the simple decoding example in Section 4.3.1;
- variational autoencoders (VAEs) combine neural networks with probabilistic latent-variable models, anticipating the diffusion-based generative models of Chapter 9;
- restricted Boltzmann machines and graph neural networks connect layer-wise neural computations to inference on graphs.

From architectures to a unifying generative and control perspective (Chapter 9).

Finally, Chapter 9 synthesizes the book’s main threads. The neural architectures of Chapter 4, the probabilistic tools of Chapter 5, the information-theoretic view of Chapter 6, the stochastic processes of Chapter 7, and the energy-based models of Chapter 8 come together to:

- develop score-based diffusion models and their bridge versions as continuous-depth, noise-driven analogues of deep networks;
- reinterpret GANs and VAEs as special cases or limits of diffusion-like constructions;
- connect reinforcement learning, Markov decision processes, and the path-integral diffusion (PID) framework to generative modeling and control.

Readers are encouraged to revisit the neural-network examples of Chapter 4 as they proceed: the same ideas — representation, dynamics, noise, and structure — will reappear in progressively richer mathematical forms throughout Chapters 5–9.

Chapter 5

Probability and Statistics

Probability Theory vs. Statistics

At its core, **probability theory** provides a mathematical framework for **modeling uncertainty**. It allows us to quantify randomness, describe the behavior of random variables, and analyze probability distributions that govern real-world stochastic processes. In contrast, **statistics** is concerned with analyzing data — often referred to in modern contexts as **data science** — wherein we use observed samples to infer underlying distributions, test hypotheses, and build predictive models.

In AI, probability theory forms the foundation of probabilistic models such as Bayesian networks, variational autoencoders, normalizing flows, and diffusion models. Meanwhile, statistical methods enable learning from data, optimizing model parameters, and validating results through statistical inference.

This book blends probability theory and statistics into a unified perspective, emphasizing their interplay in the context of generative AI. The structure of this chapter reflects this integration, treating both theoretical foundations and practical applications holistically.

Why Probability Matters in Deep Learning

Chapter 4 showed that modern neural networks — CNNs, ResNets, Neural ODEs and Transformers — learn powerful representations by gradually deforming data through compositions of trainable maps. Chapter 5 now shifts perspective: instead of studying transformations of *deterministic inputs*, we study transformations of *probability distributions*.

This conceptual shift from geometry of features to geometry of *densities* is what makes normalizing flows, variational inference, and diffusion models possible. Probability is therefore not an optional appendix to deep learning; it is the mathematical engine behind modern generative AI.

Uncertainty in Generative AI

Generative AI must grapple with multiple sources of uncertainty, requiring probabilistic and statistical tools to address challenges in model design, data interpretation, and computational

feasibility:

- **Model uncertainty:** Probabilistic models such as normalizing flows, hidden Markov models, and diffusion models must account for inherent randomness in data generation. Some probabilistic frameworks are application-specific, incorporating structural features such as Poisson rates (e.g., event modeling) or sparsity in covariance matrices (e.g., high-dimensional settings).
- **Data uncertainty:** Real-world training data is often incomplete, noisy, or biased. This motivates statistical tools such as empirical distributions, Kernel Density Estimation (KDE), Maximum Likelihood Estimation (MLE), and confidence intervals — a few of the many techniques introduced in this chapter.
- **Computational uncertainty:** Many probabilistic models are analytically intractable, requiring approximation schemes such as Monte Carlo sampling, variational inference, and reparameterization methods. Later chapters build these techniques into full generative AI pipelines.

Why This Chapter Matters in the Book

The preceding chapters developed the *geometric*, *architectural*, and *optimization*-based foundations of neural networks. However, generative AI ultimately requires reasoning about *probability distributions*, not just deterministic features.

Chapter 5 provides the mathematical language for:

- measuring distances between distributions (KL divergence, entropy);
- transforming distributions via smooth maps (change-of-variables);
- modeling high-dimensional uncertainty (joint distributions, covariance);
- understanding universal phenomena such as the Central Limit Theorem;
- studying extreme statistics relevant for risk, robustness, and anomaly detection.

These tools will be essential for:

- variational inference and VAEs (Chapter 6,
- information theory and compression principles in generative modeling (Chapter 7),
- stochastic processes and diffusion models (Chapter 8,
- the unifying synthesis of generative frameworks (Chapter 9).

In short: Chapter 5 is where the mathematics of uncertainty enters the generative story.

Organization of the Chapter

This chapter is structured to gradually build intuition, formal definitions, and applications:

- **Section 5.1: Primer on Probability Spaces and Random Variables** introduces probability spaces, random variables, expectations, and moments, establishing the core theoretical machinery.
- **Section 5.2: Transforming Probability Distributions** discusses change-of-variable formulas, pushforward distributions, empirical distributions, and leads directly into the introduction of normalizing flows — a key class of deep generative models.
- **Section 5.3: Multivariate Random Variables** extends probability to multiple dimensions, covering joint densities, independence, conditional distributions, and the multivariate Gaussian — the central object of high-dimensional modeling.
- **Section 5.4: From Aggregate Behavior to Rare Events.** This section develops a unified view of distributional limits. We begin with the Central Limit Theorem, which explains the emergence of Gaussian structure in sums of random variables and provides essential convergence and tail bounds. We then examine regimes where Gaussian approximations break down: the *rare-event limit*, where summing Bernoulli variables leads to the Poisson distribution and point-process models; and the *extreme-value limit*, where the behavior of maxima (rather than sums) gives rise to universal extreme-value laws. Together, these results form a toolkit for understanding fluctuations, anomalies, and worst-case behavior in modern machine learning systems.

We begin by introducing relevant foundational material from a combined applied mathematics and AI perspectives¹. For a rigorous measure-theoretic treatment, the classical text *Probability With Martingales* by David Williams [34] is recommended.

5.1 Primer for Probability Spaces & Random Variables

Probability provides the mathematical language for uncertainty. All models used in modern AI — from simple classifiers to deep generative models such as VAEs, GANs, Diffusion Models, and Transformers — manipulate or transform probability distributions in one form or another. Whether we *sample* noise, *estimate* likelihoods, or *push forward* distributions through learned maps, the underlying machinery rests on the foundations introduced in this section.

This chapter therefore begins with a brief mathematical primer: probability spaces, random variables, distributions, expectations, and transformations of variables. These ingredients will be reused throughout the chapter and will also motivate the computational notebooks that accompany the text.

¹Applied mathematics wise this chapter, along with the following chapters of the book, incorporates material from the author’s recent book *Principles and Methods of Applied Mathematics* [33].

5.1.1 Probability Spaces: The Foundation

A *probability space* formalizes randomness through a triple

$$(\Omega, \mathcal{F}, P),$$

where:

- Ω is the **sample space** — the set of all possible outcomes.
- \mathcal{F} is a **σ -algebra** of subsets of Ω whose elements are called *events*. It satisfies:
 - If $A \in \mathcal{F}$ then $A^c \in \mathcal{F}$.
 - If $A_i \in \mathcal{F}$ for $i \geq 1$, then $\bigcup_i A_i \in \mathcal{F}$.
 - Consequently, $\bigcap_i A_i \in \mathcal{F}$.
- $P : \mathcal{F} \rightarrow [0, 1]$ is a **probability measure** satisfying the Kolmogorov axioms:

$$P(\Omega) = 1, \quad P(A) \geq 0, \quad P(A \cup B) = P(A) + P(B) \text{ if } A \cap B = \emptyset.$$

Example 5.1.1 (Coin Toss). *For a single fair-coin toss:*

$$\Omega = \{\text{Head}, \text{Tail}\}, \quad \mathcal{F} = 2^\Omega = \{\emptyset, \{\text{Head}\}, \{\text{Tail}\}, \Omega\},$$

and

$$P(\text{Head}) = P(\text{Tail}) = \frac{1}{2}, \quad P(\emptyset) = 0, \quad P(\Omega) = 1.$$

A biased coin with parameter ρ simply modifies the measure:

$$P(\text{Head}) = \rho, \quad P(\text{Tail}) = 1 - \rho,$$

while leaving Ω and \mathcal{F} unchanged.

Exercise 5.1.1 (A Probability Space in Coding Theory). *Three information bits are encoded with one parity check and transmitted over a **Binary Erasure Channel (BEC)** with erasure probability ϵ .*

Construct the probability space:

1. Define the sample space Ω of all received 3-bit patterns including erasures (ϵ).
2. Define a natural σ -algebra \mathcal{F} of decoding-relevant events.
3. Construct a probability measure P consistent with BEC erasures.

Hint: *Probabilities factor over bit positions; a received bit is either correct (probability $1 - \epsilon$) or erased (probability ϵ).*

5.1.2 Random Variables

A **Random Variable** (RV) is a measurable function

$$X : \Omega \rightarrow \mathbb{R},$$

assigning a numerical value to each outcome.

Example 5.1.2 (Rolling a Die). *For $\Omega = \{1, \dots, 6\}$, the natural random variable is $X(\omega) = \omega$. Here X is discrete, supported on $\{1, \dots, 6\}$.*

Random variables may be:

- **Discrete:** e.g., outcomes of a die, coin flips, syndrome bits in decoding.
- **Continuous:** e.g., Gaussian noise injected into a neural network layer.

Notation. Uppercase letters denote random variables (X, Y), lowercase letters denote realizations (x, y). We write

$$P_X(x) = P(X = x)$$

for Probability Mass Function (PMF) of discrete RVs and

$$p_X(x)$$

for Probability Density Functions (PDFs) of continuous RVs.

The **Cumulative Distribution Function** (CDF) of X is

$$F_X(x) = P(X \leq x).$$

For discrete RVs it is a step function; for continuous RVs it is differentiable with $p_X(x) = F'_X(x)$.

Sampling Notation. We write

$$X \sim P_X$$

to indicate that the random variable X is generated according to the probability distribution P_X . To *sample* from a distribution means to produce one or more numerical realizations of X whose empirical behavior reflects the underlying law P_X . In practice, sampling corresponds to running a (possibly deterministic or randomized) algorithm that outputs values which are distributed according to the specified probability model.

For example, we may write

$$X \sim \text{Bernoulli}(\rho), \quad X \sim \text{Poisson}(\lambda), \quad X \sim \mathcal{N}(\mu, \sigma^2),$$

to denote that draws of X take the value 1 with probability ρ , count rare events with mean λ , or follow a Gaussian bell curve with mean μ and variance σ^2 , respectively.

Throughout the rest of the chapter (and later in the book), “generating samples” refers to using computational procedures (e.g., NumPy’s random module, inversion sampling, rejection sampling, or later, Markov-chain or diffusion-based samplers) to obtain independent realizations X_1, X_2, \dots whose aggregate statistics approximate those of the theoretical distribution P_X .

5.1.3 Expectation and Moments

The **expectation** of X is

$$\mathbb{E}[X] = \begin{cases} \sum_x x P_X(x), & \text{discrete,} \\ \int_{-\infty}^{\infty} x p_X(x) dx, & \text{continuous.} \end{cases}$$

More generally, the n -th moment is $\mathbb{E}[X^n]$, and the variance is

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

Example 5.1.3 (Moment Generating Function). *The Moment Generating Function (MGF)*

$$M_X(t) = \mathbb{E}[e^{tX}]$$

encodes all moments through differentiation:

$$\mathbb{E}[X^n] = \frac{d^n}{dt^n} M_X(t) \Big|_{t=0}.$$

Exercise 5.1.2 (Poisson Moments via MGF). *For $X \sim \text{Poisson}(\lambda)$:*

1. Compute $M_X(t) = \mathbb{E}[e^{tX}]$.
2. Use differentiation to extract $\mathbb{E}[X]$ and $\mathbb{E}[X^2]$.
3. Show that $\text{Var}(X) = \lambda$.

Exercise 5.1.3 (Gaussian Integral and a Useful Identity). (a) Evaluate the Gaussian integral

$$I = \int_{-\infty}^{\infty} e^{-t^2} dt$$

by computing I^2 in polar coordinates.

(b) For $X \sim \mathcal{N}(\mu, \sigma^2)$, use (a) to show

$$\mathbb{E}[X^2] = \sigma^2 + \mu^2.$$

5.1.4 Data–Driven Probability: Empirical Distributions

In modern machine learning, distributions rarely appear in closed form. Instead, we often have *samples* and must construct empirical approximations to probability laws. The accompanying notebook `Empirical-Distributions-1D.ipynb` demonstrates the key ideas through several one-dimensional examples.

Example 5.1.4 (Empirical PMF and CDF from Samples). *Given samples $x_1, \dots, x_N \sim P_X$, one builds:*

- an empirical PMF gives the probability of a discrete variable equaling a specific value simply via normalized counts,

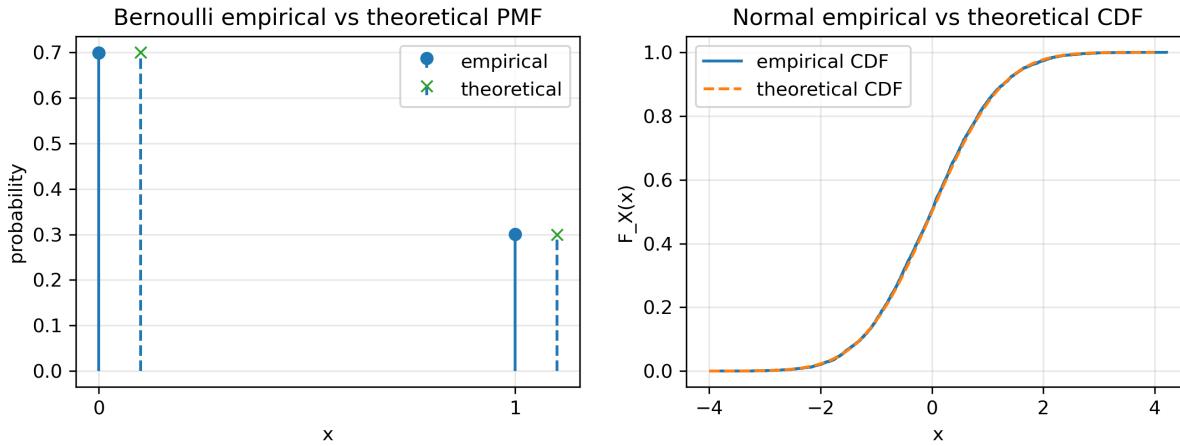


Figure 5.1: Empirical vs. theoretical (left) PMF for a Bernoulli random variable and (right) CDF for a Gaussian random variable – the staircase curve is the empirical CDF; the smooth curve is the analytic CDF, as generated by `Empirical-Distributions-1D.ipynb`.

- an empirical Cumulative Distribution Function (CDF) is

$$\hat{F}_N(x) = \frac{1}{N} \sum_{i=1}^N \mathbf{1}\{x_i \leq x\}.$$

Left panel of Fig. 5.1 shows the empirical and theoretical PMFs for a Bernoulli random variable. Right panel of Fig. 5.1 shows the empirical and theoretical CDFs for a Gaussian random variable, illustrating how the empirical CDF converges to the true CDF as N grows.

Exercise 5.1.4 (Empirical Convergence and the Law of Large Numbers). Using the notebook `Empirical-Distributions-1D.ipynb`:

1. For Bernoulli, Poisson, and Normal distributions, generate Independent Identically Distributed – i.i.d. – samples and compute the running sample mean

$$\hat{m}_N = \frac{1}{N} \sum_{i=1}^N X_i$$

as a function of N .

2. Plot \hat{m}_N versus N on a log scale and compare with the true mean, as in Fig. 5.2.
3. Explain how these plots illustrate the Law of Large Numbers.

5.1.5 Transformations of Random Variables

Later in this chapter we will study *change of variables*, *pushforward distributions*, and *normalizing flows*. The notebook `Transformations-1D.ipynb` illustrates these topics computationally.

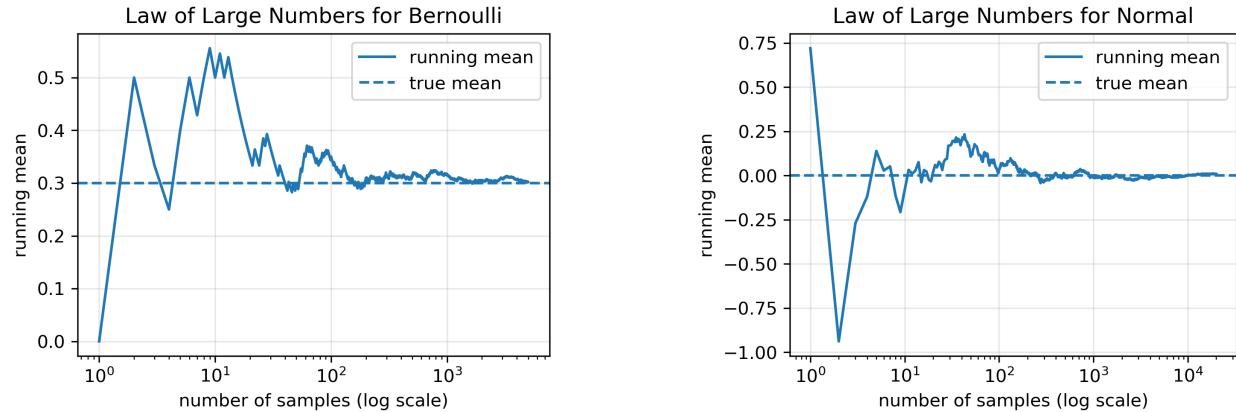


Figure 5.2: Law of Large Numbers in action: running sample mean for Bernoulli (left) and Normal (right) distributions converging toward the true mean. Generated by `Empirical-Distributions-1D.ipynb`.

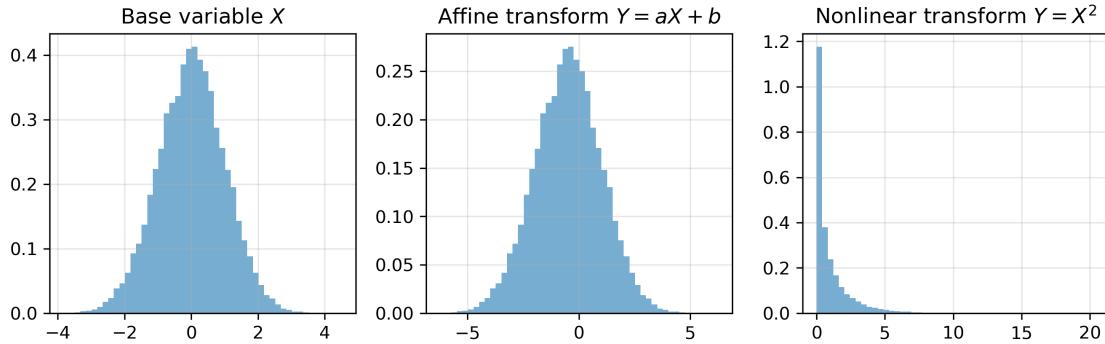


Figure 5.3: Multi-panel view of one-dimensional transformations $Y = g(X)$: base variable X , affine transform $Y = aX + b$, and nonlinear squaring $Y = X^2$. Generated by `Transformations-1D.ipynb`.

Example 5.1.5 (Deterministic Transformations $Y = g(X)$). *If X has PDF p_X and $Y = g(X)$ is a smooth, monotone transformation, then*

$$p_Y(y) = p_X(g^{-1}(y)) \left| \frac{d}{dy} g^{-1}(y) \right|.$$

The notebook implements and visualizes three cases:

- *affine map $Y = aX + b$,*
- *nonlinear squaring $Y = X^2$,*
- *saturating nonlinearity $Y = \tanh(X)$.*

Fig. 5.3 shows a multi-panel summary of these transformations, comparing empirical histograms with analytic densities where available.

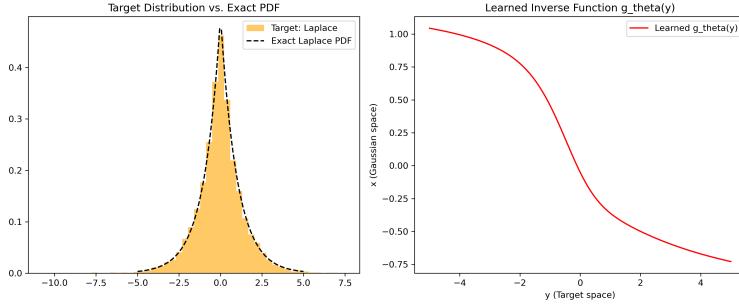


Figure 5.4: A simple one-dimensional normalizing flow: a learned inverse map g_θ pushes samples from a non-Gaussian target back to a standard Gaussian base. Generated by `Normalizing-Flow-1D.ipynb`.

Exercise 5.1.5 (Comparing Nonlinear Transformations). *Using `Transformations-1D.ipynb`:*

1. *For $X \sim \mathcal{N}(0, 1)$, numerically compare the empirical distribution of $Y = X^2$ with the analytic density obtained via change of variables.*
2. *Repeat for $Y = \tanh(X)$ with different input variances, and discuss how saturation affects the output distribution.*
3. *Comment on how local stretching/compression of the map g is reflected in the shape of p_Y .*

5.1.6 A First Normalizing Flow in 1D

Modern generative models frequently construct complex target distributions by transforming simple base distributions through *invertible* maps (flows). The notebook `Normalizing-Flow-1D.ipynb` implements a simple instance of this idea in one dimension.

Example 5.1.6 (Learning an Inverse Map with a Normalizing Flow). *Let Y be a non-Gaussian target variable (e.g., Laplace), and let $Z \sim \mathcal{N}(0, 1)$ be a standard Gaussian. A one-dimensional normalizing flow learns an invertible map f_θ such that $f_\theta(Z)$ has the same law as Y , or equivalently an inverse map g_θ that sends Y back to Z .*

The notebook trains g_θ by minimizing a KL-based objective derived from the change-of-variables formula. Fig. 5.4 shows the learned inverse transform and its effect on the distribution of the transformed samples.

Exercise 5.1.6 (Extending the 1D Flow). *Modify `Normalizing-Flow-1D.ipynb` to:*

1. *Change the target distribution (e.g., heavier tails or multimodal).*
2. *Increase the depth or width of the neural network implementing g_θ .*
3. *Compare the quality of the learned inverse map and the match to the Gaussian base for different architectures.*

Relate your observations to the theoretical change-of-variables formula introduced earlier.

5.2 Transforming Probability Distributions

In the previous section, *Primer for Probability Spaces & Random Variables*, we introduced probability spaces, random variables, their distributions, and empirical approximations. Here we build on that material and ask:

How do probability distributions change when we transform, approximate, or learn them?

We focus on three tightly connected themes:

- the **change-of-variables** formula for deterministic transformations $Y = f(X)$,
- **empirical** and **smoothed** (regularized) distributions,
- **normalizing flows**, which learn invertible transformations between simple and complex distributions.

Each topic is presented in the now familiar pattern: material → example → exercise.

5.2.1 Change of Variables in One Dimension

Let X be a continuous random variable with PDF $p_X(x)$, and let $Y = f(X)$ be a deterministic transformation. From the viewpoint of Section 5.1, the transformation f induces a new random variable Y on the same probability space, and we would like to express the PDF p_Y of Y in terms of p_X and f .

If f is differentiable and possibly *not* one-to-one, the **change-of-variables formula** in one dimension reads

$$p_Y(y) = \sum_{\text{all pre-images } x: f(x)=y} p_X(x) \left| \frac{dx}{dy} \right|. \quad (5.1)$$

The sum runs over all solutions x of the equation $f(x) = y$. When f is strictly monotone, there is a single pre-image and the sum reduces to a single term. When f is not injective (e.g., $f(x) = x^2$), multiple pre-images contribute.

Example 5.2.1. Squaring a Standard Normal: $Y = X^2$ Let

$$X \sim \mathcal{N}(0, 1), \quad p_X(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}.$$

Consider the nonlinear transformation

$$Y = X^2.$$

For $y > 0$ the equation $y = x^2$ has two solutions $x = \pm\sqrt{y}$, and

$$\frac{dx}{dy} = \pm \frac{1}{2\sqrt{y}}.$$

Applying (5.1):

$$\begin{aligned} p_Y(y) &= p_X(\sqrt{y}) \left| \frac{d}{dy} \sqrt{y} \right| + p_X(-\sqrt{y}) \left| \frac{d}{dy} (-\sqrt{y}) \right| \\ &= \frac{1}{\sqrt{2\pi}} e^{-y/2} \frac{1}{2\sqrt{y}} + \frac{1}{\sqrt{2\pi}} e^{-y/2} \frac{1}{2\sqrt{y}} \\ &= \frac{1}{\sqrt{2\pi y}} e^{-y/2}, \quad y > 0. \end{aligned}$$

Thus $Y = X^2$ has a chi-square distribution with one degree of freedom. Its tail decays exponentially in y , so for large y it resembles an exponential distribution. This is the analytic counterpart of the numerical experiment with $Y = X^2$ in the notebook `Transformations-1D.ipynb` which we already used in the preceding Section.

Exercise 5.2.1. Other Transformations of a Gaussian Let $X \sim \mathcal{N}(0, 1)$.

1. Define $Y = e^X$. Use (5.1) to derive the PDF of Y and identify its distribution.
2. Define $Z = |X|$. Compute the PDF of Z using the two pre-images $\pm z$.
3. Compare your analytic PDFs with numerical histograms produced in `Transformations-1D.ipynb`. Comment on how well the empirical distributions match the analytic densities.

5.2.2 From Spiky to Smooth: Kernel Density Estimation

In many applications, we do not know p_X explicitly. Instead, we observe N i.i.d. samples

$$x_1, \dots, x_N \sim P_X.$$

Then – and re-phrasing what we discussed in the previous Section – the **empirical distribution** is the discrete measure

$$P_{\text{emp}}(x) = \frac{1}{N} \sum_{i=1}^N \delta(x - x_i), \tag{5.2}$$

where $\delta(\cdot)$ is the Dirac delta-function ²

While (5.2) is statistically sound, it is extremely irregular: it consists of spikes at the observed samples and is not directly suitable as a smooth model. A standard way to obtain a smooth

²The **Dirac δ-function** is not a function in the classical sense but a *distribution* (or *generalized function*) acting on test functions. Formally, it is defined as the linear functional satisfying

$$\int_{-\infty}^{\infty} \delta(x - x_0) \varphi(x) dx = \varphi(x_0) \quad \text{for all smooth test functions } \varphi.$$

Intuitively, $\delta(x - x_0)$ is “infinitely peaked” at x_0 with total mass 1, and zero elsewhere. The concept was introduced by P. A. M. Dirac in the 1930s in the context of quantum mechanics, where such objects naturally arise when describing point sources, impulses, or eigen-states of continuous spectra. Rigorous mathematical foundations were later provided through the theory of distributions developed by Schwartz.

approximation is **Kernel Density Estimation** (KDE), which replaces each delta with a smooth kernel:

$$P_{\text{KDE}}(x) = \frac{1}{N} \sum_{i=1}^N K_h(x - x_i), \quad (5.3)$$

where $K_h(\cdot)$ is a smoothing kernel with bandwidth $h > 0$. A common choice is the Gaussian kernel

$$K_h(x) = \frac{1}{\sqrt{2\pi h^2}} \exp\left(-\frac{x^2}{2h^2}\right).$$

Small h yields a nearly spiky estimate (low bias, high variance), while large h produces a very smooth but potentially biased estimate.

Regularization Viewpoint. The transition from P_{emp} to P_{KDE} can be interpreted as *regularization*: instead of fitting every sample exactly with a delta spike, we enforce additional structure (smoothness) in our estimate. Similar ideas appear when we learn parametric transformations f_θ or g_θ between distributions and penalize overly complex maps.

Example 5.2.2. Regularized Transformation Fit Consider a parametric map $f_\theta : \mathbb{R} \rightarrow \mathbb{R}$ (e.g., a small neural network) that takes samples from X and aims to match a target distribution P_Y (known only through samples). A typical regularized loss has the form

$$\mathcal{L}(\theta) = \mathbb{E}_{X \sim P_{\text{emp}}} [d(f_\theta(X), Y)] + \lambda R(\theta),$$

where:

- $d(\cdot, \cdot)$ is a data-fit or discrepancy term between $f_\theta(X)$ and target samples Y ;
- $R(\theta)$ is a regularizer (e.g., weight decay, smoothness penalty) that discourages overly complex transformations;
- $\lambda > 0$ balances fit and regularization.

This structure foreshadows what we will do in normalizing flows, where f_θ or g_θ must be invertible and sufficiently smooth.

Exercise 5.2.2. Empirical vs. Smoothed Estimates Using *Empirical-Distributions-1D.ipynb* as a template:

1. Draw $N = 5000$ samples from a distribution with exponential tails, e.g.,

$$P_X(x) \propto e^{-|x|}, \quad x \in \mathbb{R}.$$

2. Plot the empirical histogram and overlay the true PDF (up to normalization).
3. Implement a Gaussian-kernel KDE (5.3) with several bandwidths h . Compare the KDE curves to the empirical histogram and to the target PDF.
4. Discuss the bias-variance trade-off as h varies. Which values of h give the most visually and quantitatively reasonable approximation?

5.2.3 From Gaussian to Arbitrary Distributions: Normalizing Flows

Example 5.2.1 showed how a *fixed* nonlinear transformation can turn a simple Gaussian into a non-Gaussian distribution. In many AI applications, we want the reverse: given samples from a complex target distribution, we want to *learn* a map that relates it to a simple base distribution such as a standard Normal. This is the idea behind **normalizing flows**.

We typically:

- choose a simple base variable $Z \sim \mathcal{N}(0, 1)$,
- learn an invertible map f_θ such that $Y = f_\theta(Z)$ matches a target distribution,
- or equivalently learn an inverse map $g_\theta = f_\theta^{-1}$ that sends target samples Y back to the base Z .

Change of Variables for Flows. If f_θ (or g_θ) is differentiable and invertible, the one-dimensional change-of-variables formula gives

$$p_Y(y) = p_Z(g_\theta(y)) |\partial_y g_\theta(y)|, \quad g_\theta = f_\theta^{-1}.$$

In many flow models we minimize a Kullback–Leibler divergence between the transformed distribution and a reference (see also Section 6.2.3), leading to loss functions of the form

$$\mathcal{L}(\theta) = \mathbb{E}_{Y \sim P_Y} \left[\underbrace{\frac{1}{2} g_\theta(Y)^2}_{\text{Gaussian energy}} + \underbrace{\log |\partial_y g_\theta(Y)|}_{\text{Jacobian term}} \right] + R(\theta), \quad (5.4)$$

where $R(\theta)$ is a regularizer.

Example 5.2.3. Gaussian–Laplace Matching via an Inverse Flow Consider a Laplace target

$$Y \sim \text{Laplace}(0, 1), \quad p_Y(y) = \frac{1}{2} e^{-|y|},$$

and a base $Z \sim \mathcal{N}(0, 1)$. We seek an inverse map g_θ such that $g_\theta(Y) \approx Z$ in distribution.

- **Analytic inverse via CDFs.** The CDF of the Laplace distribution is

$$F_Y(y) = \begin{cases} \frac{1}{2} e^y, & y < 0, \\ 1 - \frac{1}{2} e^{-y}, & y \geq 0, \end{cases}$$

while the CDF of Z is $\Phi(z)$. Matching CDFs, $F_Y(y) = \Phi(z)$, gives an exact inverse map

$$g(y) = \Phi^{-1}(F_Y(y)).$$

- **Learned inverse.** In `Normalizing-Flow-1D.ipynb`, we parametrize g_θ by a small neural network and minimize (5.4) using samples from $Y \sim \text{Laplace}(0, 1)$. A successful training run produces g_θ that closely approximates g , and transformed samples $g_\theta(Y)$ that are nearly standard Normal.

This simple one-dimensional flow illustrates how learned transformations connect the measure-theoretic change-of-variables formula of Section 5.1 to practical generative modeling.

Exercise 5.2.3. Experimenting with the 1D Normalizing Flow Using *Normalizing-Flow-1D.ipynb*

1. Train an inverse flow g_θ for the Laplace target as in the example above. Visualize:
 - the map $y \mapsto g_\theta(y)$,
 - histograms of $g_\theta(Y)$ compared to the standard Normal PDF.
2. Compare $g_\theta(y)$ to the analytic inverse $g(y) = \Phi^{-1}(F_Y(y))$. How close are they, and where do discrepancies appear?
3. Change the target distribution to a heavier-tailed law, e.g.,

$$p_Y(y) \propto (1 + y^2)^{-\alpha}, \quad \alpha > 1,$$

and repeat the experiment. Comment on how tail behavior affects training and the learned map.

5.2.4 Normalizing Flows and Optimal Transport (Preview)

In the standard normalizing-flow setup, an invertible map is trained so that its pushforward distribution matches the data distribution. In Section 5.2.3 we saw how likelihood-based training (or, equivalently, minimizing a KL divergence) ensures that the *final* distribution produced by the flow is correct. However, nothing in that formulation explicitly controls *how far* individual points move under the learned transformation.

This raises the following guiding question:

Can we encourage flows that move samples “as little as possible” while still matching the target distribution?

This question connects normalizing flows to the mathematical field of **Optimal Transport** (OT), which studies the most efficient way to move mass between two probability distributions. Although we will postpone the formal definition of transport costs and Wasserstein distances until Chapter 6, it is useful to preview how OT-style ideas arise naturally in flow-based generative modeling.

Transport-Based Viewpoint (Conceptual). Given a source distribution P_X and a target distribution P_Y , one may ask for a map T that pushes P_X onto P_Y while minimizing the average “movement cost” $c(X, T(X))$. In OT, a common choice is a quadratic cost $c(x, y) = \|x - y\|^2$, favoring transport plans that displace points minimally. A likelihood-trained normalizing flow seeks a correct pushforward, but does not address the magnitude of pointwise displacements.

Regularizing Flows Toward Minimal Transport. One way to introduce OT flavor into normalizing flows is to augment the standard loss with a *transport penalty*:

$$\mathcal{L}_{\text{OT}}(\theta) = \mathbb{E}_{X \sim P_X}[-\log p_Y(T_\theta(X))] + \lambda \mathbb{E}_{X \sim P_X}[\|X - T_\theta(X)\|^2].$$

The first term is the likelihood objective familiar from Section 5.2.3 (and fully developed via KL divergence in Chapter 6). The second term favors transformations that resemble those arising in Optimal Transport, and encourages minimal displacement among plausible maps.

Flow Matching and Continuous-Time Limits (Preview). Recent generative-modeling methods (e.g., *flow matching*) build continuous-time interpolations between P_X and P_Y by training time-dependent vector fields. These approaches blur the line between normalizing flows and diffusion-based models, and often exhibit strong connections to transport geometry. We will revisit these ideas in Chapter 9, where score-based diffusion models, probability–flow ODEs, and flow matching appear as continuous-time analogues of the invertible transformations studied in this chapter.

How This Fits into Later Chapters.

- In Chapter 6 we develop a systematic framework for *comparing distributions*, including KL divergence and, later, *Wasserstein distances*, which measure the minimal transport cost between distributions. These distances formalize the intuitive “minimal movement” principle hinted at above.
- In Chapter 9 we return to the geometry of probability transformations through score-based diffusion models and probability–flow ODEs. There we encounter *flow matching*, an OT-inspired method that constructs generative models by matching infinitesimal transports rather than static maps.
- The empirical and parametric tools introduced in Section 5.2 reappear there as building blocks for these continuous-time generative frameworks.

5.3 Multivariate Random Variables

The previous sections developed probability theory in the univariate setting: probability spaces, random variables, empirical distributions, change-of-variables, and simple transformations such as those used in normalizing flows. However, most real-world systems — from generative AI models to physical networks, financial portfolios, and sensor arrays — involve *multiple* interacting random variables. This section extends the framework to the multivariate case.

We introduce:

- Random vectors and their joint and marginal distributions;
- The notion of independence for collections of random variables;
- Algebraic and linear operations on multivariate random variables;

- The structure and special closure properties of multivariate Gaussians;
- Empirical multivariate statistics and sampling in higher dimensions, including non-Gaussian examples.

The multivariate setting is the foundation for later chapters: in Chapter 6 we compare multi-dimensional distributions using KL divergence and Wasserstein distance, and in Chapters 7–9 we study stochastic processes, diffusion models, and probability flows that operate on high-dimensional spaces.

5.3.1 Random Vectors, Joint Distributions, and Independence

A **random vector** is a collection of random variables defined on a common probability space. Formally, a d -dimensional random vector is

$$X = (X_1, \dots, X_d)^\top : \Omega \rightarrow \mathbb{R}^d.$$

Its law is a probability distribution on \mathbb{R}^d .

Joint Distributions. For discrete-valued X , the *joint PMF* is

$$P_X(x_1, \dots, x_d) = P(X_1 = x_1, \dots, X_d = x_d).$$

For continuous-valued X , the *joint PDF* $p_X(x)$ satisfies, for any rectangle $A_1 \times \dots \times A_d$,

$$P(X_1 \in A_1, \dots, X_d \in A_d) = \int_{A_1} \dots \int_{A_d} p_X(x_1, \dots, x_d) dx_d \dots dx_1.$$

Marginals. The distribution of a single coordinate X_i is obtained by summing or integrating out the others. For example, for a continuous X ,

$$p_{X_i}(x_i) = \int_{\mathbb{R}^{d-1}} p_X(x_1, \dots, x_d) dx_1 \dots dx_{i-1} dx_{i+1} \dots dx_d.$$

More generally, any subset of coordinates forms a lower-dimensional random vector with its own joint law.

Independence. Random variables X_1, \dots, X_d are *mutually independent* if their joint distribution factorizes:

$$P_X(x_1, \dots, x_d) = \prod_{i=1}^d P_{X_i}(x_i)$$

in the discrete case, or

$$p_X(x_1, \dots, x_d) = \prod_{i=1}^d p_{X_i}(x_i)$$

in the continuous case. Intuitively, knowing any subset of coordinates does not change our beliefs about the others.

Mean Vector and Covariance Matrix. The *mean* (or expectation) of a random vector X is

$$\mu = \mathbb{E}[X] = \begin{bmatrix} \mathbb{E}[X_1] \\ \vdots \\ \mathbb{E}[X_d] \end{bmatrix}.$$

The *covariance matrix* Σ is the $d \times d$ matrix with entries

$$\Sigma_{ij} = \text{Cov}(X_i, X_j) = \mathbb{E}[(X_i - \mu_i)(X_j - \mu_j)].$$

Diagonal entries are variances; off-diagonal entries measure linear dependence between coordinates.

Example 5.3.1. Joint, Marginal, and Independence in 2D Let (X_1, X_2) take values in $\{0, 1\}^2$ with joint probabilities

$$P_X(0, 0) = \frac{1}{4}, \quad P_X(0, 1) = \frac{1}{4}, \quad P_X(1, 0) = \frac{1}{4}, \quad P_X(1, 1) = \frac{1}{4}.$$

Then

$$P_{X_1}(0) = P_{X_1}(1) = \frac{1}{2}, \quad P_{X_2}(0) = P_{X_2}(1) = \frac{1}{2},$$

and $P_X(x_1, x_2) = P_{X_1}(x_1)P_{X_2}(x_2)$, so X_1 and X_2 are independent.

If instead $P_X(0, 0) = P_X(1, 1) = \frac{1}{2}$ and $P_X(0, 1) = P_X(1, 0) = 0$, the marginals are still Bernoulli with parameter $\frac{1}{2}$, but $P_X(x_1, x_2) \neq P_{X_1}(x_1)P_{X_2}(x_2)$. Here X_1 and X_2 are perfectly correlated: $X_1 = X_2$ almost surely.

Exercise 5.3.1. Construct an example of a 3-dimensional random vector (X_1, X_2, X_3) such that:

- all pairwise marginals (X_i, X_j) have the same distribution,
- but the three variables are not mutually independent.

Describe the joint PMF explicitly and check the factorization property.

5.3.2 Algebraic and Linear Operations on Random Vectors

Algebraic operations on random vectors are defined coordinate-wise. For example, for two random vectors $X, Y \in \mathbb{R}^d$,

$$Z = X + Y$$

is the random vector with coordinates $Z_i = X_i + Y_i$. Scalar multiples and more general linear combinations are defined similarly.

Expectations and Covariances under Linear Maps. If $Y = AX + b$ for a deterministic matrix A and vector b , then

$$\mathbb{E}[Y] = A\mathbb{E}[X] + b, \quad \text{Cov}(Y) = A \text{Cov}(X) A^\top.$$

These identities hold for *any* distribution of X (not just Gaussian).

The distributional shape, however, is generally *not* preserved. Summing or transforming non-Gaussian variables typically yields a different functional form. Multivariate Gaussians are special because many such operations stay within the same family; this is one reason they are so ubiquitous in probabilistic modeling.

Example 5.3.2. Sum and Scaling of Independent Gaussians Let $X_1, X_2 \sim \mathcal{N}(0, 1)$ be independent.

Sum. Define $Z = X_1 + X_2$. Using convolution of independent densities,

$$p_Z(z) = \int_{\mathbb{R}} p_{X_1}(x) p_{X_2}(z - x) dx = \frac{1}{\sqrt{4\pi}} e^{-z^2/4},$$

so $Z \sim \mathcal{N}(0, 2)$. The sum of independent Gaussians is again Gaussian.

Scaling. For a constant a , define $W = aX_1$. A 1D change of variables gives

$$p_W(w) = \frac{1}{|a|} p_{X_1}\left(\frac{w}{a}\right) = \frac{1}{\sqrt{2\pi a^2}} e^{-w^2/(2a^2)},$$

so $W \sim \mathcal{N}(0, a^2)$. Again, we remain in the Gaussian family.

Exercise 5.3.2. Let X_1, X_2 be independent with a common Laplace (double-exponential) distribution

$$p_{X_i}(x) = \frac{1}{2} e^{-|x|}.$$

1. Show that $Y = X_1 + X_2$ does not have a Laplace distribution by computing its PDF (via convolution) explicitly.
2. Compare the shape of p_Y to a Gaussian and to a Laplace; which tails are heavier?
3. Contrast this with the Gaussian example above. What does this say about the closure of distribution families under summation?

5.3.3 Multivariate Gaussian Distributions

A d -dimensional random vector X is Gaussian (or Normal) if it has density

$$p_X(x) = \frac{1}{(2\pi)^{d/2} |\Sigma|^{1/2}} \exp\left\{-\frac{1}{2}(x - \mu)^\top \Sigma^{-1}(x - \mu)\right\},$$

with mean $\mu \in \mathbb{R}^d$ and covariance matrix $\Sigma \succ 0$.

Among all distributions on \mathbb{R}^d , the multivariate Gaussian is particularly special because it is *closed* under many natural operations:

- marginalization (dropping coordinates),
- conditioning on linear observations,
- linear transformations $Y = AX + b$,
- summation of independent Gaussian vectors.

For general distributions these operations typically leave the original family, but for Gaussians they remain Gaussian. This “invariance under linear operations” is a major reason why Gaussians are so analytically tractable and so widely used.

Marginalization. Partition X into two blocks:

$$X = \begin{bmatrix} X_1 \\ X_2 \end{bmatrix}, \quad \mu = \begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix}, \quad \Sigma = \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{12}^\top & \Sigma_{22} \end{bmatrix}.$$

Then

$$X_1 \sim \mathcal{N}(\mu_1, \Sigma_{11}), \quad X_2 \sim \mathcal{N}(\mu_2, \Sigma_{22}).$$

Conditioning. The conditional distribution of X_1 given $X_2 = a$ is Gaussian:

$$X_1 | X_2 = a \sim \mathcal{N}(\mu_{1|2}, \Sigma_{1|2}),$$

with

$$\mu_{1|2} = \mu_1 + \Sigma_{12}\Sigma_{22}^{-1}(a - \mu_2), \quad \Sigma_{1|2} = \Sigma_{11} - \Sigma_{12}\Sigma_{22}^{-1}\Sigma_{12}^\top.$$

This is a concrete instance of Bayes’ rule in the Gaussian setting and will reappear in various guises (Kalman filters, Gaussian processes, linear Bayesian inference).

Linear Transformations and Sums. If $Y = AX + b$ with $A \in \mathbb{R}^{m \times d}$ and $b \in \mathbb{R}^m$, then

$$Y \sim \mathcal{N}(A\mu + b, A\Sigma A^\top).$$

In particular, if $X^{(1)}, X^{(2)}$ are independent with $X^{(k)} \sim \mathcal{N}(\mu^{(k)}, \Sigma^{(k)})$, then

$$X^{(1)} + X^{(2)} \sim \mathcal{N}(\mu^{(1)} + \mu^{(2)}, \Sigma^{(1)} + \Sigma^{(2)}).$$

Example 5.3.3. Gaussian Conditioning in Practice Consider

$$X = \begin{bmatrix} X_1 \\ X_2 \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix}\right).$$

From the formulas above,

$$\mu_{1|2} = 0 + 1 \cdot 1^{-1} \cdot (1 - 0) = 1, \quad \Sigma_{1|2} = 2 - 1 \cdot 1^{-1} \cdot 1 = 1,$$

so

$$X_1 | X_2 = 1 \sim \mathcal{N}(1, 1).$$

Such conditioning formulas underpin Bayesian linear regression, Kalman filtering, and Gaussian process prediction.

Exercise 5.3.3. Prove the marginalization, conditioning, and linear-transformation properties stated above by completing the Gaussian integrals and manipulations. Then verify them numerically by sampling from a chosen multivariate Gaussian and estimating:

- empirical marginals,
- empirical conditionals $X_1 | X_2 \in [a, a + \Delta]$,
- empirical distributions of linear transforms $Y = AX + b$.

Compare your empirical estimates with the analytical predictions as the sample size grows.

5.3.4 Empirical Multivariate Statistics

The empirical distribution and Law of Large Numbers experiments of Section 5.1 extend naturally to higher dimensions. Given i.i.d. samples

$$x_1, \dots, x_N \in \mathbb{R}^d,$$

we define the **sample mean** and **sample covariance** as

$$\hat{\mu} = \frac{1}{N} \sum_{i=1}^N x_i, \quad \hat{\Sigma} = \frac{1}{N} \sum_{i=1}^N (x_i - \hat{\mu})(x_i - \hat{\mu})^\top.$$

By the multivariate Law of Large Numbers,

$$\hat{\mu} \rightarrow \mu, \quad \hat{\Sigma} \rightarrow \Sigma \quad \text{almost surely as } N \rightarrow \infty.$$

Multivariate sampling reveals geometric phenomena not present in 1D: concentration of measure, anisotropy, correlated directions, and, for non-Gaussian data, multimodality and heavy tails. The accompanying Jupyter/Python notebook `Empirical-Multivariate.ipynb` implements these constructions and generates the figures used in this subsection. It can be used to:

- Visualize scatter plots and covariance ellipses in 2D;
- Track convergence of $\hat{\mu}$ and $\hat{\Sigma}$;
- Study the effects of correlations and linear transforms (whitening, PCA-like decorrelation);
- Compare Gaussian and non-Gaussian (mixture and heavy-tailed) clouds of points.

Example 5.3.4. Gaussian vs. Non-Gaussian Clouds in 2D

We compare three models for $X \in \mathbb{R}^2$.

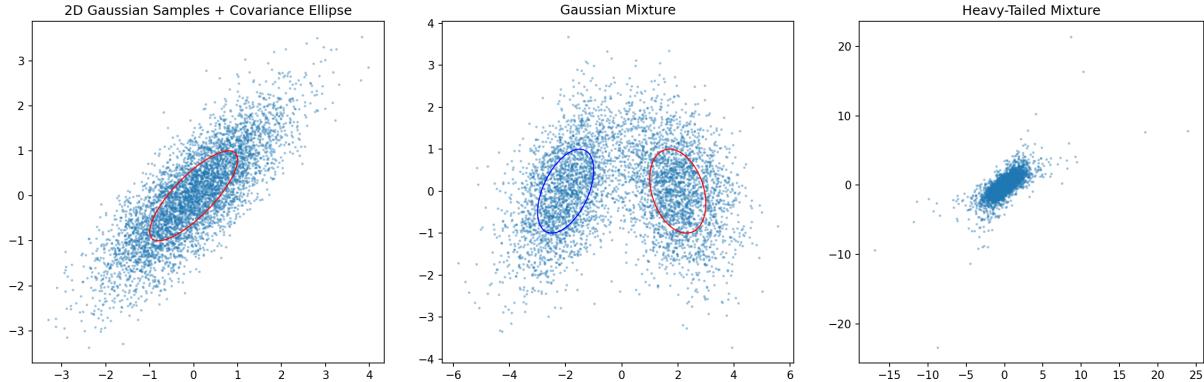


Figure 5.5: Left: Correlated Gaussian: samples and empirical covariance ellipse. Center: Two-component Gaussian mixture: bimodal cloud with a single covariance ellipse. Right: Heavy-tailed mixture: outliers and extended tails despite a finite covariance.

- **Correlated Gaussian.**

$$X^{(G)} \sim \mathcal{N}\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 & 0.8 \\ 0.8 & 1 \end{bmatrix}\right).$$

Scatter plots show an elongated elliptical cloud. As N grows, the empirical $\hat{\mu}$ and $\hat{\Sigma}$ converge to the true mean and covariance, and the covariance ellipse closely matches the visible level sets (left panel of Fig. 5.5).

- **Mixture of two Gaussians.**

$$X^{(M)} \sim \frac{1}{2} \mathcal{N}\left(\begin{bmatrix} -m \\ 0 \end{bmatrix}, I_2\right) + \frac{1}{2} \mathcal{N}\left(\begin{bmatrix} m \\ 0 \end{bmatrix}, I_2\right),$$

with $m > 0$ and I_2 the 2×2 identity. The scatter plot reveals two well-separated clusters. The empirical covariance still converges, but it cannot capture the multimodal structure: the covariance ellipse is centered between the modes and misses the “two-bump” shape (center panel of Figure 5.5).

- **Heavy-tailed mixture.** We modify the second component so that one coordinate follows a heavy-tailed distribution (e.g. a Student-t with a small number of degrees of freedom). The resulting cloud exhibits more extreme points and more spread in the heavy-tailed direction. The covariance still exists and converges, but it hides the frequency and severity of outliers (right panel of Fig. 5.5).

This contrast illustrates both the usefulness and the limitations of second-order statistics: for Gaussians, mean and covariance essentially describe the distribution; for more complex laws, they are only part of the story.

Exercise 5.3.4. Using the accompanying notebook *Empirical-Multivariate.ipynb*:

1. Generate samples from the correlated Gaussian and the two-component Gaussian mixture described in the example. Reproduce left and right panels of Fig. 5.5.

2. For the correlated Gaussian, compute $\hat{\mu}$ and $\hat{\Sigma}$ for increasing N and track their convergence. Compare your convergence curves (see notebook).
3. Plot covariance ellipses and overlay them on scatter plots. For the mixture, comment on what the covariance “sees” and what it misses.
4. Apply a whitening transform based on $\hat{\Sigma}$ and visualize the transformed samples. Are the whitened Gaussians approximately isotropic? What happens to the mixture after whitening? (See also other notebook figures.)
5. Replace one of the components with a heavy-tailed distribution (e.g. Laplace or a Student- t) and repeat. How do heavy tails manifest in sample covariance and in the scatter plots? Compare your results to right panel of Fig. 5.5.

5.4 From Aggregate Behavior to Rare Events

In earlier sections we examined how single random variables behave and how simple transformations (e.g. affine maps) affect their distributions. We now turn to a deeper and more universal question:

What happens when we combine many random variables?

Three classical asymptotic regimes appear again and again in probability, statistics, physics, and modern AI:

1. **Aggregating many small contributions** leads to the *Central Limit Theorem* (Gaussian limit).
2. **Aggregating many rare events** leads to the *Poisson limit*.
3. **Aggregating extreme outcomes** leads to *extreme-value limits* (Gumbel, Fréchet, Weibull).

These regimes complement one another: they describe limit behaviors not of a single variable, but of *collections* of variables, each producing a different form of universality.

The accompanying notebook `Aggregate-Rare-Events.ipynb` illustrates all of these regimes numerically and saves the figures used throughout this section into the `figs-final/chapter5/` folder.

We begin with the most famous result — the Central Limit Theorem (CLT) — and gradually broaden its scope and limitations.

5.4.1 The Central Limit Theorem: Weak Form

Earlier in the chapter we saw that sums of independent Gaussian variables remain Gaussian. The remarkable fact is that *even non-Gaussian* variables, when aggregated in large numbers, produce Gaussian-like behavior.

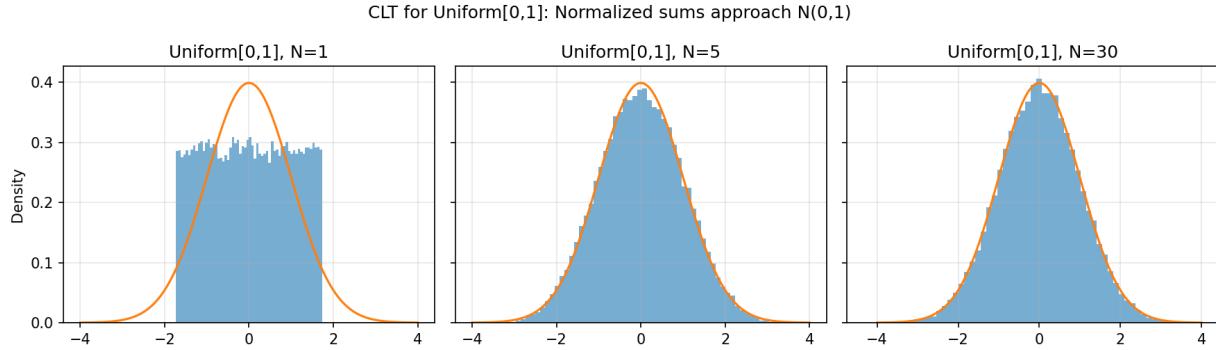


Figure 5.6: CLT for sums of i.i.d. Uniform[0, 1] variables: histograms of normalized sums for increasing N , overlaid with the standard Gaussian density. Generated by `Aggregate-Rare-Events.ipynb`.

Let X_1, \dots, X_N be i.i.d. with mean μ and variance $\sigma^2 < \infty$. Define the normalized sum

$$Z_N = \frac{X_1 + \dots + X_N - N\mu}{\sqrt{N\sigma^2}}.$$

The **Central Limit Theorem** (CLT) states:

$$Z_N \xrightarrow{d} \mathcal{N}(0, 1) \quad \text{as } N \rightarrow \infty.$$

In words:

Large sums of independent finite-variance variables become Gaussian.

This is a cornerstone of statistical modeling: many phenomena appear Gaussian not because their components are Gaussian, but because they are *aggregations of many (uncorrelated or weakly-correlated) contributions*. In the language of Section 5.1, we are looking at the distribution of a function of many i.i.d. samples from a fixed probability space.

Stronger versions. Beyond convergence in distribution, there are “almost sure” refinements (e.g. invariance principles), but the weak form suffices for most applications in applied math and AI.

Example 5.4.1. CLT in Action: Sums of Non-Gaussian Inputs

Let X_i be i.i.d. uniform on $[0, 1]$. For each N , form the normalized sum Z_N and plot its histogram alongside the standard Gaussian density. Even though the uniform law is far from Gaussian, the histograms of Z_N quickly resemble the bell curve as N grows.

Figure 5.6 shows this effect for several values of N . A similar experiment with exponential inputs is shown in Figure 5.7: despite the strong skew of the exponential law, the normalized sums again converge visually to a Gaussian.

The notebook `Aggregate-Rare-Events.ipynb` constructs these panels and can be extended with Q-Q plots or empirical CDF overlays to further quantify convergence in distribution.

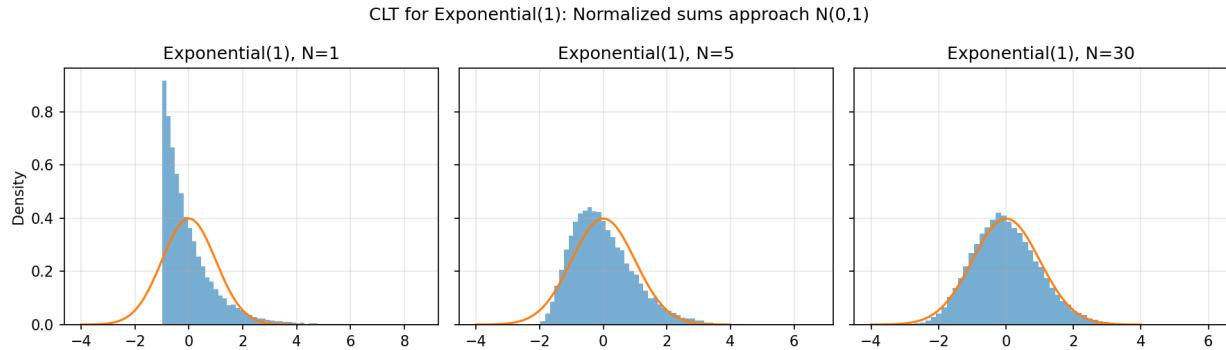


Figure 5.7: CLT for sums of i.i.d. exponential variables: strong skew in the one-variable distribution, but Gaussian behavior emerges for normalized sums. Generated by `Aggregate-Rare-Events.ipynb`.

Exercise 5.4.1. Using `Aggregate-Rare-Events.ipynb` as a template or starting from scratch, pick one non-Gaussian distribution (e.g. exponential, uniform, Bernoulli) and simulate normalized sums Z_N . For increasing N :

1. Plot histograms of Z_N and overlay the standard Gaussian density.
2. Add Q–Q plots or empirical CDFs to visualize convergence in distribution.
3. Compare convergence rates across different input distributions by inspecting how quickly the histograms and Q–Q plots align with the Gaussian reference.

5.4.2 Large Deviations: Tail Form of the CLT

While the CLT describes the *typical* fluctuations of order \sqrt{N} , we may also ask:

How likely are atypically large deviations?

Large deviation theory provides the answer. Under mild assumptions,

$$\mathbb{P}\left(\frac{1}{N} \sum_{i=1}^N X_i = x\right) \approx \exp(-N \Phi^*(x)),$$

where Φ^* is the **Cramér rate function**. It is convex, non-negative, and minimized at $x = \mu$. Thus the probability of extreme deviations decays *exponentially in N* . This complements the CLT: the CLT describes central fluctuations, while large deviation theory describes rare events far in the tails.

Example 5.4.2. Rare Averages Are Exponentially Unlikely

Consider $X_i \sim \text{Bernoulli}(1/2)$ and the sample average $\bar{X}_N = \frac{1}{N} \sum_{i=1}^N X_i$. The notebook estimates

$$\mathbb{P}(\bar{X}_N \geq 0.7)$$

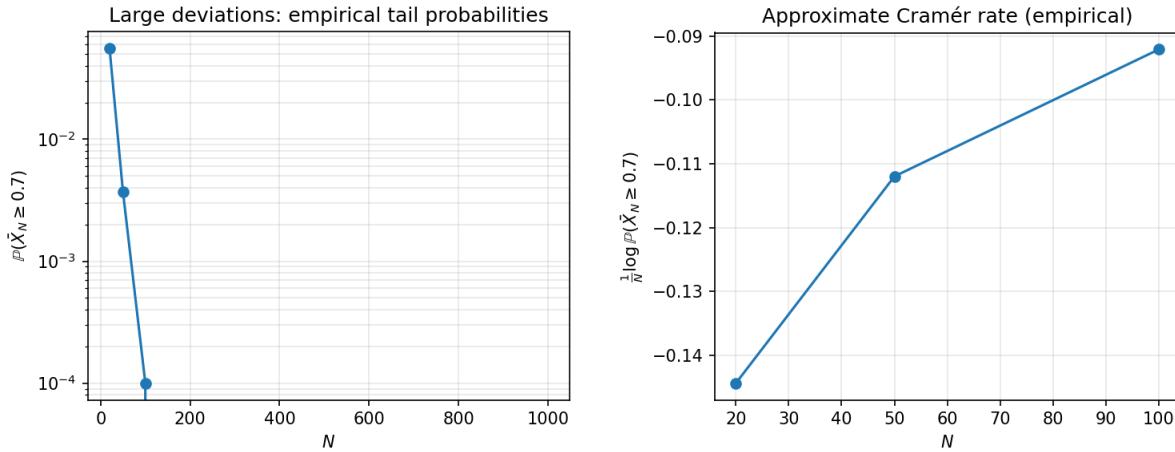


Figure 5.8: Left: Empirical tail probabilities $\mathbb{P}(\bar{X}_N \geq 0.7)$ for Bernoulli(1/2) averages, plotted versus N on a log scale. Right: Scaled log-probabilities $\frac{1}{N} \log \mathbb{P}(\bar{X}_N \geq 0.7)$ approaching a negative constant, illustrating a Cramér-type rate. Generated by `Aggregate-Rare-Events.ipynb`.

for increasing N . Left panel of Fig. 5.8 shows the empirical tail probabilities on a log scale, while the right panel plots

$$\frac{1}{N} \log \mathbb{P}(\bar{X}_N \geq 0.7)$$

versus N . The approximate linear decay in the log-probabilities and the convergence of the scaled log-probabilities to a negative constant provide (an early) numerical evidence for an exponential $\exp(-N\Phi^*(x))$ law.

Even moderate deviations (e.g. $\bar{X}_N \approx 0.6$) become exponentially unlikely as N grows; the figures make this suppression visually apparent.

Exercise 5.4.2. Using `Aggregate-Rare-Events.ipynb` or your own code:

1. For a chosen distribution (e.g. Bernoulli, uniform, or exponential), empirically estimate $\mathbb{P}(\frac{1}{N} \sum X_i \geq x)$ for several values of $x > \mu$.
2. Plot the tail probabilities on a log scale as a function of N and check whether the decay appears exponential.
3. Compute $\frac{1}{N} \log \mathbb{P}(\frac{1}{N} \sum X_i \geq x)$ and compare empirical slopes to theoretical Cramér-rate predictions where available.

5.4.3 When CLT Fails: Heavy Tails and Stable Laws

The CLT requires $\text{Var}(X) < \infty$. If the variance is infinite, the normalized sum *does not* converge to a Gaussian. Heavy-tailed distributions — often encountered in practice — can violate this assumption.

A distribution with power-law tails

$$p_X(x) \sim C |x|^{-(\alpha+1)}$$

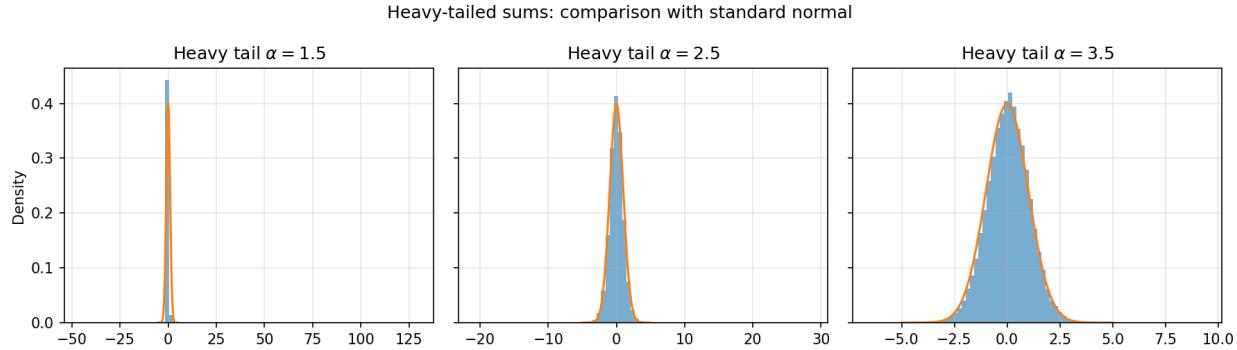


Figure 5.9: Normalized sums from finite-variance (Gaussian) vs. heavy-tailed inputs, with Gaussian reference density. Heavy tails persist under aggregation, illustrating breakdown of the CLT and emergence of non-Gaussian stable behavior. Generated by `Aggregate-Rare-Events.ipynb`.

has finite variance only if $\alpha > 2$. For $0 < \alpha \leq 2$, the variance diverges and Gaussian behavior disappears.

Instead, sums converge to a **stable distribution**, of which the Gaussian ($\alpha = 2$) is just one special case.

Example 5.4.3. Sums of Heavy-Tailed Variables

The notebook compares normalized sums of i.i.d. Gaussian and heavy-tailed variables with the same mean (zero), using histograms and Gaussian overlays. In the Gaussian case, the histograms rapidly become bell-shaped. In the heavy-tailed case, Figure 5.9 shows that even for large N , the distribution retains pronounced tails and outliers, clearly deviating from the Gaussian reference.

A particularly extreme case is the Cauchy distribution, which is stable with tail exponent $\alpha = 1$: if X_i are i.i.d. Cauchy, then $S_N = X_1 + \dots + X_N$ remains Cauchy for all N .

Exercise 5.4.3. Consider random variables with PDF

$$p(x) = \frac{C}{(1 + |x|)^{\alpha+1}}.$$

1. Show that $\mathbb{E}[X^2] < \infty$ if and only if $\alpha > 2$.
2. Using `Aggregate-Rare-Events.ipynb` as a guide, numerically simulate normalized sums for $\alpha = 1.5, 2.5, 3.5$ and compare histograms to the Gaussian density.
3. For $\alpha \leq 2$, verify that normalized sums do not resemble Gaussian distributions and discuss how the heavy tails manifest in the histograms and in occasional large outliers.

5.4.4 Rare Events in Many Trials: The Poisson Limit

The CLT describes the regime where many small contributions accumulate and the variance grows without bound. A completely different universal behavior appears when we have:

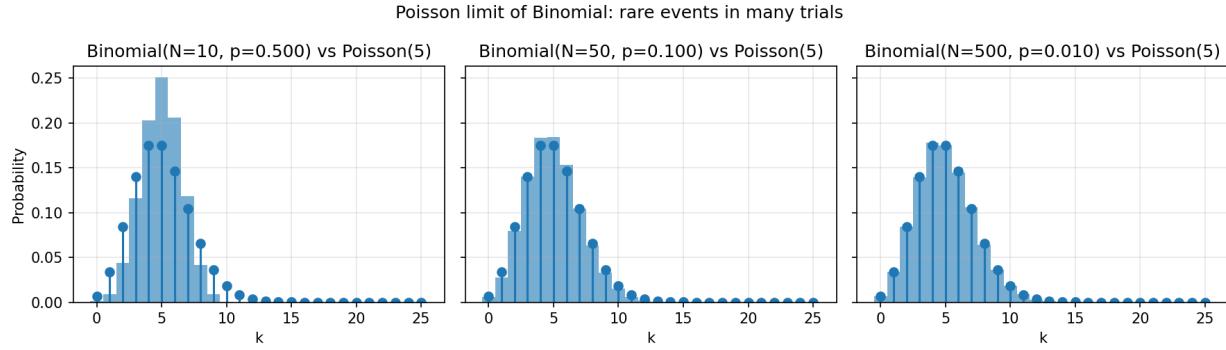


Figure 5.10: Poisson limit of binomial counts: binomial histograms for increasing N (with $\lambda = N\rho$ fixed) compared to the $\text{Poisson}(\lambda)$ PMF. Generated by `Aggregate-Rare-Events.ipynb`.

Many trials, but extremely rare successes.

Let $X_i \sim \text{Bernoulli}(\rho)$ and $S_N = \sum_{i=1}^N X_i$. If we let $N \rightarrow \infty$ while keeping $\lambda = N\rho$ fixed, then

$$S_N \xrightarrow{d} \text{Poisson}(\lambda).$$

This is the **Poisson limit theorem**: the universal distribution of counts of rare, nearly independent events.

Relation to the CLT. Here the variance stays bounded:

$$\text{Var}(S_N) = N\rho(1 - \rho) \approx \lambda,$$

so the CLT regime ($\text{variance} \rightarrow \infty$) does not apply. The Poisson limit and the CLT describe two *different scalings*: many small contributions with unbounded variance versus many rare events with bounded mean and variance.

Example 5.4.4. Modeling Defects in a Large-Scale Process

In `Aggregate-Rare-Events.ipynb` we generate samples from $\text{Binomial}(N, \rho)$ with $\rho = \lambda/N$ and fixed λ . Figure 5.10 compares the resulting histograms to the $\text{Poisson}(\lambda)$ PMF for increasing N . Already for moderate N , the binomial histograms lie almost exactly on top of the Poisson bars.

A concrete interpretation: if a machine produces a defective part with probability 0.001, then after inspecting $N = 1000$ parts, $S_N \sim \text{Binomial}(1000, 0.001)$ is well-approximated by $\text{Poisson}(1)$, greatly simplifying probability calculations.

Exercise 5.4.4. Fix $\lambda = 5$ and let $\rho = \lambda/N$.

1. Using `Aggregate-Rare-Events.ipynb`, simulate $S_N \sim \text{Binomial}(N, \rho)$ for $N = 10, 50, 100, 500, 1000$.
2. For each N , plot the histogram of S_N and overlay the $\text{Poisson}(\lambda)$ PMF.
3. Identify the smallest N at which the Poisson approximation becomes visually accurate, and discuss how this depends on λ .

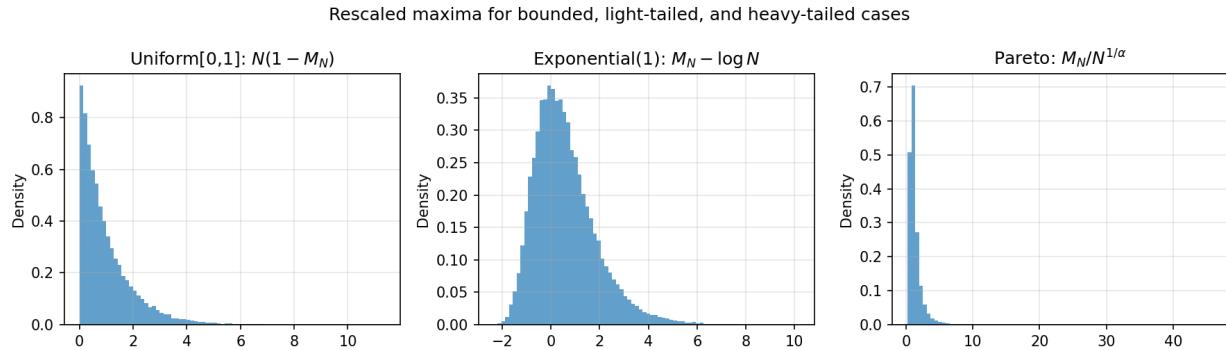


Figure 5.11: Rescaled maxima $N(1 - M_N)$ for Uniform[0, 1] variables approaching an exponential distribution, illustrating a simple extreme-value limit. Generated by `Aggregate-Rare-Events.ipynb`.

5.4.5 Beyond Sums: Extreme Value Theorems

Summation is only one way to aggregate many random variables. Another fundamental operation is taking the **maximum** or **minimum**. The limiting behavior is quite different from the CLT and depends strongly on tail characteristics.

Let $M_N = \max(X_1, \dots, X_N)$. Properly rescaled, M_N converges (in distribution) to one of three classical extreme-value laws:

- **Gumbel** (light-tailed distributions such as exponential, Gaussian),
- **Fréchet** (heavy-tailed power laws),
- **Weibull** (bounded distributions).

Sums produce Gaussian universality; maxima produce extreme-value universality.

Example 5.4.5. Maximum of Uniform Variables

The notebook simulates maxima M_N for Uniform[0, 1] variables and plots the rescaled quantity $N(1 - M_N)$. Figure 5.11 shows that as N grows, the empirical histogram approaches an exponential density:

$$N(1 - M_N) \xrightarrow{d} \text{Exp}(1).$$

Thus the distribution of extremes differs completely from both Gaussian (CLT) and Poisson (rare-event counts) limits.

Exercise 5.4.5. Extend `Aggregate-Rare-Events.ipynb` to simulate maxima M_N for:

1. Uniform variables (bounded support),
2. Exponential variables (light-tailed),
3. Pareto variables (heavy-tailed).

For each case:

- Propose a natural rescaling of M_N suggested by the theory of extreme-value limits.
- Plot histograms of the rescaled maxima for increasing N .
- Compare the empirical limits to the canonical Weibull, Gumbel, or Fréchet forms, and comment on how the tail behavior of the original distribution determines the limiting law.

Why These Universality Classes Matter for AI

The three asymptotic regimes we conclude this section with — Gaussian CLT limits, Poisson rare-event limits, and extreme-value limits — are not just abstract probability results. They appear implicitly in the behavior of modern AI systems:

- **Gaussian (CLT) regime and SGD noise.** In large models trained with mini-batch stochastic gradient descent (SGD), each gradient update is an *average* over many sample-wise contributions. By a CLT-type argument, mini-batch noise is often approximated as *Gaussian*, which underpins continuous-time diffusion approximations and Langevin-type views of optimization. This perspective will reappear when we interpret training dynamics as stochastic processes in high dimension (later in Chapters 7 and 9).
- **Poisson regime and rare events in RL.** In reinforcement learning, rewards or failures can be extremely *rare* (sparse rewards, catastrophic events). Counts of such events over many episodes naturally fall into a Poisson-like regime. Understanding when we are in a “rare-events” scaling rather than a CLT scaling affects how we design exploration strategies, off-policy evaluation, and safety-critical RL, where the expected *number* of critical events, not just their average magnitude, matters.
- **Extreme-value regime and risk-sensitive modeling.** Many real-world applications of AI are *risk-sensitive*: we care about the *worst* (or near-worst) outcomes, not just the mean. This includes tail-risk in financial models, maximum load in networks, largest error in perception systems, or worst-case loss under distribution shift. Such questions are governed by the asymptotics of *maxima* and tail behavior, i.e., extreme-value theory, and motivate objectives such as Value-at-Risk (VaR), Conditional VaR, and robust / adversarial training.

In summary, the three universality classes in this section provide a conceptual map for thinking about noise and uncertainty in AI: *Gaussian* fluctuations for aggregate gradient noise and diffusion models, *Poisson* statistics for sparse or count-based signals, and *extreme-value* behavior for safety- and risk-critical systems. We will return to these themes multiple times when we discuss information, stochastic processes, and generative AI in later chapters.

Chapter 6

Entropy and Information Theory

Why Entropy and Information Theory?

Entropy first appeared in **thermodynamics** as a measure of disorder and irreversibility in physical systems. Shannon’s fundamental insight was that the same mathematical structure captures the *uncertainty* of an information source. This bridge from physics to **information theory** transformed entropy into one of the central concepts of modern probability, statistics, machine learning, and generative AI.

In this chapter we adopt the information-theoretic viewpoint:

- **Entropy** quantifies uncertainty and the minimal number of bits needed to encode a random variable.
- **Kullback–Leibler (KL) divergence** measures how one probability distribution differs from another, forming the basis of likelihood-based training, variational inference, and normalizing flows.
- **Mutual information** quantifies how much knowledge of one variable reduces uncertainty about another, playing a central role in representation learning, bottleneck architectures, and the analysis of neural networks.

These notions did not arise from AI, but they now underpin almost every modern generative model:

- In **diffusion models**, entropy controls the trade-off between randomness and structure during denoising.
- In **variational autoencoders (VAEs)**, KL divergence shapes the latent distribution and regularizes the encoder.
- In **score-based and flow-matching models**, divergences between probability flows guide the training dynamics.

Although KL divergence dominates classical optimization-based approaches, it behaves poorly when distributions lie on low-dimensional manifolds or have non-overlapping support. This motivates the introduction of **Wasserstein distances**, which measure how probability *mass* must move in space and provide a more geometric notion of discrepancy; these will be previewed in this chapter and explored more deeply in later chapters.

Connector to the Rest of the Book. This chapter builds directly on the probability and sampling foundations established in Chapter 5. It introduces the quantitative language needed for the next stages of the book:

- **Stochastic processes** (next chapter), where entropy and KL measure complexity of paths and clarify connections between sampling, diffusion, and control.
- **Variational methods, VAEs, and normalizing flows**, where KL and cross-entropy become primary training objectives.
- **Diffusion models, flow matching, and optimal transport**, where Wasserstein geometry and score-based dynamics reveal deep structural connections.

Our goal in this chapter is therefore twofold: (1) to introduce the core information-theoretic quantities that govern uncertainty and information transfer, and (2) to prepare the conceptual groundwork for the generative modeling frameworks developed in the subsequent chapters.

6.1 Conditional Probability and Bayes’ Rule

Modern AI systems operate under uncertainty: sensors are noisy, labels are imperfect, and environments are only partially observed. Whether it is a **self-driving car** estimating its position from noisy GPS and LiDAR data, or a **medical AI** diagnosing a disease from imperfect tests, such systems must continually update their beliefs in light of new evidence. This process is formalized by **conditional probability** and **Bayes’ rule**. In this section we:

- Define conditional and joint probabilities, and derive Bayes’ theorem;
- Interpret the key components: *prior*, *likelihood*, *posterior*, and *evidence*;
- Work through a concrete, fully discrete toy example (disease–test);
- Connect to simple AI-style classifiers (Bayesian filtering, Naïve Bayes).

6.1.1 Conditional Probability, Joint Laws, and Bayes’ Rule

Let A and B be events in a probability space. The **conditional probability** of A given B is

$$\mathbb{P}(A | B) = \frac{\mathbb{P}(A \cap B)}{\mathbb{P}(B)}, \quad \mathbb{P}(B) > 0. \quad (6.1)$$

This tells us how to update our belief in A after learning that B has occurred. Rearranging gives the **multiplication rule**

$$\mathbb{P}(A \cap B) = \mathbb{P}(A | B) \mathbb{P}(B) = \mathbb{P}(B | A) \mathbb{P}(A). \quad (6.2)$$

Equating the two expressions for $\mathbb{P}(A \cap B)$ yields **Bayes' rule**:

$$\mathbb{P}(A | B) = \frac{\mathbb{P}(B | A) \mathbb{P}(A)}{\mathbb{P}(B)}, \quad \mathbb{P}(B) = \sum_a \mathbb{P}(B | A = a) \mathbb{P}(A = a). \quad (6.3)$$

The terms have standard names in Bayesian modeling:

- | | |
|---------------------|------------------------------------|
| $\mathbb{P}(A)$ | prior (belief before seeing B), |
| $\mathbb{P}(B A)$ | likelihood (data model), |
| $\mathbb{P}(A B)$ | posterior (updated belief), |
| $\mathbb{P}(B)$ | evidence or marginal likelihood. |

In AI applications we often write $A = X$ (hidden state, class, parameter) and $B = Z$ (observation, feature, measurement), so that

$$\mathbb{P}(X | Z) \propto \mathbb{P}(Z | X) \mathbb{P}(X),$$

where the proportionality constant is the evidence $\mathbb{P}(Z)$.

6.1.2 Discrete Bayes in a Toy Medical Diagnosis Model

To see Bayes' rule in action, consider a simple medical diagnosis problem. Both the (hidden) disease state and the observed test outcome are modeled as binary random variables:

$$D \in \{0, 1\}, \quad T \in \{0, 1\},$$

where

$$D = 1 \text{ means "sick"}, \quad D = 0 \text{ means "healthy"},$$

and

$$T = 1 \text{ means "test positive"}, \quad T = 0 \text{ means "test negative".}$$

We specify:

- A prior prevalence:

$$\mathbb{P}(D = 1) = \pi, \quad \mathbb{P}(D = 0) = 1 - \pi;$$

- A likelihood model for the test:

Sensitivity:	$\mathbb{P}(T = 1 D = 1) = s,$
Specificity:	$\mathbb{P}(T = 0 D = 0) = c.$

Thus,

$$\mathbb{P}(T = 1 | D = 0) = 1 - c, \quad \mathbb{P}(T = 0 | D = 1) = 1 - s.$$

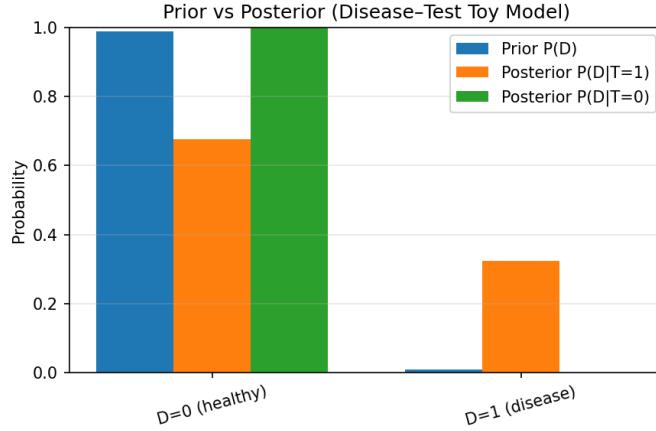


Figure 6.1: Prior vs. posterior probabilities in the toy disease–test model. Even an accurate test yields only moderate posterior probability when the disease is rare.

Example 6.1.1 (Positive Test for a Rare Disease). *Fix*

$$\pi = 0.01, \quad s = 0.95, \quad c = 0.98,$$

so the disease is rare, but the test is quite accurate. Bayes’ rule gives

$$\begin{aligned} \mathbb{P}(T = 1) &= s\pi + (1 - c)(1 - \pi), \\ \mathbb{P}(D = 1 \mid T = 1) &= \frac{s\pi}{\mathbb{P}(T = 1)}. \end{aligned}$$

Even with high sensitivity and specificity, the posterior $\mathbb{P}(D = 1 \mid T = 1)$ may remain far from 1, because the prior prevalence π is very small.

Fig. 6.1 shows a bar plot of the prior prevalence and the posterior probability after observing a positive test.

The notebook `Bayes-Toy-Discrete.ipynb` implements this model, computes the posterior using Bayes’ rule, and generates Fig. 6.1 automatically.

6.1.3 Exploring Test Quality: Sensitivity and Specificity

The same notebook also explores how the posterior $\mathbb{P}(D = 1 \mid T = 1)$ changes when we vary the *test quality* parameters while keeping the disease prevalence π fixed.

For example, we may sweep the sensitivity s from 0.5 to 0.99 while holding specificity c fixed, or vice versa. In each case, we recompute

$$\mathbb{P}(D = 1 \mid T = 1) = \frac{s\pi}{s\pi + (1 - c)(1 - \pi)},$$

and display the resulting curve.

Fig. 6.2 shows a typical result: improving sensitivity or specificity increases the posterior probability, but the effect is asymmetric when the disease is rare. Understanding these relationships is crucial for designing and interpreting diagnostic tests.

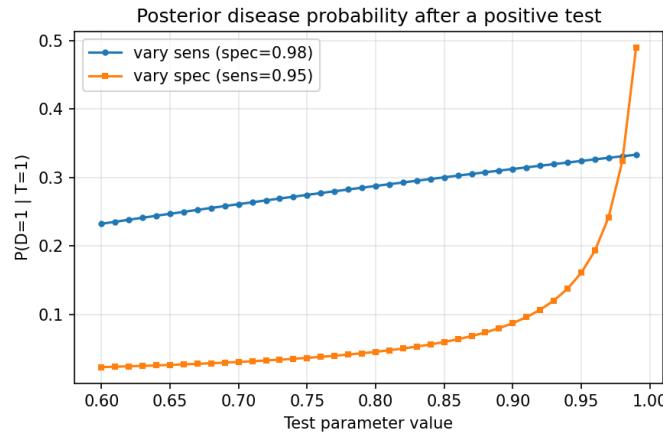


Figure 6.2: Posterior probability $\mathbb{P}(D = 1 \mid T = 1)$ as a function of test sensitivity s (with specificity c fixed). The notebook `Bayes-Toy-Discrete.ipynb` produces this figure.

Exercise 6.1.1 (Discrete Bayesian Inference Toy). *Using the notebook `Bayes-Toy-Discrete.ipynb`:*

1. Implement the disease–test model with parameters (π, s, c) and verify numerically:

- the prior $\mathbb{P}(D)$,
- the likelihood $\mathbb{P}(T \mid D)$,
- the evidence $\mathbb{P}(T = 1)$,
- the posterior $\mathbb{P}(D = 1 \mid T = 1)$.

2. Reproduce the bar plot in Fig. 6.1 comparing prior and posterior probabilities.

3. Sweep test quality parameters:

- vary sensitivity $s \in [0.5, 0.99]$ with specificity c fixed;
- optionally vary specificity c with sensitivity fixed.

Generate a plot like Fig. 6.2. Comment on how improving each parameter affects the posterior.

4. Optional: Compare which improvement (sensitivity vs. specificity) has a larger effect when the disease is rare vs. common.

6.1.4 From Naïve Bayes to Neural Networks

The toy example in the previous subsection illustrated how Bayes' rule updates a prior belief in light of new evidence. We now move to a setting more directly connected to modern machine learning – *classification of images* – one already familiar from earlier chapters. Here, the goal is to infer a latent class label C (e.g., a digit $0, \dots, 9$) from observed features F extracted from the image.

Although state-of-the-art classifiers use neural networks, the probabilistic structure underlying Bayes' rule remains the same:

$$\mathbb{P}(C | F) \propto \mathbb{P}(C) \mathbb{P}(F | C).$$

To illustrate we consider an example of a small MNIST experiment connecting three perspectives:

1. A **Naïve Bayes classifier**, which assumes conditional independence of pixel features given the class.
2. A **neural network classifier**, which learns a likelihood model $\mathbb{P}(C | F)$ directly from data.
3. **Evaluation tools** – confusion matrices and calibration curves—that will return in later sections when we discuss KL divergence and cross-entropy.

Example 6.1.2 (Naïve Bayes vs Neural Network for MNIST). *Given an image represented as a vector of discrete pixel intensities, Naïve Bayes models the likelihood as*

$$\mathbb{P}(F | C = c) = \prod_{i=1}^d \mathbb{P}(F_i | C = c),$$

treating individual pixels as conditionally independent. This assumption is unrealistic for image data – but surprisingly effective as a baseline, and invaluable as a conceptual stepping stone toward more flexible models.

A small neural network trained by minimizing cross-entropy learns a much richer representation. Its predictions

$$\hat{p}(C | F)$$

can be compared to Naïve Bayes using two complementary visualizations:

- a **confusion matrix**, showing where each model makes mistakes;
- a **calibration (reliability) curve**, comparing predicted probabilities to empirical frequencies.

These quantitative tools also prepare us for the next sections, where we connect entropy, KL divergence, and cross-entropy to empirical performance.

The notebook `Bayes-MNIST-NaiveBayes.ipynb` implements:

- training a small neural network on MNIST,
- training a Naïve Bayes classifier,
- computing confusion matrices, shown in Fig. (6.3), and calibration curves, shown in Fig. (6.4)

This notebook will be reused later when we introduce KL divergence and cross-entropy, since the predictions of both models provide natural examples of probability distributions to compare.

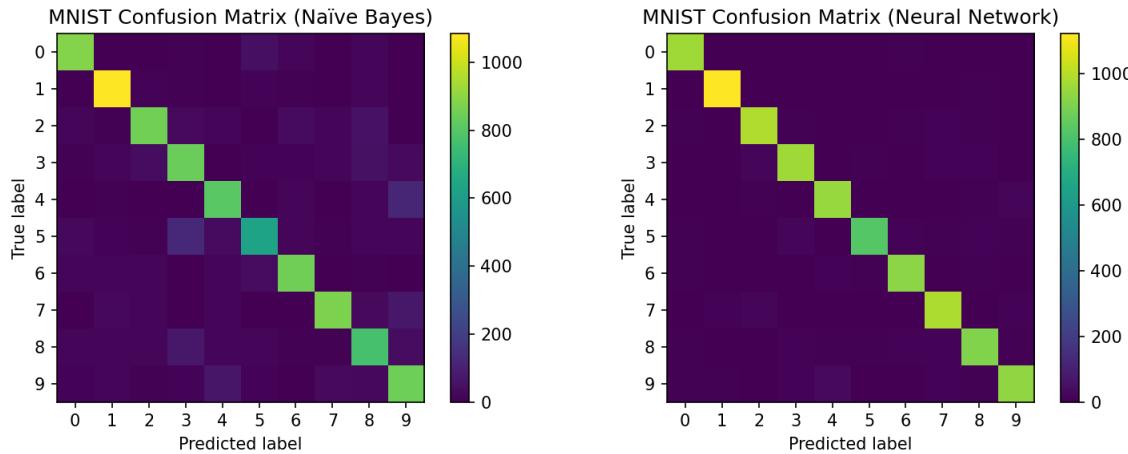


Figure 6.3: Example confusion matrices for the naïve Bayes (left) and neural network classifier (right).

Exercise 6.1.2 (Diagnosing Model Failures: Feature Dependence, Calibration, and KL Analysis). *Using the notebook `Bayes-MNIST-NaiveBayes.ipynb`, perform a deeper investigation of why Naïve Bayes underperforms compared to the neural network, and how the two models differ as probabilistic predictors.*

1. **Identify systematic failure modes.** Using the confusion matrices in Fig. 6.3, choose two pairs of digits that Naïve Bayes confuses much more often than the neural network (e.g. (3, 5) or (4, 9)).

For each selected pair:

- Visualize several misclassified images.
- Overlay (or display side-by-side) the Naïve Bayes class-conditional mean images, i.e. $\mathbb{E}[F | C = c]$.
- Explain which pixel dependencies break the Naïve Bayes independence assumption and lead to the observed errors.

2. **Calibration vs confidence.** Using the calibration curves in Fig. 6.4, select probability bins where:

- the neural network is overconfident, and
- Naïve Bayes is underconfident.

For each bin:

- inspect several test images whose predicted probability lies in that bin,
- report the empirical accuracy,
- comment on whether the miscalibration is due to poor likelihood modeling, insufficient capacity, or noisy training labels.

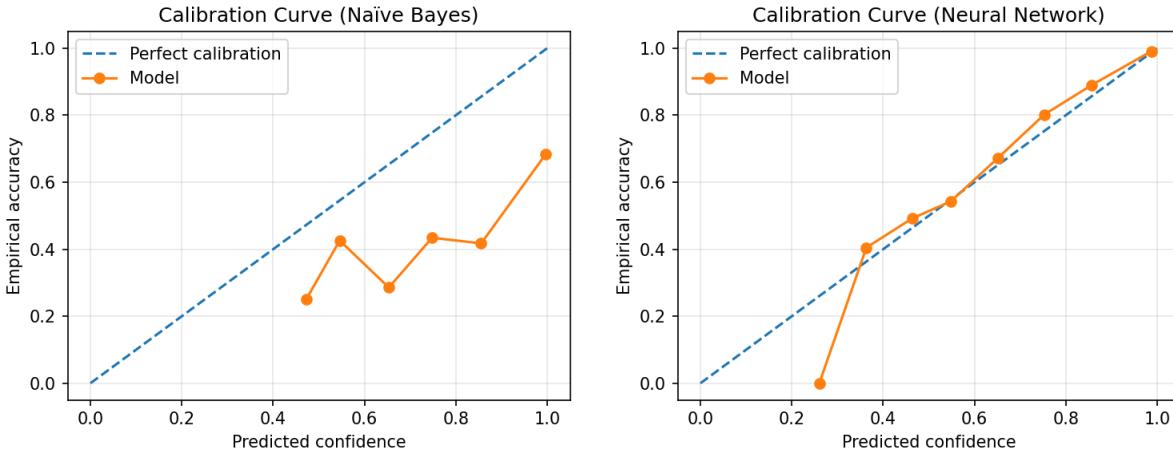


Figure 6.4: Example calibration (reliability) curves for the naïve Bayes (left) and for the neural network classifier (right). The gap between the curves and the diagonal indicates miscalibration.

3. **Per-sample KL divergence (preview of Section 6.2.3).** For every test image, compute the KL divergence between the two model predictions:

$$D_{\text{KL}}\left(p_{\text{NN}}(\cdot \mid F) \parallel p_{\text{NB}}(\cdot \mid F)\right).$$

Then:

- plot a histogram of these KL values,
- identify the images with the largest KL divergence,
- inspect them visually and explain why Naïve Bayes and the NN disagree so strongly.

(This KL analysis will be revisited later, when we study divergence-based training objectives.)

4. **Optional: selective feature removal.** Choose a pair of digits that Naïve Bayes confuses. Remove (mask) a small set of informative pixels (e.g. central strokes), retrain both models, and compare:

- changes in confusion rates,
- changes in calibration error,
- changes in the KL divergence distribution.

Discuss whether the neural network is more robust to missing / corrupted features than Naïve Bayes.

6.2 Entropy: Quantifying Uncertainty

Entropy in AI serves two primary roles:

- **Measuring Uncertainty:** The entropy of a probability distribution quantifies how uncertain or unpredictable a system is. High entropy corresponds to greater randomness, while low entropy suggests more certainty.
- **Guiding Learning and Decision-Making:** Many AI models – such as **variational autoencoders (VAEs)**, **generative adversarial networks (GANs)**, and **diffusion models** – explicitly optimize entropy-based loss functions to balance structure and randomness.

For example:

- In **reinforcement learning**, entropy regularization ensures that agents explore diverse strategies.
- In **generative models**, entropy helps control randomness in data generation, making outputs more varied or precise.

6.2.1 Definition and Interpretations of Entropy

For a discrete random variable X with probability mass function $P(X)$, the (Shannon) entropy is

$$H(X) = - \sum_{x \in \mathcal{X}} P(x) \log P(x). \quad (6.4)$$

For a continuous random variable, the *differential* entropy is

$$H(X) = - \int p(x) \log p(x) dx. \quad (6.5)$$

Entropy admits several complementary interpretations:

- **Uncertainty measure.** A uniform distribution has maximal entropy; a deterministic distribution has zero entropy.
- **Expected information content.** Entropy is the expected number of nats (or bits) required to encode a sample drawn from P .
- **Decision-theoretic perspective.** In reinforcement learning, entropy regularization helps balance exploration and exploitation.

Example 6.2.1 (Entropy of Class Distributions). *A classification dataset consists of samples (X, Y) , where Y is the discrete class label. The entropy of the class distribution,*

$$H(Y) = - \sum_{y \in \mathcal{Y}} P_Y(y) \log_2 P_Y(y),$$

quantifies class imbalance and the inherent label uncertainty.

For MNIST, which contains ten digit classes that are nearly uniformly represented, the theoretical entropy is

$$H(Y) = \log_2 10 \approx 3.32 \text{ bits.}$$

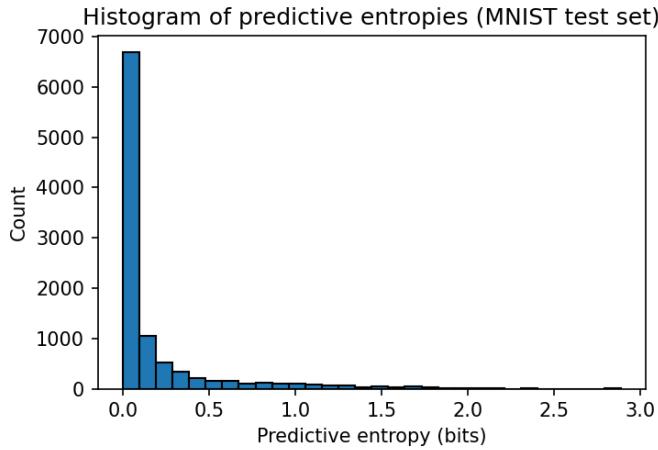


Figure 6.5: Histogram of predictive entropies $H_{\text{pred}}(x)$ for MNIST test images. Most predictions are low-entropy (high confidence), but a nontrivial tail highlights ambiguous digits.

Given a classifier producing a predictive distribution $\hat{p}(y | x)$ for each input x , the predictive entropy

$$H_{\text{pred}}(x) = - \sum_{y \in \mathcal{Y}} \hat{p}(y | x) \log \hat{p}(y | x)$$

measures the model's uncertainty on that particular example. Low entropy = confident prediction; high entropy = ambiguity or model uncertainty.

The `Entropy-Classification-Experiment-torch.ipynb` notebook computes this predictive entropy over the MNIST test set and visualizes its distribution in Fig. 6.5. The bulk of the mass lies near zero—indicating confident predictions—while a visible tail corresponds to challenging or visually ambiguous digits.

High- and low-entropy examples. To further interpret predictive entropy, the notebook displays the test samples with the lowest and highest entropy values.

- **Lowest entropy:** clean, prototypical digits for which the classifier confidently assigns nearly all probability mass to one class.
- **Highest entropy:** ambiguous or atypical digits, often resembling multiple classes (e.g., 4 vs. 9, 3 vs. 5).

Representative grids generated by the notebook are shown in Figs. 6.6.

Exercise 6.2.1 (Entropy in Data and in Predictive Models). Using the notebook `Entropy-Classification-Experiment.ipynb`:

1. **Dataset entropy.** Compute the class entropy $H(Y)$ for MNIST and for any user-constructed imbalanced variant (e.g., subsample the digit “1” by a factor of 10). How does class imbalance affect entropy?

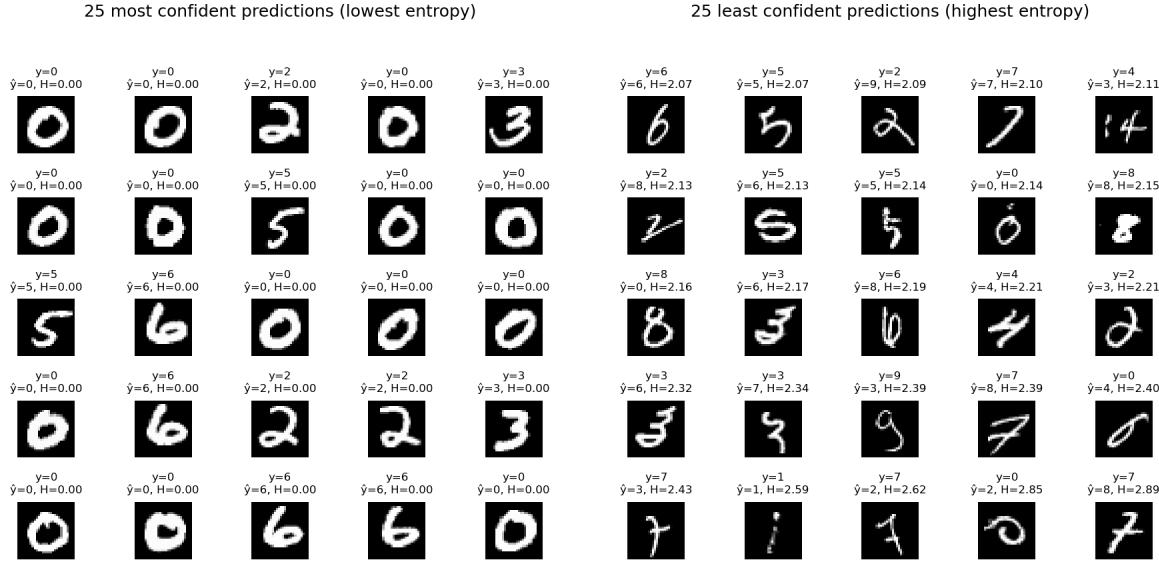


Figure 6.6: MNIST test samples with *lowest* (left) and *highest* (right) predictive entropy. These prototypical digits with lowest predictive entropy lead to highly confident (near-delta) predictions; while images correspondent to highest predictive entropy are visually ambiguous, leading the classifier to distribute its probability mass across multiple plausible classes.

2. **Predictive-entropy vs. accuracy.** Split the test set into correctly and incorrectly classified samples. Compare the predictive-entropy histograms for the two groups. Does higher entropy correlate with errors?
3. **Extreme cases.** Identify test samples with (i) maximum predictive entropy and (ii) minimum predictive entropy. Display the images and their predicted distributions. Explain why the model is (un)certain.
4. **Optional: temperature scaling.** Apply temperature scaling $p_T(y | x) \propto p(y | x)^{1/T}$ for several values of T . Analyze how predictive entropy changes and relate this to calibration behavior (see Section 6.2.3).

6.2.2 Mutual Information

Mutual information quantifies the amount of information shared between two random variables X and Y . It measures how much knowing Y reduces uncertainty about X and vice versa. The mutual information is defined as:

$$I(X; Y) = H(X) - H(X | Y), \quad (6.6)$$

where $H(X)$ is the entropy of X and $H(X | Y)$ is the conditional entropy of X given Y . This formulation makes it clear that mutual information represents the reduction in uncertainty about X after observing Y .

Equivalently, mutual information can be expressed as:

$$I(X; Y) = \sum_{x,y} P(x, y) \log \frac{P(x, y)}{P(x)P(y)}, \quad (6.7)$$

which shows that mutual information measures the divergence between the joint distribution $P(X, Y)$ and the product of the marginal distributions $P(X)P(Y)$. If X and Y are independent, then $P(X, Y) = P(X)P(Y)$, leading to $I(X; Y) = 0$.

Intuition and Visualization: Mutual information can be intuitively understood using **Venn diagrams** in an information-theoretic sense. The entropy of X and Y can be visualized as two overlapping sets, where the overlap represents their shared information.

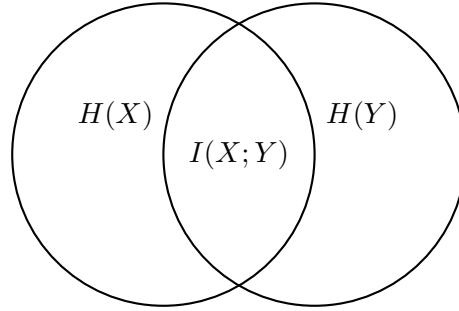


Figure 6.7: Mutual information $I(X; Y)$ as the intersection of entropy $H(X)$ and $H(Y)$.

In the diagram shown in Fig. (6.7: The left circle represents $H(X)$, the uncertainty in X ; The right circle represents $H(Y)$, the uncertainty in Y ; The overlapping area represents $I(X; Y)$, the mutual information, i.e., the reduction in uncertainty about one variable given the other; The non-overlapping parts correspond to the conditional entropies $H(X | Y)$ and $H(Y | X)$.

Example 6.2.2. Example: Coin Flip with Noise: Consider a scenario where we flip a fair coin (X) but then communicate the result through a noisy channel (Y) where there is a 10% chance that the transmitted value is incorrect: If there were no noise, $I(X; Y) = H(X)$ because knowing Y would fully determine X ; If the noise were extreme (randomizing Y completely), then $I(X; Y) = 0$, as Y contains no information about X .

Exercise 6.2.2. Computing Mutual Information for a Simple Distribution: Consider a binary random variable X that takes values $\{0, 1\}$ with equal probability, and another random variable Y that is a noisy observation of X , with error probability p . That is,

$$P(Y = X) = 1 - p, \quad P(Y \neq X) = p.$$

Compute:

1. The entropies $H(X)$ and $H(Y)$.
2. The conditional entropy $H(X | Y)$.

3. The mutual information $I(X; Y)$.

How does $I(X; Y)$ behave as p varies?

Exercise 6.2.3. Mutual Information Between Images and Classifier Predictions (MNIST) Extend the notebook *Entropy-Classification-Experiment.ipynb* to numerically explore mutual information between the true class label Y and the classifier's predictive distribution $\hat{p}(y | x)$ on MNIST.

Recall that for discrete variables,

$$I(Y; \hat{Y}) = H(Y) - H(Y | \hat{Y}),$$

and that for a probabilistic classifier we may approximate

$$H(Y | X = x) \approx - \sum_y \hat{p}(y | x) \log \hat{p}(y | x) = H_{\text{pred}}(x),$$

the predictive entropy already computed in the notebook.

In this exercise:

1. **Estimate the conditional entropy.** Using the predictive entropies computed over the MNIST test set, estimate

$$H(Y | X) \approx \mathbb{E}_{x \sim \text{test set}} [H_{\text{pred}}(x)].$$

2. **Compute the mutual information.** Using the empirical class entropy $H(Y)$ already computed in the notebook and your estimate of $H(Y | X)$, compute

$$I(Y; X) \approx H(Y) - H(Y | X).$$

Interpret this quantity: how many bits of class information does the trained classifier typically “recover” from an image?

3. **Per-class analysis.** For each digit $y \in \{0, \dots, 9\}$, compute the class-conditional average predictive entropy

$$\mathbb{E}[H_{\text{pred}}(X) | Y = y].$$

Visualize these ten values as a bar plot. Which digits have the lowest uncertainty? Which have the highest? Relate this to the geometry of the digit classes (e.g., 1 vs. 8).

4. **High-information vs. low-information images.** Identify:

- the 25 images with lowest predictive entropy (highest information);
- the 25 with highest predictive entropy (lowest information).

Display both grids (similar to earlier entropy visualizations). Comment on visual patterns: Do high-information images look more prototypical? Are low-information images ambiguous or unusually shaped?

5. **Optional: Comparing models.** Train a second classifier (e.g. a deeper MLP or a CNN) in the same notebook. Repeat items 1–4 for both models. Compare the mutual-information estimates:

$$I_{\text{MLP}}(Y; X) \quad \text{vs.} \quad I_{\text{CNN}}(Y; X).$$

Does the better-performing model recover more information about the label?

This exercise connects mutual information to practical classifier behavior: $H(Y)$ reflects dataset uncertainty, $H(Y | X)$ reflects residual model uncertainty, and their difference quantifies how much information a trained classifier extracts from each image.

6.2.3 KL Divergence: Comparing Distributions

The **Kullback–Leibler (KL) divergence** quantifies how much one probability distribution P differs from another Q . For a discrete variable with support \mathcal{X} , it is defined as

$$D_{\text{KL}}(P \| Q) = \sum_{x \in \mathcal{X}} P(x) \log \frac{P(x)}{Q(x)}. \quad (6.8)$$

- $D_{\text{KL}}(P \| Q) = 0$ iff $P = Q$ almost everywhere.
- Larger values indicate increasing dissimilarity between P and Q .

KL divergence is not a metric. KL divergence does not satisfy symmetry or the triangle inequality:

1. **Non-symmetry:** $D_{\text{KL}}(P \| Q) \neq D_{\text{KL}}(Q \| P)$.
2. **No triangle inequality.**
3. **Non-negativity:** $D_{\text{KL}}(P \| Q) \geq 0$.

Thus KL divergence is best interpreted as a *directed measure of information loss* when approximating a “true” distribution P with a model Q .

Classifier outputs as distributions. A K -class classifier maps each input F to a probability vector $p(\cdot | F)$. If two models—here a Neural Network (NN) and a Naïve Bayes classifier (NB)—produce distributions $p_{\text{NN}}(\cdot | F)$ and $p_{\text{NB}}(\cdot | F)$ the KL divergence provides a natural way to measure how strongly they disagree on each input.

Example 6.2.3 (KL Divergence Between Naïve Bayes and NN on MNIST). *Continuing the MNIST experiment of Subsection 6.1.4, the notebook Bayes-MNIST-NaiveBayes.ipynb computes, for every test image F , the predictive distributions*

$$p_{\text{NB}}(\cdot | F), \quad p_{\text{NN}}(\cdot | F),$$

and the directed divergences

$$D_{\text{KL}}(p_{\text{NN}}(\cdot | F) \| p_{\text{NB}}(\cdot | F)), \quad D_{\text{KL}}(p_{\text{NB}}(\cdot | F) \| p_{\text{NN}}(\cdot | F)).$$

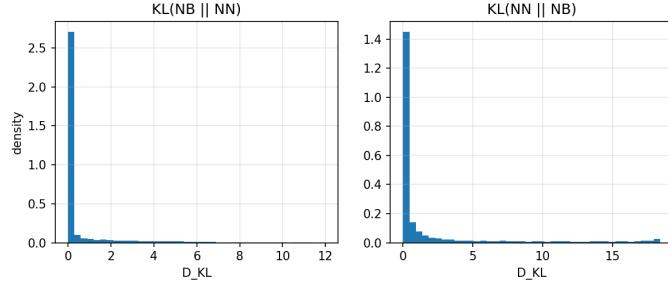


Figure 6.8: Histogram of $D_{\text{KL}}(p_{\text{NN}}(\cdot | F) \| p_{\text{NB}}(\cdot | F))$ over the MNIST test set. A heavy tail indicates inputs on which the neural network and Naïve Bayes produce substantially different predictive distributions.

A first global summary is provided by the histogram of $D_{\text{KL}}(p_{\text{NN}} \| p_{\text{NB}})$ over the test set, shown in Fig. 6.8. Most KL values are small—the two models usually agree as probabilistic predictors—but the distribution exhibits a clear tail corresponding to inputs on which the models diverge significantly.

Exercise 6.2.4 (KL Divergence and Model Disagreement on MNIST). Using the (extended) notebook `Bayes-MNIST-NaiveBayes.ipynb`:

1. **Global statistics.** Compute the mean KL divergences

$$\overline{D}_{\text{KL}}^{\text{NN} \parallel \text{NB}} = \mathbb{E}\left[D_{\text{KL}}(p_{\text{NN}}(\cdot | F) \| p_{\text{NB}}(\cdot | F))\right],$$

and

$$\overline{D}_{\text{KL}}^{\text{NB} \parallel \text{NN}} = \mathbb{E}\left[D_{\text{KL}}(p_{\text{NB}}(\cdot | F) \| p_{\text{NN}}(\cdot | F))\right].$$

Comment on which direction is larger and why the asymmetry arises.

2. **Condition on correctness.** Partition the test set into four groups:

- both models correct,
- only NN correct,
- only NB correct,
- both models wrong.

For each group, compute the mean KL divergence in both directions. How does KL divergence correlate with model accuracy and disagreement?

3. **Per-class disagreement.** For each true digit $c \in \{0, \dots, 9\}$, compute the mean $D_{\text{KL}}(p_{\text{NN}} \| p_{\text{NB}})$ over all test images with label c and visualize the ten values as a bar plot. Which digits show the strongest disagreement between models? Relate your findings to the confusion matrices from Fig. 6.3.
4. **Optional: inspect high- KL examples.** Extend the notebook to select a small set of images with the largest values of $D_{\text{KL}}(p_{\text{NN}} \| p_{\text{NB}})$. For each such image, display:

- the input image,
- the top predicted probabilities from both models.

Qualitatively describe what types of digits lead to large KL divergence (e.g., visually ambiguous digits, digits with unusual stroke patterns, etc.).

KL divergence as an optimization objective. Beyond post-hoc comparison of predictors, KL divergence also appears directly in training objectives. In normalizing flows (Section 5.2), one minimizes a KL divergence between the data distribution and a transformed base distribution, leading to loss functions of the form

$$\mathcal{L}(\theta) = \mathbb{E}_{Y \sim P_Y} [g_\theta(Y)^2 + \log|J_{g_\theta}(Y)|], \quad (6.9)$$

where g_θ is a learned transform and $J_{g_\theta}(Y)$ is its Jacobian determinant. Later chapters will connect these KL-based objectives to maximum likelihood and cross-entropy in generative models.

6.2.4 Cross-Entropy and Its Connection to KL Divergence

The **cross-entropy** between two discrete distributions P and Q (with common support \mathcal{X}) is defined as

$$H(P, Q) = - \sum_{x \in \mathcal{X}} P(x) \log Q(x). \quad (6.10)$$

It quantifies how “surprised” we would be, on average, if data generated from the true distribution P were encoded or predicted using the model distribution Q .

Cross-entropy is intimately connected to KL divergence. Using the entropy of P ,

$$H(P) = - \sum_x P(x) \log P(x),$$

we have the identity

$$D_{\text{KL}}(P \| Q) = H(P, Q) - H(P). \quad (6.11)$$

Since $H(P)$ does not depend on Q , minimizing the cross-entropy $H(P, Q)$ with respect to Q is equivalent to minimizing the KL divergence $D_{\text{KL}}(P \| Q)$. This is the conceptual reason why **cross-entropy is the standard loss function in modern classification and generative modeling**.

Why “cross”-entropy? Entropy $H(P)$ measures the intrinsic uncertainty of a *single* distribution P . Cross-entropy $H(P, Q)$ combines two distributions:

- the true distribution P ,
- the predictive or model distribution Q .

It asks: *How many bits would we need to encode samples from P using a code optimized for Q instead of P ?* This mismatch cost is exactly the KL divergence plus the irreducible entropy of P .

Cross-entropy in modern AI. Cross-entropy appears in several core learning pipelines:

- **Neural networks.** For a one-hot label y and predictive distribution \hat{p} , the loss

$$\mathcal{L}_{\text{CE}} = - \sum_i y_i \log \hat{p}_i$$

promotes high probability on the correct class.

- **Language models.** Next-token prediction is trained with cross-entropy, equivalent to maximum likelihood.
- **Normalizing flows.** Flow models minimize KL divergence between the data distribution and a transformed base distribution; this reduces to a cross-entropy term plus a Jacobian correction.
- **Importance sampling and adaptive sampling.** Cross-entropy is central to the *cross-entropy method* of Rubinstein, where one iteratively updates a parametric distribution by minimizing cross-entropy to a set of high-reward samples.

Empirical cross-entropy vs. KL divergence. Cross-entropy has a crucial practical advantage over KL divergence:

- KL divergence $D_{\text{KL}}(P \parallel Q)$ requires evaluating $\log P(x)$, which is impossible when P is available only through samples (empirical distribution).
- Cross-entropy $H(P_{\text{emp}}, Q) = -\frac{1}{N} \sum_{i=1}^N \log Q(x_i)$ is always well-defined, because it involves only $\log Q(x)$.

Thus cross-entropy provides a stable, tractable substitute for KL when the “true” distribution is empirical (a sum of delta functions). This is why maximum-likelihood training of neural networks reduces to minimizing empirical cross-entropy.

Exercise 6.2.5 (Cross-Entropy, Entropy, and KL Divergence). *This exercise connects analytic calculations with empirical evaluations using the distributions saved by the notebook Bayes-MNIST-NaiveBayes.ipynb.*

1. **Entropy calculations.** Compute the entropy of:

- a uniform distribution over 4 categories;
- a “peaked” 4-category distribution $(0.9, 0.0333, 0.0333, 0.0333)$;
- the Gaussian $N(0, 1)$ (use the analytic formula).

2. **KL divergence calculations.** Compute:

- $D_{\text{KL}}(N(\mu_1, \sigma^2) \parallel N(\mu_2, \sigma^2))$;
- $D_{\text{KL}}(\text{Laplace}(0, 1) \parallel N(0, 1))$ numerically on a grid.

3. **Empirical cross-entropy on MNIST.** Using the predictive distributions saved in the notebook:

- compute the empirical cross-entropy of the Naïve Bayes predictions relative to the true labels;
- compute the empirical cross-entropy of the neural network predictions relative to the true labels;
- compare these values and relate them to the confusion matrices in Fig. 6.3.

4. **Cross-entropy vs. KL divergence (empirical).** For each test image F , compute

$$-\log p_{\text{NN}}(y \mid F), \quad -\log p_{\text{NB}}(y \mid F),$$

where y is the true label. Compare the distributions of these two per-sample losses to the KL-based disagreement analysis from Subsection 6.2.3. Discuss:

- why cross-entropy can be computed directly from true labels,
- why KL between model predictions requires two full distributions,
- how the two diagnostics provide complementary information.

5. **Optional: replacing cross-entropy by KL in training.** Modify the MNIST neural network so that it is trained with the loss

$$\mathcal{L}_{\text{KL}} = D_{\text{KL}}(y_{\text{onehot}} \parallel p_{\text{NN}}(\cdot \mid F)),$$

which is numerically identical to cross-entropy. Verify this equivalence in code and compare learning curves.

6.2.5 Wasserstein Distance: Geometry of Probability Distributions

KL divergence and cross-entropy quantify how one distribution differs *informationally* from another. But there are important situations where these quantities behave poorly:

- KL divergence becomes infinite when the supports do not overlap;
- small translations of a distribution can cause large KL divergence;
- neither KL nor cross-entropy reflect the *geometry* of the space in which samples live.

The **Wasserstein distance** – also called **Earth-Mover Distance** (EMD) – addresses these limitations by measuring the *cost of transporting mass* from one distribution to another. Rather than comparing probabilities pointwise, Wasserstein distance compares distributions according to how difficult it is to move probability mass across the underlying space.

Definition (1-Wasserstein distance). For two probability distributions P and Q on \mathbb{R}^d ,

$$W_1(P, Q) = \inf_{\gamma \in \Gamma(P, Q)} \mathbb{E}_{(X, Y) \sim \gamma} [\|X - Y\|],$$

where $\Gamma(P, Q)$ is the set of all couplings (joint distributions) with marginals P and Q . Intuitively: $W_1(P, Q)$ is the minimum amount of “work” required to transport mass from P to match Q , using Euclidean distance as the cost of moving unit mass.

Higher-order Wasserstein distances W_p are defined analogously.

Why Wasserstein? A geometric metric. Wasserstein distances:

- are true metrics on distributions;
- remain finite even when supports do not overlap;
- behave smoothly under translations and deformations of distributions;
- incorporate the geometry of data (important for images, audio, embeddings).

In modern AI this geometric sensitivity is crucial:

- *Generative Adversarial Networks* (GANs) utilize Wasserstein distance as a metric to avoid gradient collapse;
- *Diffusion models* implicitly minimize Wasserstein-type objectives;
- *Optimal transport (OT)* provides the mathematical foundation for alignment, matching, barycenters, and flow-based generative models.

Thus Wasserstein distance is the “right” metric whenever one cares about *how* distributions differ in space, not just *whether* they differ.

Closed-form case: 1D and Gaussian measures. For one-dimensional distributions with cumulative distribution functions F_P and F_Q ,

$$W_1(P, Q) = \int_0^1 |F_P^{-1}(u) - F_Q^{-1}(u)| du.$$

For Gaussian distributions in \mathbb{R}^d ,

$$W_2^2(\mathcal{N}(m_1, \Sigma_1), \mathcal{N}(m_2, \Sigma_2)) = \|m_1 - m_2\|^2 + \text{Tr}\left(\Sigma_1 + \Sigma_2 - 2(\Sigma_2^{1/2} \Sigma_1 \Sigma_2^{1/2})^{1/2}\right),$$

reflecting both the shift in means and the mismatch in covariance geometry. These closed forms motivate the following example.

Example 6.2.4 (Wasserstein Distance Between Simple Distributions). Consider three Gaussian distributions on \mathbb{R} :

$$P_1 = \mathcal{N}(0, 1), \quad P_2 = \mathcal{N}(2, 1), \quad P_3 = \mathcal{N}(0, 4).$$

For one-dimensional distributions, the W_1 (earth-mover) distance admits the closed-form quantile formula

$$W_1(P, Q) = \int_0^1 |F_P^{-1}(u) - F_Q^{-1}(u)| du.$$

Applying it to the three Gaussians gives

$$W_1(P_1, P_2) = 2, \quad W_1(P_1, P_3) = \frac{2}{\sqrt{\pi}} \approx 1.128.$$

These values highlight an essential geometric aspect of Wasserstein distance:

- Shifting a Gaussian by 2 units requires transporting mass by exactly 2, hence $W_1(P_1, P_2) = 2$.
- Increasing variance spreads mass but does not require a uniform shift, resulting in a smaller cost for $W_1(P_1, P_3)$.

By contrast, KL divergence reacts strongly to changes in variance, so it judges P_3 to be far more dissimilar from P_1 than P_2 — the opposite of the Wasserstein ordering.

The accompanying notebook `Wasserstein-1D-Gaussians.ipynb`:

- samples from each distribution,
- computes empirical Wasserstein distances using `scipy`,
- compares empirical estimates to the theoretical values above,
- visualizes:
 1. histograms of samples,
 2. quantile functions $F^{-1}(u)$,
 3. absolute quantile gaps $|F_P^{-1}(u) - F_Q^{-1}(u)|$ whose integral is W_1 .

Representative outputs are shown in Fig. (6.9).

Exercise 6.2.6 (Exploring Wasserstein Geometry). Using the notebook `Wasserstein-1D-Gaussians.ipynb`

1. Compute empirical $W_1(P_i, P_j)$ for all three pairs and compare to theoretical values.
2. Plot the quantile functions $F_{P_i}^{-1}(u)$ and the absolute transport distance $|F_{P_i}^{-1}(u) - F_{P_j}^{-1}(u)|$.
3. Replace one Gaussian by a heavy-tailed Laplace distribution. How does W_1 compare to KL divergence in this case?
4. (**Advanced**) Fit a small neural network that transports samples from P_1 to P_2 by minimizing empirical W_1 . Visualize how the learned map behaves.

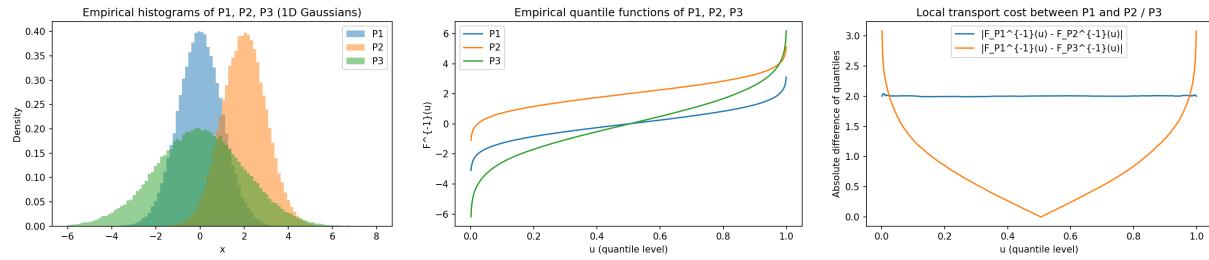


Figure 6.9: Left: Histograms of samples from P_1 , P_2 , and P_3 . The shift in the mean of P_2 and the increased spread of P_3 are clearly visible. Center: Quantile functions of the three Gaussians. In 1D, the Wasserstein distance equals the area between these curves. The constant horizontal shift between P_1 and P_2 explains why $W_1(P_1, P_2) = 2$. Right: Pointwise transport cost $|F_{P_i}^{-1}(u) - F_{P_j}^{-1}(u)|$ ($i = 1, 2, 3$). The shaded area under each curve equals the Wasserstein distance. The cost curve for the variance change (P_3) is smaller overall, matching the fact that $W_1(P_1, P_3) < W_1(P_1, P_2)$.

Forward Look: Optimal Transport and Flow Matching

The Wasserstein distance discussed here is only the beginning of a broader story. Optimal transport (OT) provides a geometric framework for comparing and transforming probability distributions, and its ideas reappear in increasingly sophisticated forms throughout modern generative modeling.

In the *Synthesis Chapter* 9, we return to OT in two deeper contexts:

- **Continuous-time OT maps and flows**, which offer principled ways to morph one distribution into another through dynamical systems;
- **Flow matching**, a recent family of generative-model training objectives that align learned vector fields with OT-inspired transport flows.

These constructions reveal how classical OT geometry underlies diffusion models, normalizing flows, and score-based generative methods. The simple 1D examples in this chapter serve as a conceptual anchor for the more general and powerful OT machinery developed later.

6.3 Information Theory and Neural Networks

Information theory provides a rigorous language for describing how neural networks *store*, *compress*, *transmit*, and *transform* information. Although originally developed for communication systems, Shannon's framework now underpins many of the most important principles in modern deep learning, from representation learning and model capacity to generalization and robustness.

This section develops the interplay between **entropy**, **compression**, and **expressivity** in neural networks. We connect the information-theoretic quantities introduced in the previous section (entropy, KL divergence, cross-entropy, mutual information, Wasserstein distance)

to the mechanics of neural networks as learning and inference systems.

Shannon's foundational theorems—the **Source Coding Theorem** and the **Channel Coding Theorem**—establish fundamental limits on data compression and reliable communication. These limits have concrete implications for deep learning: how efficiently networks can encode inputs, how much information must pass between layers, and what constraints govern representation bottlenecks.

In the subsections that follow, we examine:

- how entropy and compression relate to the capacity and architecture of neural networks;
- how mutual information illuminates the role of intermediate representations;
- how information bottlenecks (explicit and implicit) shape generalization;
- how modern generative models leverage information-theoretic objectives.

The goal is not only to present classical theorems, but to show how they *actively structure the learning dynamics and representational geometry of neural networks*.

6.3.1 Source Coding Theorem (Lossless Compression)

The **Source Coding Theorem** — Shannon's fundamental result on lossless compression — establishes the ultimate limit on how efficiently data from a probabilistic source can be encoded.

For a discrete memoryless source producing a random variable X with Probability Mass Function (PMF) $P(X)$, the *minimum achievable average code length per symbol* under any lossless encoding scheme is asymptotically bounded below by the **Shannon entropy**

$$H(X) = - \sum_x P(x) \log_2 P(x).$$

In other words, no lossless compression algorithm can achieve an average rate smaller than $H(X)$ bits per symbol, and conversely, Shannon proved that one *can* in principle construct codes whose rate approaches $H(X)$ arbitrarily closely when encoding long sequences of i.i.d. samples.

Compression viewpoint. Entropy thus quantifies the *irreducible information content* of the source. Any redundancy in the data (non-uniform probabilities, correlations, structure) can be exploited by a good encoder to reduce the average number of bits needed. But once all redundancy has been removed, the entropy barrier $H(X)$ cannot be crossed without introducing loss.

Asymptotic and non-constructive nature. While the theorem characterizes the optimal rate, it is not itself a coding algorithm. It guarantees existence but does not provide explicit codes, and its optimality statements hold only in the limit of infinite sequence length. Practical compression methods — e.g., Huffman coding, arithmetic coding, and dictionary-based schemes such as Lempel–Ziv — approximate the theoretical optimum for finite sequences.

Example 6.3.1 (Huffman Coding and Near-Optimal Compression). Consider a source that emits four symbols $\{A, B, C, D\}$ with probabilities

$$P(A) = 0.5, \quad P(B) = 0.25, \quad P(C) = 0.15, \quad P(D) = 0.1.$$

A fixed-length binary code requires 2 bits per symbol. A Huffman code assigns shorter codes to more probable symbols:

$$A \rightarrow 0, \quad B \rightarrow 10, \quad C \rightarrow 110, \quad D \rightarrow 111.$$

The expected code length becomes

$$L = 0.5(1) + 0.25(2) + 0.15(3) + 0.1(3) = 1.75 \text{ bits.}$$

The entropy of the source is

$$H(X) = - \sum_x P(x) \log_2 P(x) \approx 1.72 \text{ bits.}$$

Thus the Huffman code comes very close to Shannon's theoretical lower bound, illustrating how entropy quantifies the best possible lossless compression.

Classical vs. Learned Compression

Classical source coding constructs explicit prefix-free bit strings that minimize average code length. In contrast, **neural networks perform learned compression**:

- autoencoders learn low-dimensional latent representations that approximately satisfy the spirit of source coding,
- Variational Autoencoders (VAEs) explicitly trade off compression and reconstruction using KL divergence,
- modern generative models learn latent spaces whose dimensionality and entropy reflect the underlying data complexity.

Although the mechanisms differ, the organizing principle is the same: *entropy places fundamental limits on how concisely data can be represented*. Later subsections return to this viewpoint when discussing the information bottleneck, latent-variable models, and generative compression.

Exercise 6.3.1 (Designing and Evaluating Optimal Codes). 1. Compute the entropy $H(X)$ for the four-symbol distribution in the example.

2. Construct the Huffman code and verify its expected length L .
3. Modify the distribution (e.g. make it more skewed or more uniform) and repeat the entropy and code-length calculations. How does the gap $L - H(X)$ behave?
4. For a longer discrete source where the alphabet is moderately large (e.g. 20 symbols), numerically generate random pmfs and compare Huffman code lengths to entropy across many trials. What empirical patterns emerge?

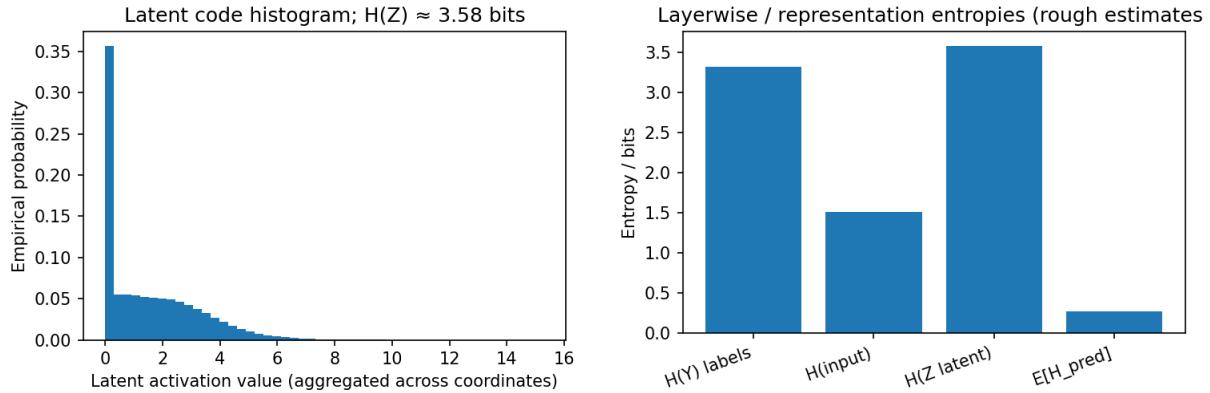


Figure 6.10: Left: Histogram of latent activations in the final hidden layer and empirical entropy estimate $H(Z)$ obtained by binning (Gaussian Model Mixture). The entropy typically decreases compared to raw inputs, indicating task-dependent compression. Right: Layerwise entropy estimates $H(Z_i)$ for a small MNIST CNN. Early layers retain high entropy (broad feature variability), while deeper layers compress toward the label entropy $H(Y)$, consistent with a learned encoding pipeline.

6.3.2 Neural Networks as Encoding Schemes

Neural networks can be understood not only as function approximators – the viewpoint of Chapter 4 – but also as **learned encoding schemes** in the sense of Shannon’s source coding theory. A standard feed-forward classifier performs a sequence of transformations

$$X \longrightarrow Z \longrightarrow Y,$$

where X is the raw input, Z is an internal representation (latent encoding), and Y is the predicted label distribution. From an information-theoretic perspective, the entropy $H(Z)$ measures how much information the network retains about the input after encoding it at a given layer.

For a well-trained classifier, one expects that the deepest representation extracts precisely the information needed for classification:

$$H(Z_{\text{deep}}) \approx H(Y),$$

so that the representation is neither too redundant (overly large entropy) nor too compressed (discarding task-relevant information).

This new viewpoint aligns neural networks with classical source coding: the network acts as a learned encoder whose latent space Z must preserve just enough information to determine the label Y , analogous to optimal lossless compression constrained by the downstream task.

Example 6.3.2 (Estimating Encoding Entropy in a Neural Network). *Consider a small CNN trained on MNIST. For each test image:*

1. Extract latent vectors from a designated hidden layer.
2. Discretize the activations (e.g., via binning or by fitting a Gaussian mixture model).

3. Estimate the entropy

$$H(Z) = - \sum_z P_Z(z) \log_2 P_Z(z).$$

4. Compare $H(Z)$ to $H(Y)$. Well-trained models typically satisfy $H(Z) \approx H(Y)$ in deep layers, while earlier layers retain superfluous variability.

The `entropyNN.ipynb` notebook automates this workflow, producing Figs. 6.10.

Exercise 6.3.2 (Layer-wise Entropy and Information Flow in Neural Networks). Train a small CNN on MNIST or CIFAR-10 and analyze its internal representations as follows:

1. **Activation extraction:** For a fixed test set, record activations from each hidden layer Z_1, Z_2, \dots, Z_L .
2. **Entropy estimation:** Discretize each activation tensor (per neuron or jointly via PCA) and compute empirical entropies $H(Z_i)$.
3. **Comparison with label entropy:** Compute the class entropy $H(Y)$. Identify layers for which $H(Z_i)$ approaches $H(Y)$.
4. **Analysis:** Plot $H(Z_i)$ vs. depth. Discuss how compression emerges across layers, and relate your findings to generalization and robustness.

Compression, Redundancy, and Generalization

Neural networks walk a fine line between two extremes:

- **Redundancy** (under-compression): Large $H(Z)$ means the network preserves too much input variability, increasing the risk of overfitting.
- **Over-compression:** If $H(Z)$ becomes smaller than $H(Y)$, task-relevant information may be destroyed, harming accuracy.
- **Effective encoders:** Good classifiers typically compress representations so that $H(Z_{\text{deep}}) \approx H(Y)$, echoing the optimality principle of Shannon's source coding theorem.

This perspective also underlies the *information bottleneck* principle and sets the stage for later discussions of variational autoencoders (VAEs) and diffusion models.

6.3.3 Autoencoders and Nonlinear Compression

Classical dimensionality-reduction methods such as PCA or SVD provide *linear* compression: they represent data as linear combinations of a small number of orthogonal directions. Many datasets encountered in modern machine learning—images, speech, trajectory data, molecular conformations—contain nonlinear geometric structure that cannot be captured efficiently by linear projections.

A **nonlinear autoencoder** addresses this limitation by learning an *encoder–decoder pair*:

$$X \longrightarrow Z \longrightarrow \hat{X},$$

where:

- the **encoder** maps a high-dimensional input X to a compressed latent representation Z ,
- the **decoder** reconstructs an approximation \hat{X} of the original input.

The reconstruction loss is typically

$$\mathcal{L}_{\text{AE}} = \mathbb{E}_X \left[\|X - \hat{X}\|^2 \right],$$

and the user chooses the *bottleneck size* $\dim(Z)$, which determines the compression ratio. From an information-theoretic point of view, an autoencoder is a *learned compression scheme*. The latent entropy $H(Z)$ provides a quantitative measure of how much information the autoencoder retains after compression:

$$H(Z) \approx \text{bits needed to represent the latent code } Z.$$

A smaller bottleneck or stronger regularization often leads to smaller entropy — but may also degrade reconstruction quality.

Example 6.3.3 (Entropy of Autoencoder Latent Codes). *The accompanying notebook `autoencoder-entropy.ipynb` trains a small autoencoder on MNIST with a configurable bottleneck dimension.*

For a trained model, the notebook performs:

1. **Encoding.** Compute latent vectors $Z = f_{\theta}(X)$ for the whole test set.
2. **Discretization.** Each latent coordinate is binned into a small number of intervals (e.g., 20–50 bins per dimension), producing a discrete empirical distribution P_Z .
3. **Entropy estimation.** The latent entropy is approximated by:

$$H(Z) \approx - \sum_z P_Z(z) \log_2 P_Z(z).$$

4. **Comparison across bottleneck sizes.** The notebook repeats this procedure for

$$\dim(Z) \in \{8, 16, 32, 64\}.$$

This produces the curve $\dim(Z) \mapsto H(Z)$, which illustrates how compression strength controls the information content of the representation.

Representative outputs appear in Figs. 6.11.

This experiment makes the abstract quantity $H(Z)$ concrete and shows how autoencoders perform nonlinear compression in practice.

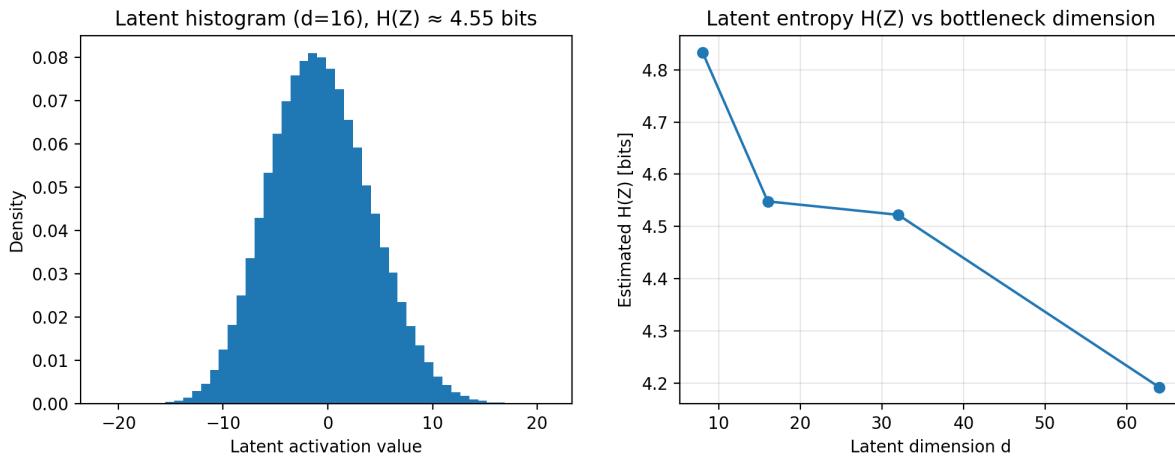


Figure 6.11: Left: Histogram of latent activations for a trained autoencoder (for a chosen bottleneck size), together with the estimated latent entropy $H(Z)$. Increasing compression tends to concentrate activations and reduce entropy. Right: Estimated latent entropy $H(Z)$ as a function of bottleneck dimension. Larger bottlenecks retain more information; smaller bottlenecks enforce stronger compression.

Exercise 6.3.3 (Entropy and Compression in Autoencoders). *Using the notebook `autoencoder-entropy.ipynb`:*

1. Train autoencoders with bottleneck dimensions $d \in \{8, 16, 32, 64\}$.
2. For each model, extract latent representations Z of the MNIST test set and estimate the latent entropy $H(Z)$.
3. Plot $H(Z)$ as a function of d , as in Fig. 6.11.
4. Compare reconstruction errors across bottleneck sizes. How does the trade-off between compression and reconstruction quality manifest?
5. Discuss whether the autoencoder is overcompressing (too little information retained) or undercompressing (latent space carries unnecessary redundancy).

From Autoencoders to Variational and Flow-Based Models

Autoencoders illustrate the core idea of *learned compression*. In later chapters we will see two major generalizations:

- **Variational Autoencoders (VAEs):** introduce an explicit probabilistic model for latent variables Z , with KL divergence controlling information flow.
- **Flow-based Models and Diffusions:** treat the encoder as an invertible map, enabling exact likelihoods and connecting learned representations to optimal transport.

Both perspectives refine and extend the basic compression viewpoint developed in this subsection.

6.3.4 The Information Bottleneck Principle and U-Net as a Non-linear Compressor

A unifying perspective on feature extraction and representation learning in neural networks is provided by the **Information Bottleneck** (IB) principle [35]. Given an input X , an encoded representation Z , and a task variable Y , the IB objective seeks a representation that is both *minimal* (removing irrelevant information about X) and *sufficient* (retaining task-relevant structure):

$$\min_{p(z|x)} I(X; Z) - \beta I(Z; Y),$$

where $I(\cdot; \cdot)$ denotes mutual information and $\beta > 0$ controls the trade-off between compression and predictive usefulness. Although computing $I(X; Z)$ and $I(Z; Y)$ exactly is challenging in high dimensions, the IB principle provides an intuitive conceptual framework: *good representations discard nuisance variability while preserving the information needed for the task.*

U-Net as an Architectural Information Bottleneck. The **U-Net** architecture [36], originally proposed for biomedical image segmentation, offers a vivid architectural example of the bottleneck principle in action. A U-Net consists of three main components:

- a **contracting path** (encoder) that repeatedly downsamples and compresses spatial structure,
- a **bottleneck layer** of reduced spatial extent and increased channel depth,
- an **expanding path** (decoder) that upsamples and combines compressed features with high-resolution skip connections from the encoder.

The encoder compresses X into a low-resolution latent representation Z , enforcing an implicit information bottleneck. The skip connections mitigate excessive information loss by reintroducing fine spatial detail during decoding. Thus, a U-Net simultaneously exhibits the two competing pressures emphasized by IB: *compress aggressively in the bottleneck, yet preserve task-relevant information* needed for accurate reconstruction or segmentation.

Example 6.3.4 (U-Net Compression Effects on MNIST Reconstruction). *The accompanying notebook `UNet-MNIST-light.ipynb` trains two lightweight U-Nets on MNIST with significantly different compression strengths:*

$$\text{depth} = 1 \quad (\text{weak compression}), \quad \text{depth} = 3 \quad (\text{strong spatial bottleneck}).$$

The difference in performance, particularly in discarded information, is visualized using the Absolute Error Map $|X - \hat{X}|$, where \hat{X} is the reconstruction.

Compression Discard: Absolute Error Maps

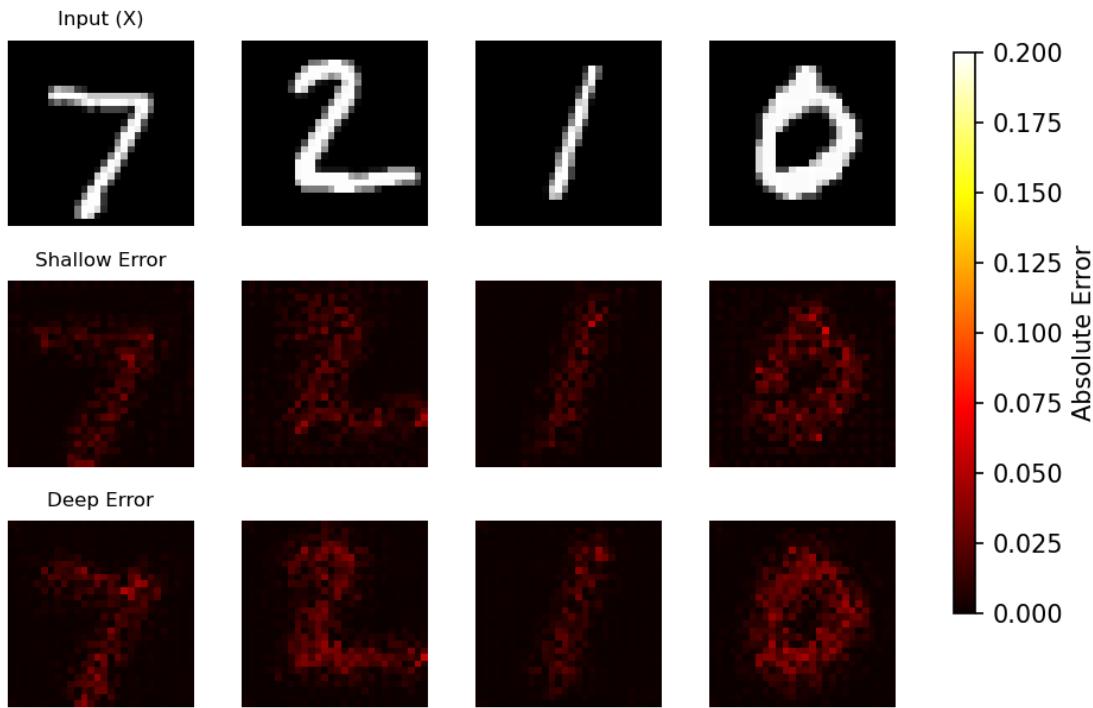


Figure 6.12: Absolute Error Maps ($|X - \hat{X}|$) for two U-Net variants. The top row shows the input X . The middle row shows the error for the shallow (depth = 1) U-Net, which is minimal (less red). The bottom row shows the error for the deep (depth = 3) U-Net, which is significantly higher (more red) and concentrated around the digit boundaries, confirming that the strong bottleneck successfully discarded high-frequency spatial information.

1. **Shallow U-Net (depth = 1):** With no spatial downsampling, the error is minimal, confirming little information loss.
2. **Deep U-Net (depth = 3):** Aggressive spatial downsampling (e.g., to 7×7 features) enforces strong compression, resulting in non-zero error concentrated at the sharp boundaries.
3. **Compression Proxy:** The estimated entropy $H(Z)$ of the latent space (a proxy for $I(X; Z)$) is measurably lower in the depth = 3 model.

Representative results are shown in Fig. 6.12. The depth = 3 network's stronger bottleneck visibly discards more information, producing higher error concentrated on high-frequency details (edges and corners) that are most difficult to represent in a small feature map. This behavior aligns with the IB principle: increasing the strength of the bottleneck reduces $I(X; Z)$, though at the cost of reconstruction fidelity.

Exercise 6.3.4 (Information Bottleneck Behavior in U-Net Architectures). Using the notebook `UNet-MNIST-light.ipynb`:

1. **Vary compression strength:** Train U-Nets of depth $d \in \{1, 2, 3\}$ and record reconstruction losses.
2. **Analyze latent statistics:** Extract the bottleneck activations for each model and plot activation histograms and variances as rough proxies for compression (smaller variance \Rightarrow stronger compression).
3. **Assess IB trade-offs:** For each depth, discuss qualitatively how a proxy for $I(X; Z)$ decreases as compression strengthens, while reconstruction quality (a proxy for $I(Z; Y)$) deteriorates.
4. **Role of skip connections:** Remove or thin out skip connections and examine how reconstruction quality changes. How do skip connections balance compression with preservation of spatial detail?

6.3.5 Channel Coding Theorem and Its Application to Neural Networks

Claude Shannon's **Channel Coding Theorem** establishes a fundamental limit on reliable communication over a noisy channel. If information is transmitted at rate R (bits per channel use) through a channel of capacity

$$C = \max_{P(X)} I(X; Y),$$

then reliable decoding is possible if and only if $R < C$. Above capacity, errors are unavoidable no matter how clever the code.

This viewpoint is surprisingly fruitful when thinking about how information flows through deep neural networks. Each hidden layer can be treated as a *communication channel* transmitting information about the input X toward the target Y :

$$X \rightarrow Z_1 \rightarrow Z_2 \rightarrow \dots \rightarrow Z_L \rightarrow Y.$$

Key analogies:

- **Layer-by-Layer Transmission:** Each hidden layer Z_i passes on a compressed description of X . If compression is too aggressive (a narrow bottleneck), then the next layer cannot reliably recover the features needed for prediction.
- **Noise, Dropout, and Stochasticity:** Regularizers such as dropout act as injected noise, reducing the “effective capacity” of the layer—just as physical noise reduces channel capacity in communication systems.
- **Bottleneck Geometry:** Autoencoders, U-Nets, and classification CNNs all impose information bottlenecks. The channel coding theorem reminds us that these bottlenecks have a *maximum rate* beyond which information simply cannot pass reliably.

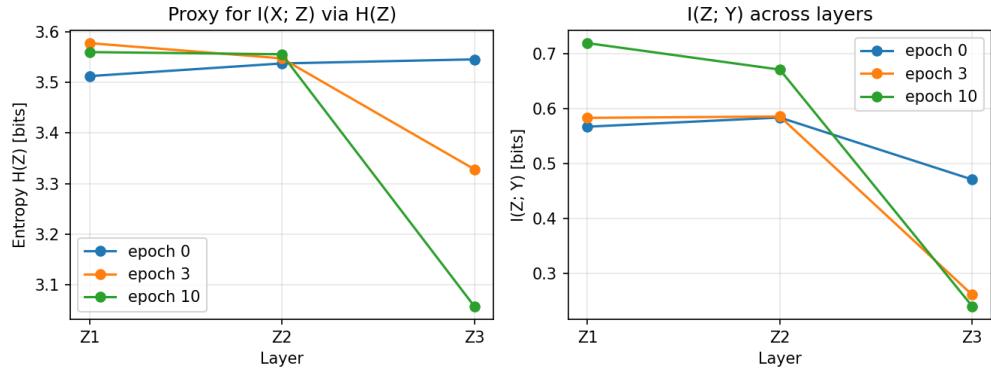


Figure 6.13: Information flow across layers in the MNIST CNN from Example 6.3.5. The left panel shows the entropy $H(Z_i)$ of scalar summaries of the intermediate layers Z_1, Z_2, Z_3 at epochs 0, 3, 10, serving as a rough proxy for $I(X; Z_i)$. The right panel shows the estimated mutual information $I(Z_i; Y)$ with the labels. As training proceeds, deeper layers compress (entropy decreases) while becoming more predictive (larger $I(Z_i; Y)$), illustrating a capacity-like bottleneck effect analogous to channel coding.

Information as a Conserved Quantity

From the channel-coding perspective, the “job” of a neural network is not to preserve *all* information about the input, but rather to preserve precisely the subset relevant for the task. Training gradually pushes intermediate representations Z_i toward states that carry high mutual information with Y , while shedding irrelevant variability in X . This creates a natural lens through which to view the dynamics of representation learning.

Example 6.3.5 (Information Flow in a CNN Viewed as a Noisy Channel). *To make this analogy concrete, we consider a small convolutional neural network trained on MNIST, instrumented so that the intermediate activations*

$$Z_1, Z_2, Z_3$$

are available during evaluation. We approximate the entropy $H(Z_i)$ (as a proxy for $I(X; Z_i)$) and the mutual information $I(Z_i; Y)$, using histogram-based estimators applied to simple scalar summaries of each layer.

The accompanying Jupyter notebook `CNN-MNIST-channel.ipynb` provides a full implementation: training, activation extraction, entropy estimation, mutual information diagnostics. The output of the notebook—summarized in Fig. 6.13 — reveals several characteristic information-flow patterns:

- **Early layers** exhibit relatively high entropy, reflecting sensitivity to many fine-grained pixel-level variations present in the input.
- **With increasing depth**, the representations undergo visible compression: the entropy $H(Z_i)$ decreases across layers. At the same time, the mutual information with the

labels, $I(Z_i; Y)$ increases from epoch 0 to epoch 10, indicating that deeper layers discard irrelevant input variability while increasingly aligning their representations with the task-relevant structure.

- **Dropout as channel noise:** repeating the experiment with larger dropout rates reduces the achievable $I(Z_i; Y)$, demonstrating a clear “capacity-like” limitation: noisier layers function as lower-capacity channels, restricting the amount of label-relevant information that can be reliably transmitted downstream.

This experiment operationalizes the channel coding theorem intuition: a layer of limited capacity (narrow, noisy, or both) cannot transmit arbitrary amounts of information, but it can be trained to transmit precisely the information relevant for classification.

Exercise 6.3.5 (Tracking Mutual Information Through Layers and Epochs). Using the provided notebook `CNN-MNIST-channel.ipynb`, carry out the following steps:

1. Train the supplied CNN for 10 epochs with dropout $p = 0.3$.
2. Extract the intermediate activations Z_1, Z_2, Z_3 at epochs 0, 3, 10 and estimate:

$$H(Z_i), \quad I(Z_i; Y),$$

using the histogram-based estimators implemented in the notebook.

3. Plot and compare the curves (already generated by the notebook):

$$H(Z_i) \quad \text{and} \quad I(Z_i; Y)$$

across layers and epochs. Does the entropy generally decrease with depth? Does $I(Z_i; Y)$ increase, indicating a sharpening of class-specific information?

4. Repeat the experiment with a different dropout rate (e.g. $p = 0.0$ or $p = 0.5$). Compare how the noise level affects the ability of deeper layers to preserve predictive information. Is there evidence of a “capacity limit” analogous to Shannon’s theorem?

Relate your findings to the communication-channel view of neural networks. In particular, interpret where in your model the “bottleneck” lies and how training adapts the internal code to operate below the effective channel capacity.

6.3.6 Efficient Memory and Neural Network Storage

Efficient memory and robust retrieval are essential for understanding both *generalization* and *memorization* in neural networks. A well-trained network must faithfully store task-relevant structure while discarding irrelevant detail. This tension between **memorization** and **compression** mirrors classical communication systems, where reliable transmission requires coding schemes adapted to the channel’s limited capacity.

Memorization vs. Compression as an Information-Theoretic Trade-Off

Over-parameterized networks can memorize the entire training set — including idiosyncratic noise — yet successful generalization requires compressing away most of this information. From an information-theoretic viewpoint, a neural network acts much like a channel encoder: it must map high-dimensional data to lower-capacity internal representations. Too much memorization leads to overfitting; too much compression destroys useful signal. Balancing the two is central to modern deep learning.

Associative Memory and Hopfield Networks. Classical **Hopfield networks** [37] provide an early model of associative memory, storing patterns as stable attractors in a dynamical system. For a binary state vector $\mathbf{x} \in \{-1, +1\}^n$ and weight matrix $W \in \mathbb{R}^{n \times n}$, the system evolves to minimize the energy

$$E(\mathbf{x}) = -\frac{1}{2} \mathbf{x}^\top W \mathbf{x} + \sum_i b_i x_i.$$

With **Hebbian learning** [38],

$$W_{ij} \propto \langle x_i x_j \rangle,$$

the network stores training patterns as energy minima. Given a corrupted input $\mathbf{x}^{(0)}$, the iterative update

$$x_i^{(t+1)} = \text{sgn}\left(\sum_j W_{ij} x_j^{(t)} + b_i\right)$$

drives the state toward a nearby attractor, recovering the original memory. This process is reminiscent of *error correction*: the dynamics undo a small amount of noise by converging to a stored pattern.

Modern Hopfield Networks. Recent work [39, 40] has generalized the Hopfield energy function to support exponentially many attractors, enabling associative retrieval modules to be embedded within contemporary deep networks. These modern Hopfield systems offer:

- high-capacity memory storage,
- fast content-based retrieval,
- and compatibility with architectures such as Transformers.

They can be viewed as adaptive lookup tables governed by energy minimization, blending neuroscience-inspired dynamics with modern AI computation.

Hebbian vs. Linear Codes. An enduring question is how associative memories compare with engineered error-correcting codes:

- **Hopfield networks** store patterns in distributed weights and retrieve them from partial or corrupted cues, but lack explicit guarantees on correction radius or worst-case decoding.

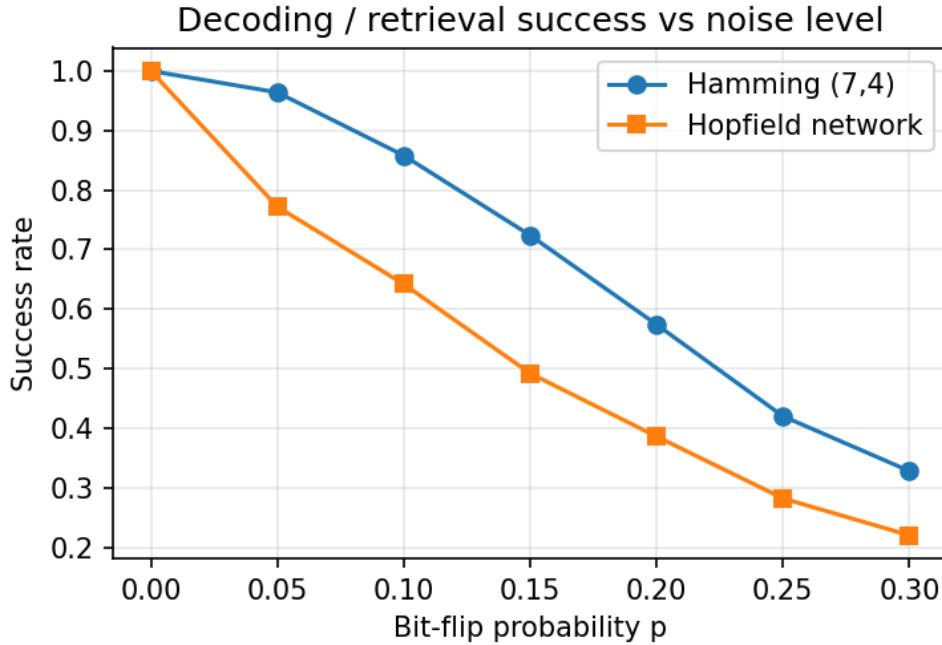


Figure 6.14: Retrieval success rates for the (7, 4) Hamming decoder (top curve) and a Hopfield network storing two corresponding 7-bit codewords (bottom curve), as a function of bit-flip noise level p . Hamming decoding maintains near-perfect accuracy for small noise levels due to its guaranteed single-error correction radius, whereas Hopfield retrieval succeeds frequently but degrades more rapidly with increasing noise.

- **Linear codes** (e.g., Hamming codes) provide formal distance guarantees and maximum-likelihood decoding, ensuring reliable correction under specific noise models.

Associative memory is flexible and brain-inspired; linear codes are rigid but provably reliable.

Example 6.3.6 (Encoding and Retrieval: Hamming Code vs. Hopfield Network). **Linear Code (Hamming).** The classical (7, 4) Hamming code maps a 4-bit message $\mathbf{m} \in \{0, 1\}^4$ into a 7-bit codeword $\mathbf{c} = \mathbf{m}G \pmod{2}$ using a generator matrix G . Decoding is performed by computing the syndrome $H\mathbf{c}$ via a parity-check matrix H , which identifies and corrects any single-bit error. Thus, for small noise levels, the Hamming code provides formal recovery guarantees.

Hopfield Associative Memory. A Hopfield network stores binary patterns as stable attractors of an energy function. Storing a 7-bit pattern $\mathbf{x}^* \in \{-1, +1\}^7$ amounts to forming

$$W = \mathbf{x}^*(\mathbf{x}^*)^\top,$$

(optionally normalized and with zero diagonal). Given a noisy initialization $\mathbf{x}^{(0)}$, the asynchronous update rule

$$x_i^{(t+1)} = \text{sgn}\left(\sum_j W_{ij} x_j^{(t)}\right)$$

drives the state toward a nearby attractor, thereby performing noise-correction. However, unlike the Hamming code, Hopfield retrieval has no rigorous correction radius: for larger

noise levels or when multiple patterns are stored, the dynamics may converge to the wrong attractor or a spurious fixed point.

Empirical Comparison. The accompanying notebook *Hopfield-vs-Hamming.ipynb* stores only two Hamming codewords in the Hopfield network (to avoid overloading the $n = 7$ system) and evaluates retrieval success under random bit-flip noise $p \in [0, 0.3]$. The results, summarized in Fig. 6.14, show:

- For **small noise**, Hamming decoding succeeds with probability nearly 1, while Hopfield retrieval also succeeds frequently but with visibly lower accuracy.
- As **noise increases**, Hamming decoding gradually degrades as multiple-bit errors become likely, whereas Hopfield performance declines more quickly due to limited attraction basins and the possibility of converging to the wrong attractor.
- The comparison illustrates the distinction between engineered error-correcting codes with explicit guarantees and associative memories that rely on energy-based dynamics without worst-case bounds.

Exercise 6.3.6 (Exploring Capacity and Noise Sensitivity in Hopfield Retrieval). *Using the notebook *Hopfield-vs-Hamming.ipynb*:*

1. Repeat the main experiment for different numbers of stored patterns $K \in \{1, 2, 3, 4\}$. For each K , construct a Hopfield network from K Hamming codewords and measure retrieval success under the same noise sweep $p \in [0, 0.3]$.
2. Plot the success curves for each K (overlaid) and compare: how does Hopfield performance deteriorate as more patterns are stored?
3. Estimate the “effective” attraction basin size for each K by identifying the largest p for which retrieval success is above 0.9.
4. Compare your findings with the classical Hopfield capacity $K \approx 0.138n$ for $n = 7$, and contrast this with the strict correction radius of the Hamming code.
5. Discuss: does increasing K cause new spurious attractors? How do empirical results relate to the absence of worst-case guarantees in Hopfield retrieval?

Chapter 7

Stochastic Processes

The preceding chapters developed the mathematical foundations that govern learning via optimization (Ch. 3) and information flow in neural networks (Ch. 4) with probabilistic modeling and latent variables (Ch. 5), and the information-theoretic structure of modern architectures (Ch. 6). These viewpoints emphasized that learning systems operate by *transforming, propagating, and compressing uncertainty*. Stochastic processes provide the natural mathematical language for describing such uncertainty. They are the backbone of sampling, inference, noise injection, model training, and generative mechanisms.

Stochastic processes arise throughout the modern Generative AI ecosystem:

- **Exact and approximate sampling** from distributions is the essence of GenAI, and e.g. central to auto-regressive models, VAEs, EBMs, and diffusion models.
- **Markov chains** underlie both classical Monte Carlo methods and modern architectures such as masked auto-regressive models and token-level Transformers, but it is also a key mathematical ingredient behind diffusion models.
- **Diffusion processes**, with roots from Brownian motion, they are the engines encoding the forward-noising and reverse-de-noising dynamics in score-based diffusion models.
- **Markov Chain Monte Carlo** of various kinds – including and Importance sampling – form the algorithmic bridge between structured (e.g. via graph) mid-size "physical" models and high-dimensional generative modeling.
- **Stochastic differential equations (SDEs)** and their time-reversal laws reappear in the foundations of generative diffusion, Langevin dynamics, energy-based models, and further down the road (discussed in the two last chapters of the book) in path-integral and optimal-transport formulations.

This chapter introduces these stochastic tools in a systematic and unified manner. Each subsection follows a structure already familiar from earlier chapters: concise mathematical development, a worked example with a supporting notebook and figure, and an exercise that extends or stress-tests the ideas. Throughout, short connector boxes highlight how each concept reappears in modern neural generative models and how it prepares us for the energy-based, Langevin, and score-driven frameworks of the following chapters.

Chapter Layout.

- **Section 7.1: Exact Sampling.** Inverse transform sampling and chain-rule sampling as the mathematical foundation of auto-regressive and flow-based models.
- **Section 7.2: Importance Sampling.** Reweighting, proposal mismatch, and effective sample size as precursors to variational inference and gradient-based samplers.
- **Section 7.3: Diffusion and Brownian Motion.** The heat equation, Brownian motion, and the stochastic calculus that underpins diffusion-based generative modeling.
- **Section 7.4: Markov Chains.** Finite-state stochastic dynamics, stationary distributions, and their connection to auto-regressive architectures and sequence models.
- **Section 7.5: MCMC.** Classical Markov-chain sampling algorithms and their limitations, motivating gradient-based and score-based alternatives.
- **Section 7.6: Beyond Markov – Auto-regressive Modeling.** Sequential, conditional, and non-Markovian models that directly connect to Transformers and next-token prediction.

By the end of this chapter, we will have assembled the full stochastic vocabulary required to understand generative diffusion processes, energy-based models, and the path-integral perspective developed in Chapters 8 and 9.

7.1 Exact Sampling

Sampling from probability distributions is a fundamental operation in statistics, Bayesian inference, and generative modeling of AI. **Exact Sampling** refers to methods that produce independent and identically distributed (i.i.d.) samples from a given target distribution without bias and in a finite number of steps – even though potentially exponential number of steps in the system size. Unlike approximate methods such as Markov Chain Monte Carlo (MCMC) – which become exact only asymptotically when the number of samples is sent to infinity – exact sampling techniques ensure that each sample is drawn precisely according to the specified distribution.

7.1.1 Inverse Transform Sampling

Inverse Transform Sampling (ITS) provides an exact and universal method for drawing samples from any one-dimensional distribution with a known cumulative distribution function (CDF). In Section 5.2.3 we saw how invertible maps can transport a simple Gaussian source into complex high-dimensional data distributions. ITS is the *one-dimensional* version of the same idea: instead of transporting a Gaussian, we transport a uniform random variable through the inverse CDF.

Let $U \sim \text{Uniform}(0, 1)$ and let F be the CDF of a target distribution. Then

$$X = F^{-1}(U)$$

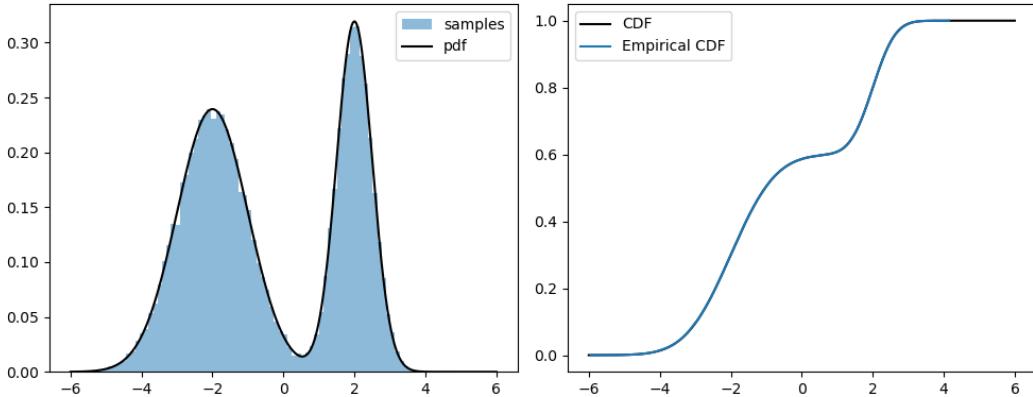


Figure 7.1: Inverse Transform Sampling for a two-component Gaussian mixture. *Left:* target pdf and histogram of ITS samples. *Right:* theoretical CDF $F(x)$ vs. empirical CDF of samples, illustrating exactness. All figures produced by `ITS-1D.ipynb`.

is an exact sample from that distribution. The source need not be Gaussian; any easily sampled base distribution with a well-defined CDF can be used.

Inverse Transform Sampling in Generative AI

In diffusion models, the forward process amounts to repeatedly sampling from Gaussian kernels — an exact operation akin to inverse-CDF sampling. ITS thus provides an instructive toy example of “known” sampling steps that make the forward diffusion analytically tractable. The reverse direction, learned from data, replaces F^{-1} with a neural score model.

Example 7.1.1 (Sampling from a Non-Gaussian Density via ITS). *Consider the Gaussian mixture*

$$p(x) = 0.6 \mathcal{N}(x; -2, 1) + 0.4 \mathcal{N}(x; +2, 0.5^2),$$

a simple non-Gaussian distribution exhibiting multi-modality. We numerically construct its CDF and obtain F^{-1} via interpolation. The ITS procedure is:

$$U \sim \mathcal{U}(0, 1), \quad X = F^{-1}(U).$$

The notebook `ITS-1D.ipynb` computes F , constructs F^{-1} on a fine grid, and generates 50,000 samples. Fig. 7.1 compares the theoretical density with the empirical histogram and shows the agreement between theoretical and empirical CDFs.

Exercise 7.1.1 (Inverse CDF Sampling for Arbitrary Densities). *Use the notebook `ITS-1D.ipynb` as a template.*

1. Replace the Gaussian mixture with a heavy-tailed distribution (e.g., Student- t). Numerically construct its CDF and inverse CDF.
2. Generate 100,000 samples and compare the empirical CDF to the theoretical CDF.

3. Quantify accuracy using either the Kolmogorov–Smirnov distance or KL divergence between the empirical histogram and the true density.

Comment on how numerical errors in F^{-1} interpolation affect sample quality.

Why is ITS only a 1D method? Inverse Transform Sampling relies on the fact that in one dimension the *CDF is an invertible scalar function*:

$$F(x) = \mathbb{P}(X \leq x) \Rightarrow X = F^{-1}(U), U \sim \mathcal{U}(0, 1).$$

The construction works because $F : \mathbb{R} \rightarrow [0, 1]$ is monotone and invertible almost everywhere. In higher dimensions, however, there is no canonical multivariate analogue of a CDF that yields an equally simple inverse map.

For a random vector $\mathbf{X} \in \mathbb{R}^d$ the CDF is

$$F(\mathbf{x}) = \mathbb{P}(X_1 \leq x_1, \dots, X_d \leq x_d),$$

which is a *scalar* function on \mathbb{R}^d . Such an F cannot be inverted to produce a d -dimensional sample: the map $F : \mathbb{R}^d \rightarrow [0, 1]$ collapses all d degrees of freedom into a single number. Recovering \mathbf{X} from $U \in [0, 1]$ would require an inverse map from a scalar to a vector, which is impossible without supplying additional structure.

Higher-dimensional ITS via the chain rule. Note that there is one natural way to generalize ITS to multivariate distributions: apply ITS to *each conditional distribution* in the chain-rule factorization

$$p(x_1, \dots, x_d) = p(x_1) p(x_2 | x_1) \cdots p(x_d | x_{1:d-1}).$$

Each 1D conditional distribution admits its own inverse CDF,

$$x_i = F_{X_i|X_{1:i-1}}^{-1}(u_i | x_{1:i-1}), \quad u_i \sim \mathcal{U}(0, 1),$$

and this produces an exact sampler.

This is precisely the mechanism underlying:

- auto-regressive flows (MAF, MADE);
- normalizing flows with triangular Jacobians;
- exact direct-sampling models with a partition-function oracle;
- ancestral sampling in graphical models.

In these settings, the multidimensional sampling task reduces to a sequence of one-dimensional inverse-CDF evaluations.

This last remark brings us naturally to the next subsection.

7.1.2 Exact Sampling from Multivariate Distributions via Chain Rule

Exact sampling from a multivariate distribution

$$p(\mathbf{x}) = \frac{1}{Z} f(x_1, x_2, \dots, x_n)$$

is generally computationally difficult because computing the global partition function Z typically requires summing over an exponentially large state space. However, if an oracle is available to compute *partial partition functions*, then the high-dimensional sampling problem decomposes into a sequence of tractable one-dimensional conditional sampling problems.

The chain rule for probabilities gives

$$p(x_1, x_2, \dots, x_n) = p(x_1) p(x_2 | x_1) \cdots p(x_n | x_{1:n-1}).$$

Each conditional takes the form

$$p(x_i | x_{1:i-1}) = \frac{f(x_1, \dots, x_i)}{Z(x_1, \dots, x_{i-1})}, \quad Z(x_{1:i-1}) = \sum_{x_i \in \mathcal{X}} f(x_1, \dots, x_i),$$

where $Z(x_{1:i-1})$ is a partial partition function computed by the oracle. Sampling proceeds sequentially, starting with

$$p(x_1) = \frac{f(x_1)}{Z}, \quad Z = \sum_{x_1 \in \mathcal{X}} f(x_1).$$

Example 7.1.2 (Chain-Rule Sampling in a 2D Toy Model). *Consider the two-dimensional discrete distribution*

$$f(x_1, x_2) = \exp(-x_1^2 - 2x_1x_2 + 0.2x_2^2), \quad x_1, x_2 \in \{-3, -2, \dots, 3\}.$$

The oracle computes

$$Z(x_1) = \sum_{x_2} f(x_1, x_2), \quad Z = \sum_{x_1} Z(x_1).$$

The notebook `ChainRuleSampling-2D.ipynb` draws exact samples via the chain rule, plots the 2D histogram of (x_1, x_2) , and compares it to the normalized $f(x_1, x_2)$. Fig. 7.2 displays the theoretical density and the exact samples obtained by ancestral sampling.

Exercise 7.1.2 (Auto-regressive Sampling in Higher Dimensions). *Extend Example 7.1.2 to dimension $n = 5$.*

1. *Construct a function $f(x_{1:5})$ that factorizes weakly but not trivially (e.g., weak pairwise couplings).*
2. *Use a partial-partition oracle (implemented in Python) to compute $Z(x_{1:i-1})$ for $i = 1, \dots, 5$.*
3. *Produce 20,000 exact samples and visualize all pairwise marginals.*
4. *Compare the empirical marginals to the analytically normalized distribution.*

Comment on computational cost and how it scales with dimension.

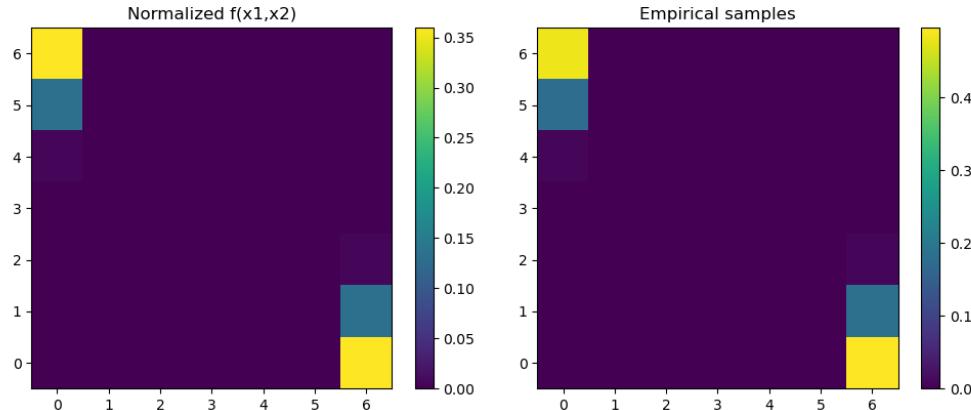


Figure 7.2: Exact ancestral sampling from the 2D toy distribution in Example 7.1.2. *Left:* heat map of the normalized $f(x_1, x_2)$. *Right:* empirical histogram of (x_1, x_2) samples obtained by the chain-rule method. Figures produced by `ChainRuleSampling-2D.ipynb`.

Chain-Rule Sampling and Auto-regressive Generative Models

This exact ancestral-sampling procedure mirrors the structure of auto-regressive models used in sequence modeling and Transformers. Given the past (x_1, \dots, x_{i-1}) , the model produces a conditional distribution over x_i . Exact chain-rule sampling is possible only when partial partition functions are tractable; in high-dimensional settings, this limitation motivates approximate versions such as masked auto-regressive flows and neural auto-regressive Transformers.

7.2 Importance Sampling and its Applications

Sampling efficiently from complex distributions is a core challenge in computational statistics, Bayesian inference, and generative modeling. **Importance Sampling (IS)** provides a principled, unbiased way to compute expectations with respect to a target distribution $p(x)$ by instead drawing samples from a simpler proposal distribution $q(x)$ and reweighting the samples.

7.2.1 General Formulation of Importance Sampling

Suppose we wish to compute

$$\mathbb{E}_p[f(x)] = \int f(x)p(x) dx,$$

but sampling from p is difficult. Let q be a tractable proposal distribution whose support covers that of p . Rewriting the expectation:

$$\mathbb{E}_p[f(x)] = \int f(x) \frac{p(x)}{q(x)} q(x) dx,$$

and drawing samples $x_i \sim q$, we obtain the *importance sampling estimator*

$$\widehat{\mathbb{E}}_p[f] = \frac{1}{N} \sum_{i=1}^N w(x_i) f(x_i), \quad w(x_i) = \frac{p(x_i)}{q(x_i)}.$$

This estimator is unbiased and consistent provided $q(x) > 0$ whenever $p(x) > 0$.

A well-chosen proposal q should place mass in regions where p is large, so that the weights $w(x)$ are stable and have low variance. Poor choice of q leads to heavy-tailed weight distributions and unstable estimates.

Example 7.2.1 (Estimating a Rare-Event Probability). *Consider estimating the rare event probability*

$$P(X > 3) = \int_3^\infty p(x) dx, \quad X \sim \mathcal{N}(0, 1).$$

Direct Monte Carlo is inefficient: with 10^5 samples we expect only about 135 hits in $x > 3$. We introduce a proposal distribution

$$q(x) = \mathcal{N}(3, 1),$$

which places much more mass in the rare-event region. The IS estimate becomes:

$$P(X > 3) \approx \frac{1}{N} \sum_{i=1}^N w(x_i), \quad w(x_i) = \frac{p(x_i)}{q(x_i)}, \quad x_i \sim q.$$

The notebook `ImportanceSampling-RareEvent.ipynb` simulates this experiment and produces Fig. 7.3 showing:

- the target density and the shifted proposal,
- Monte Carlo vs. IS estimate as N increases,
- a dramatic reduction in estimator variance under IS.

7.2.2 Importance Sampling for Posterior Estimation

In Bayesian inference, the target distribution is a posterior

$$p(\theta | y) \propto p(y | \theta)p(\theta),$$

which is often known only up to a normalizing constant. Importance sampling allows us to estimate posterior expectations using samples $\theta_i \sim q(\theta)$ from a convenient proposal. Weights are

$$w(\theta_i) = \frac{p(y | \theta_i)p(\theta_i)}{q(\theta_i)}.$$

Normalizing weights

$$\tilde{w}_i = \frac{w(\theta_i)}{\sum_{j=1}^N w(\theta_j)}$$

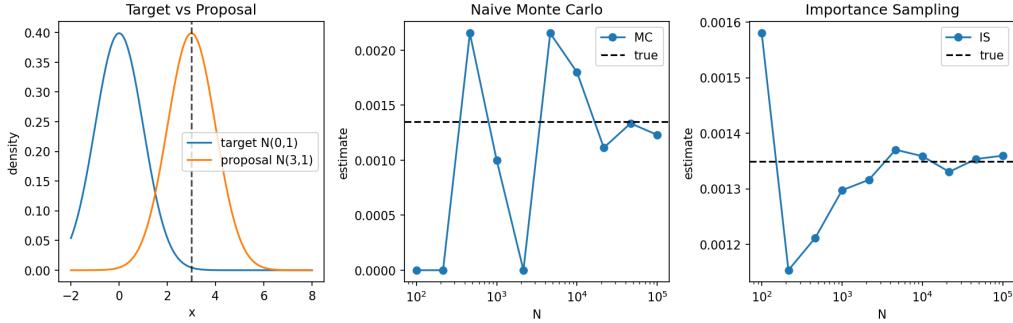


Figure 7.3: Importance sampling for the rare-event probability $P(X > 3)$. *Left:* target density p and proposal q . *Middle:* Monte Carlo estimator using $x_i \sim \mathcal{N}(0, 1)$. *Right:* IS estimator with $x_i \sim \mathcal{N}(3, 1)$, showing orders of magnitude lower variance. All generated by `ImportanceSampling-RareEvent.ipynb`.

yields the self-normalized IS estimator

$$\mathbb{E}_{p(\theta|y)}[f(\theta)] \approx \sum_{i=1}^N \tilde{w}_i f(\theta_i).$$

Example 7.2.2 (Posterior Estimation in a Gaussian Model). *Let the model be*

$$y \mid \theta \sim \mathcal{N}(\theta, 1), \quad \theta \sim \mathcal{N}(0, 10), \quad y = 3.$$

The posterior is Gaussian but we pretend it is unknown in order to test IS. We use a Gaussian proposal

$$q(\theta) = \mathcal{N}(\mu_q, \sigma_q^2), \quad (\mu_q, \sigma_q^2) = (2, 2),$$

and compute weights

$$w(\theta) = \frac{\exp\left(-\frac{1}{2}(y - \theta)^2\right) \exp\left(-\frac{\theta^2}{20}\right)}{\exp\left(-\frac{(\theta - \mu_q)^2}{2\sigma_q^2}\right)}.$$

The notebook `ImportanceSampling-GaussianPosterior.ipynb` evaluates:

- weighted posterior mean and variance estimates,
- the weight distribution and its heavy-tailed behavior,
- the effective sample size (ESS):

$$\text{ESS} = \frac{(\sum_i w_i)^2}{\sum_i w_i^2},$$

a key diagnostic of proposal quality.

Fig. 7.4 compares the true posterior to the IS approximation.

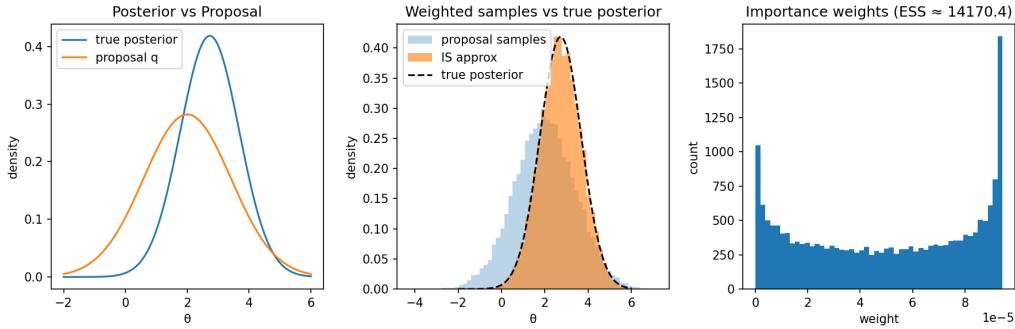


Figure 7.4: Importance sampling for the Gaussian posterior in Example 7.2.2. *Left:* true posterior density and proposal q . *Middle:* weighted sample histogram approximating the posterior. *Right:* importance weights and resulting ESS. Figures produced by `ImportanceSampling-GaussianPosterior.ipynb`.

Importance Sampling in Generative AI

Importance sampling illustrates fundamental limitations of sampling in high-dimensional spaces: proposal mismatch causes weight degeneracy and vanishing ESS. This curse of dimensionality motivates Markov-chain methods (Section 7.5), Langevin dynamics, and ultimately the score-based diffusion samplers used in modern generative models. Many training objectives in diffusion models and VAEs can be interpreted as reducing the mismatch between the learned model and the target distribution, thereby improving IS-like efficiency.

Exercise 7.2.1 (Diagnostics and Proposal Design in Importance Sampling). *Use the notebook `ImportanceSampling-GaussianPosterior.ipynb` as a base.*

- Experiment with several proposals $q(\theta)$ by varying (μ_q, σ_q^2) . For each choice, compute the weight histogram, ESS, and the IS estimate of the posterior mean.*
- Plot ESS as a function of the proposal variance and identify the optimal region. Compare with intuition from KL divergence between q and the posterior.*
- Modify the example so that the likelihood is heavy-tailed (e.g. using a Student- t model). Study how importance weights behave and how ESS changes.*
- Discuss the failure mode of IS in moderate dimension ($d = 10$), where even mildly mismatched proposals produce $ESS \ll 1$.*

Summarize the regimes where IS is effective and where alternative sampling methods (MCMC, SDE-based methods, diffusion models) are necessary.

7.2.3 Adaptive Importance Sampling and the Cross-Entropy Method

Importance Sampling (IS) is effective only when the proposal distribution $q(x)$ places mass in the same regions as the target $p(x)$. When the mismatch is large, weights become

heavy-tailed and the effective sample size (ESS) collapses. *Adaptive Importance Sampling (AIS)* aims to fix this by iteratively *adapting* the proposal distribution to better approximate the target.

A powerful and widely used AIS framework is the **Cross-Entropy (CE) Method**, introduced by Rubinstein [41, 42]. Originally developed for rare-event simulation and combinatorial optimization, CE has become a pillar of adaptive Monte Carlo methods, reinforcement learning, sequence design, and modern generative modeling (e.g., adaptive proposal training, energy-based model sampling).

Goal of Adaptive IS

Suppose we want to evaluate

$$\mathcal{I} = \mathbb{E}_p[f(x)]$$

but sampling directly from p is difficult. We choose a parametric proposal family $q_\theta(x)$ and seek parameters θ^* such that q_θ approximates the optimal IS proposal

$$q^*(x) \propto |f(x)|p(x),$$

which in general is intractable.

AIS updates θ iteratively from

$$\theta_{t+1} \leftarrow \arg \max_{\theta} \mathbb{E}_{q_{\theta_t}} \left[w_{\theta_t}(x) \log q_{\theta}(x) \right], \quad w_{\theta_t}(x) = \frac{p(x)}{q_{\theta_t}(x)}.$$

This is equivalent to minimizing the KL divergence

$$\theta_{t+1} = \arg \min_{\theta} D_{\text{KL}}(q^* \parallel q_{\theta}),$$

and can be viewed as a weighted maximum-likelihood update.

Let us adopt general CE method – discussed in Section 6.2.4 – to Importance Sampling. In the CE method, the update is expressed through the *cross-entropy*

$$\text{CE}(q^*, q_{\theta}) = -\mathbb{E}_{q^*}[\log q_{\theta}(x)].$$

Since q^* is unknown, we approximate expectations with weighted samples $x_i \sim q_{\theta_t}$.

A defining feature of the CE method is that the weights are constructed from *elite samples*. For rare-event probability estimation, the elite set corresponds to the top ρ -quantile of the likelihood ratio, or equivalently, the set of samples that fall in the rare region.

Let E_t be the elite set at iteration t :

$$E_t = \{x_i : S(x_i) \geq \gamma_t\},$$

where $S(x)$ is a score function (e.g. indicator of a rare event) and γ_t is the empirical ρ -quantile threshold. The CE update becomes a maximum-likelihood fit:

$$\theta_{t+1} = \arg \max_{\theta} \sum_{x_i \in E_t} \log q_{\theta}(x_i).$$

Thus CE re-fits q_{θ} to the elite samples at every iteration, gradually shifting mass toward the regions where $p(x)$ is large.

Example 7.2.3 (Adaptive IS via CE for Fitting a Target Distribution). *To illustrate the mechanics of Adaptive Importance Sampling (AIS) and the Cross-Entropy (CE) update, we consider a clean, well-controlled setting where the goal is to adapt a Gaussian proposal distribution toward a known target. This avoids the instability of rare-event optimization and shows clearly how adaptive IS reduces weight variance and improves ESS.*

Setup. Let the target be

$$p(x) = \mathcal{N}(2, 0.75^2),$$

and let the proposal family be

$$q_{\theta}(x) = \mathcal{N}(\mu, \sigma^2).$$

We begin with a poor initial guess, for example

$$(\mu_0, \sigma_0) = (0, 3).$$

AIS / CE Update. At iteration t :

1. Draw samples $x_i \sim q_{\theta_t}$.
2. Compute importance weights

$$w_i = \frac{p(x_i)}{q_{\theta_t}(x_i)}, \quad \tilde{w}_i = \frac{w_i}{\sum_j w_j}.$$

3. Perform a weighted maximum-likelihood update:

$$\mu^{(w)} = \sum_i \tilde{w}_i x_i, \quad (\sigma^2)^{(w)} = \sum_i \tilde{w}_i (x_i - \mu^{(w)})^2.$$

4. Apply smoothing:

$$\mu_{t+1} = \alpha \mu^{(w)} + (1 - \alpha) \mu_t, \quad \sigma_{t+1}^2 = \alpha (\sigma^2)^{(w)} + (1 - \alpha) \sigma_t^2,$$

with $\alpha \in (0, 1)$ and a small variance floor to avoid degeneracy.

This update decreases the KL divergence $D_{\text{KL}}(p\|q_{\theta})$ and improves the quality of the proposal. The accompanying notebook `AdaptiveIS-CE-fitGaussian.ipynb` implements this procedure. Figure 7.5 shows:

- **Proposal evolution:** (μ_t, σ_t) moves smoothly from a broad, misaligned initial proposal toward the true target.
- **Weight stabilization:** the variance of the normalized weights decreases over iterations.
- **ESS improvement:** the effective sample size increases significantly as the proposal approaches the target.

This example demonstrates the core principle of AIS: iteratively adapting the proposal sharply reduces weight degeneracy, increasing sampling efficiency.

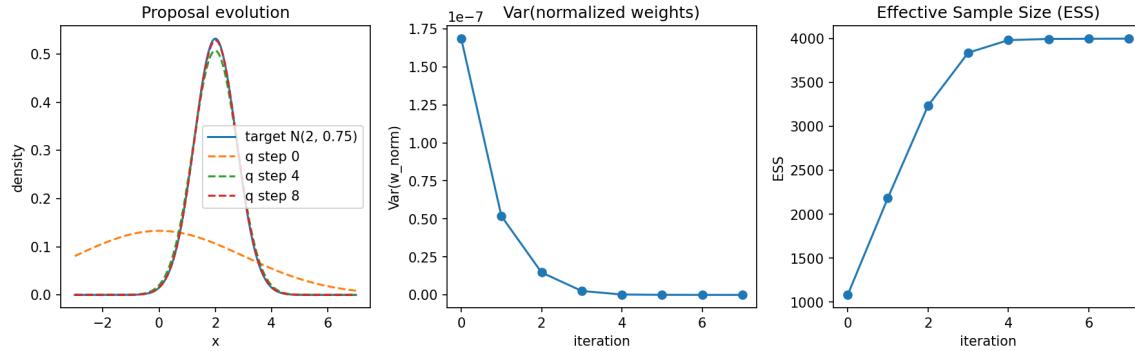


Figure 7.5: Adaptive Importance Sampling using the Cross-Entropy update applied to fitting a Gaussian target $p(x) = \mathcal{N}(2, 0.75^2)$. *Left:* evolution of the proposal densities over iterations. *Middle:* variance of normalized importance weights decreases. *Right:* ESS increases substantially as the proposal improves. Figures generated by `AdaptiveIS-CE-fitGaussian.ipynb`.

Exercise 7.2.2 (Adaptive IS for a Non-Gaussian Target). *Implement AIS and CE for sampling a heavy-tailed target distribution*

$$p(x) \propto \frac{1}{1+x^2}, \quad x \in \mathbb{R},$$

using a Gaussian proposal family $q_\theta = \mathcal{N}(\mu_t, \sigma_t^2)$.

- (a) Derive the weighted MLE update for (μ_t, σ_t^2) .
- (b) Implement the CE version where elite samples replace weights. Compare convergence behavior.
- (c) Track and plot ESS over iterations. Describe how AIS improves proposal quality.
- (d) Discuss why heavy-tailed targets can destabilize AIS and how CE (elite re-fitting) partially mitigates such instability.

Adaptive Importance Sampling in Modern AI

AIS and the CE method appear implicitly throughout modern generative modeling:

- In diffusion models, optimal proposal schedules minimize a KL objective structurally identical to CE.
- In reinforcement learning, policy-gradient and trajectory-optimization schemes (e.g. CEM-RL) directly descend the CE objective.
- In energy-based and score-based models, AIS is used to calibrate and refine approximate samplers.

Viewed broadly, AIS provides the conceptual bridge between classical IS and the adaptive, KL-driven training procedures that dominate contemporary generative AI.

7.3 Diffusion and Brownian Motion

Diffusion processes form the mathematical backbone of modern generative models based on noise injection and gradual denoising, including *score-based models* and *Denoising Diffusion Probabilistic Models (DDPMs)*. In these models, data are progressively transformed into noise by a forward diffusion process and then reconstructed by learning how to reverse that diffusion.

This section develops the theory of diffusion from first principles. We proceed in the following steps:

1. Introduce Brownian motion as the canonical continuous-time stochastic process with independent Gaussian increments.
2. Derive the diffusion (heat) equation from Brownian motion using both path integrals and Chapman–Kolmogorov consistency.
3. Generalize diffusion by introducing drift through a potential, leading to Langevin dynamics and the Fokker–Planck equation.
4. Interpret diffusion as convolution with the heat kernel and connect this viewpoint to discrete approximations used in generative AI.

These ideas will culminate in a conceptual bridge to diffusion-based generative models, where learning the reverse-time dynamics enables sampling from complex data distributions.

7.3.1 Diffusion from Brownian Motion

Brownian motion is the simplest nontrivial continuous-time stochastic process and serves as the universal scaling limit of random walks. Its defining properties — independent increments, Gaussian fluctuations, and Markovian evolution — make it the natural “base process” for forward diffusion in generative models. In score-based and diffusion probabilistic models, the forward process is precisely a Brownian motion (possibly with time-dependent variance), while the learning problem focuses on approximating the reverse-time dynamics.

Consider the simplest case of Brownian motion, where a particle diffuses with no advection. The stochastic differential equation (SDE) for Brownian motion is given by ¹

$$dX_t = \sqrt{2D} dW_t, \tag{7.1}$$

where

¹The Wiener process, named after Norbert Wiener, provides the first rigorous mathematical formulation of Brownian motion. Although Brownian motion had been observed since the 19th century and modeled statistically by Einstein and Smoluchowski, it was Wiener who, in the early 1920s, defined the continuous-time stochastic process that now bears his name. His construction used functional analysis to describe the probability space of continuous paths, formalizing properties such as continuity, Gaussian increments, and the Markov property. Wiener’s 1923 paper [43] introduced “differential space,” and in 1924 in [44] he clarified the link between Brownian motion and harmonic analysis — laying the groundwork for modern stochastic calculus and the path-space approach to diffusion.

- X_t is the position of the particle at time t ;
- D is the diffusion coefficient;
- W_t is the standard Wiener process (or Brownian motion).

The Wiener process W_t is defined by the properties:

1. $W_0 = 0$;
2. W_t has independent increments;
3. The increment $W_{t+\epsilon} - W_t$ is normally distributed with mean 0 and incremental variance ϵ :

$$W_{t+\epsilon} - W_t \sim \mathcal{N}(0, \epsilon) \quad (7.2)$$

As we see below this translates into accumulation of variance over time:

$$\text{Var}(\sqrt{2D} W_t) = 2D t.$$

7.3.2 From the Stochastic Differential Equation to the Path Integral

To connect the stochastic differential equation (SDE) description of Brownian motion with the path integral representation, we begin by discretizing the SDE and examining the resulting probability distribution over paths.

Recall that the SDE for Brownian motion is given by Eq. (7.1). We discretize the time interval $[0, t]$ into N steps of size $\epsilon = t/N$ each, and define $X_n = X(n\epsilon)$, $n = 1, \dots, N$. The increment of the Wiener process satisfies Eq. (7.2) which implies that:

$$X_{n+1} - X_n \sim \mathcal{N}(0, 2D\epsilon).$$

Therefore, the transition probability density from X_n to X_{n+1} is given by the Gaussian:

$$p(X_{n+1}|X_n) = \frac{1}{\sqrt{4\pi D\epsilon}} \exp\left(-\frac{(X_{n+1} - X_n)^2}{4D\epsilon}\right).$$

The joint probability density for the full trajectory $\{X_1, \dots, X_{N-1}\}$ is:

$$p(X_1, \dots, X_{N-1}) \propto \left(\prod_{n=1}^{N-1} \frac{1}{\sqrt{4\pi D\epsilon}} \right) \exp\left(-\sum_{n=0}^{N-1} \frac{(X_{n+1} - X_n)^2}{4D\epsilon}\right),$$

and thus the marginal probability density of observing $X_N = x$ given $X_0 = x_0$ – thus marginalized over x_1, \dots, x_{N-1} is

$$p(x_N|x_0) \approx \int \left(\prod_{n=1}^{N-1} \frac{dX_n}{\sqrt{4\pi D\epsilon}} \right) \exp\left(-\sum_{n=0}^{N-1} \frac{(X_{n+1} - X_n)^2}{4D\epsilon}\right).$$

Recognizing the exponent as a Riemann sum approximation of an integral, we transition to the continuum limit:

$$\sum_{n=0}^{N-1} \frac{(X_{n+1} - X_n)^2}{\epsilon} \rightarrow \int_0^t \left(\frac{dX(\tau)}{d\tau} \right)^2 d\tau.$$

Thus, we arrive at the so-called *path integral*, also called Feynman-Kac, formulation²:

$$p(x_t | x_0) = \int_{X(0)=x_0}^{X(t)=x_t} \mathcal{D}[X] \exp \left[-\frac{1}{4D} \int_0^t \left(\frac{dX(\tau)}{d\tau} \right)^2 d\tau \right],$$

where $\mathcal{D}[X]$ denotes the path integral measure defined as the continuum limit of the finite-dimensional product of Gaussian integrals.

From Path Integral to Diffusion. Using the notation introduced earlier in this subsection, we now derive the diffusion equation starting from the path integral formulation. The marginal probability density can be written as the continuum limit of a discrete sum over intermediate positions:

$$p_t(x) = \lim_{N \rightarrow \infty} \int dX_1 \cdots dX_{N-1} \prod_{n=0}^{N-1} K(X_{n+1}, X_n; \epsilon),$$

where the short-time propagator K is given by:

$$K(X_{n+1}, X_n; \epsilon) \approx \frac{1}{\sqrt{4\pi D\epsilon}} \exp \left[-\frac{(X_{n+1} - X_n)^2}{4D\epsilon} \right].$$

This Gaussian kernel encodes the transition probability for a diffusing particle over a single time step of duration ϵ .

Marginalization over Intermediate Times. The product

$$\prod_{n=0}^{N-1} K(X_{n+1}, X_n; \epsilon)$$

²The Feynman-Kac formula is a foundational result linking stochastic processes and partial differential equations (PDEs), uniting ideas from physics and probability. Physicist Richard Feynman introduced the path integral formulation in quantum mechanics during the 1940s as a way to compute quantum amplitudes by summing over all possible trajectories, each path weighted by a complex exponential of the classical action. Around the same time, mathematician Mark Kac developed a probabilistic method for solving parabolic PDEs by computing expectations over Brownian motion trajectories. Kac's work, particularly his 1949 paper, provided a rigorous mathematical foundation for interpreting solutions to the Schrödinger and heat equations as averages over stochastic paths — a viewpoint deeply rooted in the earlier work of Norbert Wiener. In fact, Kac's construction built directly on Wiener's rigorous definition of Brownian motion and his development of path-space integration in the early 1920s. The resulting Feynman-Kac formula can be seen as a synthesis: while Feynman's path integrals were originally heuristic and formal in the physics tradition, Kac's use of Wiener integrals provided the analytical machinery to render them precise in imaginary (Euclidean) time. Today, this formulation is a cornerstone of mathematical physics, quantitative finance, and generative modeling via stochastic differential equations. See Wiener (1923, 1924) [43, 44], Kac (1949)[45], and Feynman & Hibbs (1965)[46].

can be interpreted as a chain of conditional transition probabilities. The marginal probability $P(X_N = x, t \mid X_0 = x_0)$ is computed by integrating over all intermediate positions X_1, \dots, X_{N-1} , which embodies the structure of the Chapman–Kolmogorov equation³:

$$p_t(x) = \int dX' p_{t-\epsilon \rightarrow t}(x \mid X') p_{t-\epsilon}(X').$$

Derivation of the Diffusion Equation. We analyze the small-time limit of the Chapman–Kolmogorov relation:

$$p_{t+\epsilon}(x) = \int_{-\infty}^{\infty} dX' K(x, X'; \epsilon) p_t(X').$$

Expanding $p_t(X')$ about x via Taylor series:

$$p_t(X') = p_t(x) + (X' - x) \frac{\partial p_t(x)}{\partial x} + \frac{(X' - x)^2}{2} \frac{\partial^2 p_t(x)}{\partial x^2} + \dots,$$

and applying standard Gaussian integrals:

$$\begin{aligned} \int_{-\infty}^{\infty} \frac{dX'}{\sqrt{4\pi D\epsilon}} \exp\left[-\frac{(x - X')^2}{4D\epsilon}\right] &= 1, \\ \int_{-\infty}^{\infty} \frac{(X' - x) dX'}{\sqrt{4\pi D\epsilon}} \exp\left[-\frac{(x - X')^2}{4D\epsilon}\right] &= 0, \\ \int_{-\infty}^{\infty} \frac{(X' - x)^2 dX'}{\sqrt{4\pi D\epsilon}} \exp\left[-\frac{(x - X')^2}{4D\epsilon}\right] &= 2D\epsilon, \end{aligned}$$

we find:

$$p_{t+\epsilon}(x) \approx p_t(x) + D\epsilon \frac{\partial^2 p_t(x)}{\partial x^2}.$$

Taking the limit $\epsilon \rightarrow 0$ yields the **diffusion equation**:

$$\frac{\partial p_t(x)}{\partial t} = D \frac{\partial^2 p_t(x)}{\partial x^2},$$

which is a deterministic Partial Differential Equation (PDE) describing the time evolution of the probability density $p_t(x)$ in one spatial dimension.

Example 7.3.1 (Brownian Motion and the Heat Equation in One Dimension). *Consider the Brownian motion*

$$dX_t = \sqrt{2D} dW_t, \quad X_0 = 0,$$

whose marginal density satisfies the heat equation

$$\partial_t p_t(x) = D \partial_x^2 p_t(x), \quad p_0(x) = \delta(x).$$

³This integral identity is attributed to Sydney Chapman and Andrey Kolmogorov, who independently formalized the evolution of probability distributions under Markovian dynamics. Chapman's work (circa 1928) laid the foundation in statistical mechanics, while Kolmogorov's 1931 axiomatization of probability theory gave the result its modern mathematical rigor. The resulting identity — a recursive composition of transition probabilities — now serves as a central organizing principle in stochastic processes, underpinning both forward and backward equations in diffusion and Markov chains.

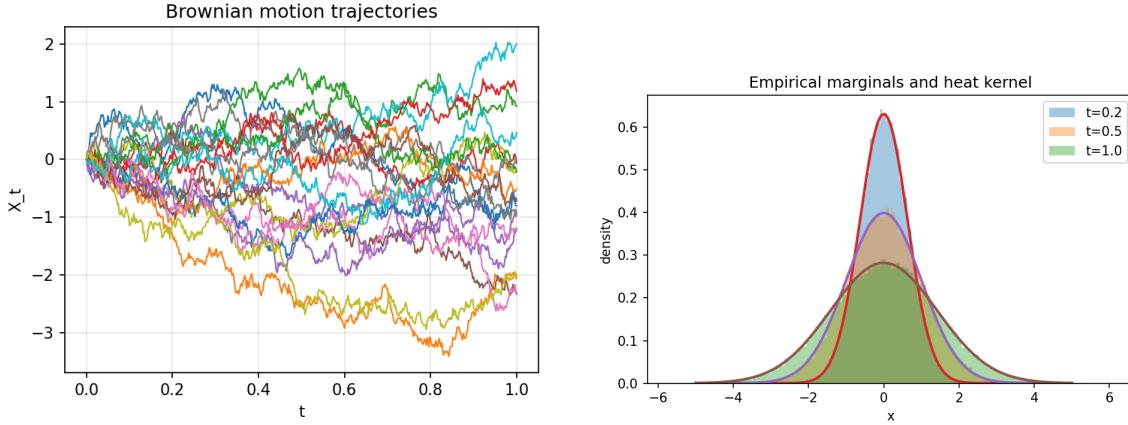


Figure 7.6: Brownian motion and diffusion. *Left:* sample trajectories of 1D Brownian motion generated via Euler–Maruyama discretization. *Right:* empirical marginal distributions at increasing times, compared with the analytic heat kernel solution of the diffusion equation. Figures generated by `BrownianMotion-and-HeatEquation.ipynb`.

The analytic solution is the heat kernel:

$$p_t(x) = \frac{1}{\sqrt{4\pi Dt}} \exp\left(-\frac{x^2}{4Dt}\right),$$

which is simply a Gaussian whose variance grows linearly with time.

A standard numerical method for simulating Stochastic Differential Equations (SDEs) is the Euler–Maruyama scheme⁴. We implement it by discretizing time as $t_k = k\Delta t$ and approximating the SDE by

$$X_{k+1} = X_k + \sqrt{2D} \Delta W_k, \quad \Delta W_k \sim \mathcal{N}(0, \Delta t),$$

or equivalently

$$X_{k+1} = X_k + \sqrt{2D \Delta t} \xi_k, \quad \xi_k \sim \mathcal{N}(0, 1).$$

The accompanying notebook `BrownianMotion-and-HeatEquation.ipynb` illustrates three complementary viewpoints:

- **Path space:** simulated Brownian trajectories using Euler–Maruyama;
- **Marginals:** empirical histograms at multiple times compared to the analytic heat kernel;
- **PDE evolution:** numerical solution of the heat equation via convolution with the Gaussian kernel (FFT-based).

⁴The Euler–Maruyama method is the stochastic analogue of the classical Euler method for ordinary differential equations. It was introduced by Gisiro Maruyama in 1955 as a rigorous discretization scheme for stochastic differential equations driven by Wiener processes. Maruyama showed that replacing infinitesimal Wiener increments dW_t by Gaussian random variables $\sqrt{\Delta t} \xi_k$, with $\xi_k \sim \mathcal{N}(0, 1)$, yields a convergent approximation to the true stochastic process. The method is named in analogy with the deterministic Euler scheme and is now a cornerstone of numerical stochastic analysis and simulation-based modeling.

These three perspectives — trajectories, probability densities, and PDEs — are mathematically equivalent and form the foundation of diffusion-based generative modeling.

Exercise 7.3.1 (Brownian Motion with Absorbing and Reflecting Boundaries). Consider standard Brownian motion

$$dX_t = \sqrt{2D} dW_t, \quad X_0 = x_0 \in (0, L),$$

but now confined to an interval $[0, L]$.

Discrete simulation. Use the Euler–Maruyama update

$$X_{k+1} = X_k + \sqrt{2D \Delta t} \xi_k, \quad \xi_k \sim \mathcal{N}(0, 1),$$

and impose the boundary condition after each step, according to the cases below. Extend the notebook `BrownianMotion-and-HeatEquation.ipynb` accordingly.

- (a) **Absorbing boundary at 0 and L.** Declare a trajectory killed (absorbed) when it first exits the interval. Simulate many trajectories and estimate:

- the survival probability

$$S(t) = \mathbb{P}(\tau > t), \quad \tau = \inf\{t \geq 0 : X_t \notin (0, L)\},$$

- the empirical distribution of the first passage time τ ,
- the probability of exiting through the right endpoint $\mathbb{P}(X_\tau = L)$.

- (b) **Reflecting boundary at 0 and L.** Modify the simulation so that paths are reflected at the boundaries: whenever a step proposes $X_{k+1} < 0$ set $X_{k+1} \leftarrow -X_{k+1}$, and whenever $X_{k+1} > L$ set $X_{k+1} \leftarrow 2L - X_{k+1}$. Simulate trajectories and estimate the marginal density $p_t(x)$ at multiple times. What distribution do you observe as $t \rightarrow \infty$?

- (c) **Compare the two cases.** For absorbing boundaries, mass disappears over time (trajectories are killed), while for reflecting boundaries, total probability is conserved. Explain how this is visible in your simulation outputs (histograms and $S(t)$).

- (d) **Bonus: PDE and boundary conditions.** Let $p_t(x)$ denote the density of X_t . Argue that in both cases the interior evolution is still governed by the diffusion equation

$$\partial_t p_t(x) = D \partial_x^2 p_t(x), \quad x \in (0, L),$$

but the boundary conditions differ:

- absorbing boundary: $p_t(0) = p_t(L) = 0$ (Dirichlet),
- reflecting boundary: $\partial_x p_t(0) = \partial_x p_t(L) = 0$ (Neumann).

Give a short intuitive explanation in terms of probability flux.

Implementation note. For (a), keep track of the first time a path leaves $(0, L)$; for (b), implement reflection step-by-step. Use the same Δt , number of trajectories, and time horizons in both cases so that comparisons are meaningful.

Generalization to Higher Dimensions. The above derivation assumes $x \in \mathbb{R}$, i.e., motion in a one-dimensional spatial domain. However, the formalism generalizes naturally to higher dimensions, where $x \in \mathbb{R}^d$, for arbitrary spatial dimension $d = 1, 2, \dots$. In this setting, the first derivative ∂_x becomes the gradient ∇ , and the second derivative ∂_x^2 is replaced by the Laplacian operator:

$$\Delta = \sum_{i=1}^d \frac{\partial^2}{\partial x_i^2}.$$

Accordingly, the diffusion equation in higher dimensions becomes:

$$\frac{\partial p_t(x)}{\partial t} = D \Delta p_t(x),$$

where now $\nabla p_t(x)$ is a vector field, and $\Delta p_t(x)$ captures the spatial spread of probability mass across dimensions.

In summary – by discretizing time, we expressed the path integral as a product of transition kernels and marginalized over intermediate positions. Through the Chapman–Kolmogorov identity and a Taylor expansion of the probability density, we derived the diffusion equation in the continuous limit. This derivation illustrates the deep connection between the stochastic microscopic dynamics (in the form of random trajectories) and the deterministic macroscopic PDE that governs the evolution of marginal distributions.

7.3.3 Generalization: Diffusion with Drift Induced by a Potential

We now generalize the diffusion process by introducing a drift term governed by the gradient of a potential function $U(x)$. This leads to the following Stochastic Differential Equation (SDE):

$$dX_t = -\nabla U(X_t) dt + \sqrt{2D} dW_t, \quad (7.3)$$

where $\nabla U(X_t)$ is shorthand for $\nabla_X U(X)$ evaluated at $X = X_t$. This equation is commonly referred to as the *Langevin equation*⁵.

In this setting, the time evolution of the probability density $p_t(x)$ is governed by the *Fokker–Planck equation*⁶:

$$\frac{\partial p_t(x)}{\partial t} = \nabla_x \cdot (\nabla U(x) p_t(x)) + D \nabla^2 p_t(x).$$

In the *stationary state*, defined by the condition $\partial_t p_t^{(\text{st})}(x) = 0$, and assuming the system satisfies the so-called *detailed balance* (DB) condition – i.e., the probability flux vanishes –

⁵Named after Paul Langevin, who in 1908 proposed this equation to describe the stochastic dynamics of particles immersed in a fluid, combining deterministic drag with random thermal fluctuations. Langevin's formulation extended earlier work on Brownian motion by introducing a force term, thereby opening the path to modern theories of nonequilibrium statistical mechanics.

⁶The equation is named after Adriaan Fokker and Max Planck. Fokker derived a form of the equation in his study of Brownian motion under external forces, while Planck had earlier obtained a similar equation describing radiation and entropy in statistical physics. The modern name reflects their combined contributions to the theory of stochastic processes and statistical mechanics.

we have:

$$-\nabla_x U(x) p_t^{(\text{st})}(x) - D \nabla_x p_t^{(\text{st})}(x) = 0.$$

(This notion of detailed balance will be generalized in the next section to the case of discrete-time Markov chains, where time and space are discrete.)

Rewriting the above equation yields:

$$\nabla_x p_t^{(\text{st})}(x) = -\frac{1}{D} \nabla_x U(x) p_t^{(\text{st})}(x).$$

Integrating both sides leads to the stationary distribution:

$$p_t^{(\text{st})}(x) = \frac{1}{Z} \exp\left(-\frac{U(x)}{D}\right),$$

which is known as the *Boltzmann, Gibbs, or Boltzmann–Gibbs distribution*, where Z is the *partition function* ensuring normalization:

$$Z = \int_{\mathbb{R}^d} \exp\left(-\frac{U(x)}{D}\right) dx.$$

Ornstein–Uhlenbeck Process: A Solvable Langevin Model

A particularly important special case of the Langevin equation arises when the potential is quadratic,

$$U(x) = \frac{\lambda}{2}x^2,$$

which yields a linear drift. The resulting SDE,

$$dX_t = -\lambda X_t dt + \sqrt{2D} dW_t, \quad (7.4)$$

is known as the *Ornstein–Uhlenbeck (OU) process*⁷.

The corresponding Fokker–Planck equation is

$$\frac{\partial p_t(x)}{\partial t} = \frac{\partial}{\partial x} (\lambda x p_t(x)) + D \frac{\partial^2 p_t(x)}{\partial x^2}.$$

Exact Solution and Stationarity. If $X_0 = x_0$, the OU process admits an explicit solution:

$$X_t = x_0 e^{-\lambda t} + \sqrt{2D} \int_0^t e^{-\lambda(t-s)} dW_s,$$

from which one finds

$$\mathbb{E}[X_t] = x_0 e^{-\lambda t}, \quad \text{Var}(X_t) = \frac{D}{\lambda} (1 - e^{-2\lambda t}).$$

⁷The Ornstein–Uhlenbeck process was introduced in 1930 by Leonard Ornstein and George Eugene Uhlenbeck as a refinement of Brownian motion that incorporates mean-reverting dynamics [47]. Their motivation was to model the velocity of a Brownian particle subject to linear friction, extending earlier work by Einstein and Langevin. Because of its linear drift and Gaussian noise, the OU process is exactly solvable and has since become a standard model in physics, finance, biology, and machine learning.

As $t \rightarrow \infty$, the distribution converges to the stationary Gaussian

$$p^{(\text{st})}(x) = \sqrt{\frac{\lambda}{2\pi D}} \exp\left(-\frac{\lambda x^2}{2D}\right),$$

which is precisely the Boltzmann distribution associated with the quadratic potential $U(x)$.

Conceptual Role. The OU process plays a distinguished role as the simplest nontrivial diffusion with drift:

- it is Gaussian at all times,
- it exhibits exponential relaxation toward equilibrium,
- it provides a local linear approximation to general Langevin dynamics near stable equilibria.

For these reasons, OU-type noise processes are widely used to model regularizing noise in neural networks and as analytically tractable building blocks in diffusion-based generative models.

Double-Well Potential

While the Ornstein–Uhlenbeck (OU) process provides an exactly solvable example with a single stable equilibrium, many systems of interest in physics, chemistry, and machine learning exhibit *multiple metastable states*. A double-well potential offers the simplest setting in which diffusion, drift, and noise interact to produce barrier crossing, long correlation times, and nontrivial sampling behavior.

Example 7.3.2 (Langevin Diffusion in a Double-Well Potential). *To illustrate diffusion with drift, consider the one-dimensional Langevin equation*

$$dX_t = -U'(X_t) dt + \sqrt{2D} dW_t,$$

with the double-well potential

$$U(x) = \frac{1}{4}(x^2 - 1)^2.$$

This potential has two stable minima at $x = \pm 1$ separated by an energy barrier at $x = 0$. The stationary distribution predicted by the Fokker–Planck equation under detailed balance is the Boltzmann distribution

$$p^{(\text{st})}(x) \propto \exp\left(-\frac{U(x)}{D}\right),$$

which concentrates near the minima for small D and spreads out as D increases.

The notebook `Langevin-DoubleWell.ipynb` simulates this process using the Euler–Maruyama scheme and visualizes both sample trajectories and empirical marginals. It highlights how stochastic noise enables barrier crossing between the two wells and how long-time averages recover the Gibbs distribution.

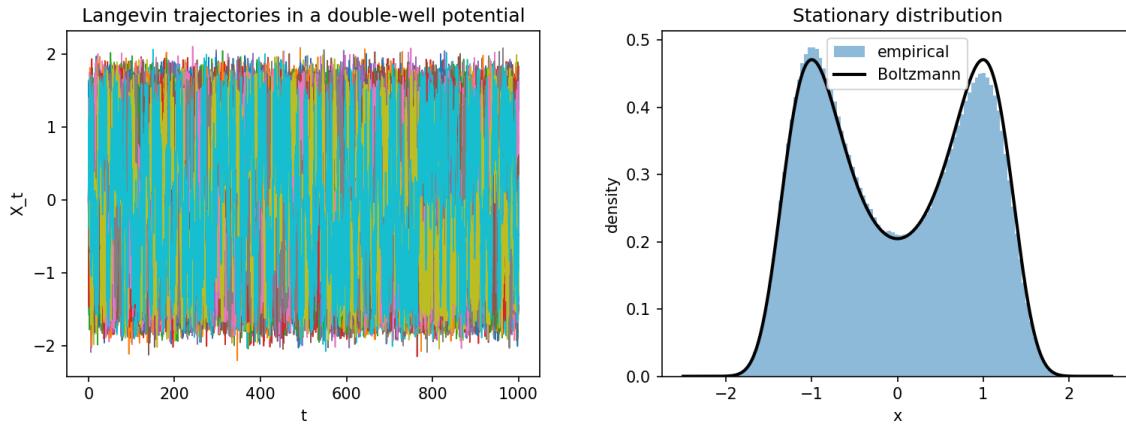


Figure 7.7: Langevin diffusion in a double-well potential. *Left:* sample trajectories showing metastability and noise-induced transitions between wells. *Right:* empirical stationary distribution compared with the analytic Boltzmann–Gibbs density. Figures generated by `Langevin-DoubleWell.ipynb`.

Exercise 7.3.2 (Langevin Sampling, Metastability, and Stationarity). *Consider the Langevin SDE*

$$dX_t = -U'(X_t) dt + \sqrt{2D} dW_t, \quad U(x) = \frac{1}{4}(x^2 - 1)^2.$$

- (a) Derive the associated Fokker–Planck equation and verify that the Boltzmann distribution
$$p^{(\text{st})}(x) \propto e^{-U(x)/D}$$
is stationary under detailed balance.
- (b) Implement the Euler–Maruyama scheme and simulate trajectories for different values of the diffusion coefficient \$D\$. How does \$D\$ affect the frequency of transitions between the two wells?
- (c) Estimate the empirical stationary distribution from long-time simulation and compare it to the analytic Boltzmann distribution, as in Fig. 7.7.
- (d) Explain why small \$D\$ leads to metastability and long correlation times, and relate this to challenges in sampling multimodal distributions.
- (e) (Bonus) Discuss how adding a non-gradient drift term would break detailed balance and change the stationary behavior.

In summary – the derivation + example + exercise presented above demonstrate how the diffusion equation emerges naturally from Brownian motion via the path integral formulation. Extending the model to include a drift term given by the gradient of a potential \$U(x)\$ leads to the Fokker–Planck equation. The stationary solution under detailed balance conditions yields the Gibbs distribution—a fundamental object in statistical mechanics and machine learning. These ideas will be pivotal in our discussion of diffusion-based generative models, where the interplay of noise, structure, and reversibility enables powerful sampling mechanisms.

From Brownian Motion to Score-Based Diffusion Models

Brownian motion (often with drift, e.g. of the Ornstein-Uhlenbeck process) provides the canonical *forward diffusion* used in score-based generative models and DDPMs. As time increases, the data distribution is convolved with Gaussian noise, eventually approaching a simple reference distribution.

Score-based models learn the *score function*

$$\nabla_x \log p_t(x),$$

along this diffusion trajectory. By reversing the diffusion process using this learned score, one can transform noise back into structured data.

Thus, Brownian motion, the heat equation, and Langevin dynamics form the mathematical foundation underlying modern diffusion-based generative AI.

7.4 Markov Chains

A *Markov chain* is a discrete-time stochastic process $\{X_n\}_{n \geq 0}$ with a finite (or countable) state space \mathcal{S} that satisfies the Markov property:

$$P(X_{n+1} = j \mid X_n = i, X_{n-1} = i_{n-1}, \dots, X_0 = i_0) = P(X_{n+1} = j \mid X_n = i)$$

for all n and all $i_0, i_1, \dots, i_{n-1}, i, j \in \mathcal{S}$.

Consider examples:

- **Two-State Chain:** Consider a simple system with two nodes A and B . A possible transition matrix is

$$P = \begin{pmatrix} 0.8 & 0.2 \\ 0.3 & 0.7 \end{pmatrix}.$$

This chain is irreducible (each state is reachable from every other state) and, if it is also aperiodic (no cyclic behavior), it converges to a unique stationary distribution.

- **Three-State Cycle:** Suppose we have three states A , B , and C with the transition matrix

$$P = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}.$$

Although every state is reachable from every other state (irreducibility), this chain is periodic (with period 3) and does not mix unless additional randomness (or self-loops) is introduced.

- **Chain (Linear Sequence):** Consider a chain with n states arranged in a line, where each state transitions only to its immediate neighbor(s). For example, a simple chain may have transition probabilities such that for $i = 2, \dots, n - 1$

$$P(i, i-1) = \alpha, \quad P(i, i+1) = 1 - \alpha,$$

with suitable modifications at the boundaries. Under appropriate conditions, such chains can converge to a unique stationary distribution.

Stationary vs. Non-Stationary Markov Chains

A Markov chain is said to be *stationary* if its transition probabilities are independent of time. That is, the one-step transition probability $P(X_{n+1} = j | X_n = i)$ does not vary with n . In contrast, a *non-stationary* Markov chain has time-dependent transition probabilities. In this section we focus on stationary chains, while non-stationary dynamics will be revisited later in the context of diffusion processes in AI.

Mixing and Ergodicity

Informally, a Markov chain $\{X_n\}$ is said to be *mixing* if its distribution converges to a unique stationary distribution π regardless of the initial state. More formally, if the chain is *irreducible* (every state is reachable from every state) and *aperiodic* (the chain does not cycle deterministically), then it is *ergodic*. The **ergodic theorem** guarantees that for any initial distribution, the distribution of X_n converges to π as $n \rightarrow \infty$.

A practical measure of mixing is the *mixing time*, which quantifies how quickly the chain's distribution approaches π . In many applications, a short mixing time is desirable for efficient sampling or rapid convergence in iterative algorithms.

Global Balance and Detailed Balance

For a stationary Markov chain with transition matrix P and stationary distribution π , the *global balance condition* is expressed as

$$\pi(j) = \sum_{i \in \mathcal{S}} \pi(i)P(i,j), \quad \text{for all } j \in \mathcal{S}.$$

A stronger condition is the *detailed balance condition*:

$$\pi(i)P(i,j) = \pi(j)P(j,i), \quad \text{for all } i, j \in \mathcal{S}.$$

Detailed balance implies reversibility and greatly simplifies the design and analysis of Markov chains—especially in Markov Chain Monte Carlo (MCMC) methods where it ensures that the chain converges to a desired target distribution.

One practical strategy to ensure ergodicity (and hence convergence) is to incorporate rejection sampling or loop structures in the chain so that, even if some transitions are rejected, the chain remains irreducible and aperiodic.

Example 7.4.1. Toy Diffusion Model for Image Intensities Consider a simplified model for the forward diffusion process in image generation. Let the pixel intensity take on three discrete values, with the state space $\mathcal{S} = \{0, 1, 2\}$ representing “dark”, “medium”, and “bright” levels, respectively. To mimic the noise injection in diffusion models, we design a non-reversible Markov chain that gradually “blurs” the image by driving the distribution toward a

more uniform (noisy) state. One such toy model is given by the transition matrix:

$$P = \begin{pmatrix} 0.8 & 0.15 & 0.05 \\ 0.2 & 0.6 & 0.2 \\ 0.1 & 0.2 & 0.7 \end{pmatrix}.$$

In this matrix, the probabilities are biased so that, for example, even if the process starts at a well-defined state (say, 0 or 2), repeated transitions gradually mix the state probabilities. Notice that P is not symmetric; hence the detailed balance condition is not satisfied, $\pi(i)P(i,j) \neq \pi(j)P(j,i)$, for any candidate stationary distribution π . This intentional violation of detailed balance reflects the non-equilibrium nature of the forward diffusion process used in modern generative diffusion models.

After several steps, an initial pixel intensity (or image) becomes increasingly randomized, effectively “destroying” the original structure. This toy example thus captures the essential idea of noise injection in diffusion models: as time increases, the state distribution diffuses toward a uniform or nearly uniform distribution. The reverse process—recovering the original image from this noisy state—is discussed in later sections.

Exercise 7.4.1. *Analyzing the Mixing Behavior of a Toy Diffusion Process* Consider the toy diffusion model for image intensities described in the previous example with state space $\mathcal{S} = \{0, 1, 2\}$ and transition matrix

$$P = \begin{pmatrix} 0.8 & 0.15 & 0.05 \\ 0.2 & 0.6 & 0.2 \\ 0.1 & 0.2 & 0.7 \end{pmatrix}.$$

- (a) Compute the one-step transition probabilities from each state and verify that the chain is irreducible.
- (b) Explain why this Markov chain does not satisfy the detailed balance condition, and discuss the implications of breaking detailed balance in the context of non-equilibrium diffusion processes.
- (c) Suppose you initialize a pixel with a fixed intensity (e.g., state 2). Simulate two steps of the chain and compute the distribution over states. How does the distribution evolve toward a more mixed state?
- (d) Discuss how short mixing times are critical in practical AI applications such as sampling in diffusion models, and briefly describe a practical criterion (for example, based on the spectral gap of P) to assess if the Markov chain mixes well.

This exercise links the theory of Markov chains with AI by providing a concrete, albeit simplified, illustration of the forward diffusion process used in image generative models. In subsequent sections, we will build on these ideas to develop and analyze the reverse (denoising) process in diffusion models.

In the next section, we will build on these ideas by introducing MCMC methods that leverage detailed balance to ensure convergence to a specified distribution.

Perron–Frobenius Theorem for Markov Chains and Ergodicity

For a finite Markov chain with state space \mathcal{S} and transition matrix P (with nonnegative entries and rows summing to one, also called double-stochastic), the Perron–Frobenius theorem provides a concrete guarantee tailored to our needs. In particular, if P is irreducible (every state is reachable from every other state) and aperiodic (the chain does not cycle deterministically), then:

- The largest eigenvalue of P is $\lambda_1 = 1$.
- There exists a unique stationary distribution π with strictly positive entries such that

$$P\pi = \pi.$$

This result implies that the Markov chain is *ergodic*; regardless of the initial state, the chain converges to the unique stationary distribution π as the number of steps increases. In practical terms, this means that iterative procedures based on P will converge to a unique fixed point.

Furthermore, the convergence rate is governed by the *spectral gap* of P , defined as

$$\text{gap} = 1 - \max\{|\lambda_2|, |\lambda_3|, \dots\},$$

where λ_2 is the second largest eigenvalue in magnitude. A larger spectral gap indicates a faster mixing time, which is crucial for the efficiency of many algorithms in AI.

Exercise 7.4.2. *Analyzing Convergence of a PageRank-Type Algorithm via Perron–Frobenius*
Consider a web graph with three pages, and assume that the following matrix P represents the (row-stochastic) transition probabilities between these pages:

$$P = \begin{pmatrix} 0.05 & 0.90 & 0.05 \\ 0.80 & 0.10 & 0.10 \\ 0.10 & 0.00 & 0.90 \end{pmatrix}.$$

- (a) Verify that P is irreducible and aperiodic.
- (b) Does the algorithm reach the stationary distribution? Is it unique?
- (c) What is the rate of the power method == an iterative algorithm used to approximate the dominant eigenvector of P – convergence? How does it depend on the spectral gap, defined as $\text{gap} = 1 - |\lambda_2|$, where λ_2 is the second largest eigenvalue in magnitude of P . The power method starts from an arbitrary positive vector $\pi^{(0)}$, and iterates $\pi^{(k+1)} = P\pi^{(k)}$, normalizing the result at each step.

7.4.1 Arrow of Time and Dynamic Programming

A central aspect of Markov chains (the general topic of this section) is that they evolve forward in time in a manner that depends only on the present state. This “memoryless” property – the defining feature of Markov processes – enables us to make predictions one step

at a time, building from the past to the future without needing a full history. However, many algorithmic approaches in probability, statistics, and AI (including reinforcement learning) adopt a *backward* or *reverse* viewpoint: they solve a problem starting from some terminal or boundary condition in the future and proceed *back* through time. Reconciling these two perspectives – the forward Markov property vs. the backward solution method – is at the heart of the “arrow of time” concept in Dynamic Programming (DP).

Computational Insight: If one were to naively enumerate all possible trajectories over T time steps, the total number of multi-step scenarios could grow exponentially in T (for instance, if each state or action can branch into multiple futures). DP – focusing on computing marginal probability of available state at the current time – circumvents this combinatorial explosion. By exploiting the Markov property – the fact that each new state’s distribution or value depends only on the *immediately preceding* state (and possibly an action) – DP methods break the problem down into *one-step-at-a-time* recurrences. This structure reduces the complexity from potentially exponential to a computational effort on the order of $O(T)$ (or $O(\text{states} \times T)$ in a discrete setting), demonstrating a remarkable and powerful gain in efficiency.

Forward vs. Backward Evolution in Markov Chains

Recall from earlier in this section that a (discrete-time) Markov chain is given by a sequence of random variables

$$X_0, X_1, X_2, \dots$$

with the property

$$P(X_{n+1} = x \mid X_n, X_{n-1}, \dots, X_0) = P(X_{n+1} = x \mid X_n).$$

In other words, once X_n is known, the next state X_{n+1} is conditionally independent of all previous states $\{X_{n-1}, \dots, X_0\}$. This supports a *forward-in-time* approach to sampling or generation: given X_n , we sample X_{n+1} , then from that X_{n+2} , and so on.

Meanwhile, many classical algorithms from control, optimization, and reinforcement learning — for instance, *Bellman’s principle of optimality* or *Q*-learning (see, e.g., [48]) — adopt a *backwards* viewpoint. They start with a “final cost” or “terminal condition” at the end of the horizon (time T) and then step *back* in time, computing the best decisions, predicted marginal probabilities, costs, or value functions at each earlier time. This works precisely because the Markov property still supports a recursion in the backward direction, once the final reward or cost is known. This combination of a forward Markov property with a backward recursion is sometimes called the “arrow of time” paradox: the system evolves forward, yet the best way to compute certain functions of interest (e.g., the optimal cost-to-go) is often to proceed backward.

Bellman’s Equation and the Dynamic Programming Principle

The DP setting applies equally to the case when we sum over the states (i.e., marginalizing probabilities) or optimize (e.g., maximizing over the most probable states or minimizing expected costs). More generally, in Markov Decision Processes (MDPs), we optimize over

actions at each step. (For a more extended discussion of MDPs in a reinforcement-learning context, see Section 9.4.2.)

Marginalization. In the setting of *marginalization* over a Markov chain, one might be interested in how the distribution of X_{n+1} depends on X_n . Then at each step we have

$$P(X_{n+1} = x \mid X_n) \implies P(X_{n+1} = x) = \sum_y \mathbb{P}(X_{n+1} = x \mid X_n = y) \mathbb{P}(X_n = y).$$

Such *summing or integrating over the next states* proceeds naturally forward, but can also be reversed under certain conditions if we know $\mathbb{P}(X_{n+1})$ at a future time.

Optimization / MDP Setting. In a *Markov Decision Process* (MDP), the state X_n evolves according to both the current state and an action a from some set $\mathcal{A}(x)$. We typically have a one-step cost (or reward) function $r(x, a)$, and a discount factor $\gamma \in (0, 1]$. The *value function* $V_n(x)$ measures the best achievable expected cost (or reward) from time n onward, starting at state x . Bellman's equation then expresses how V_n relates to V_{n+1} . Concretely:

$$V_n(x) = \max_{a \in \mathcal{A}(x)} \left\{ r(x, a) + \gamma \mathbb{E}[V_{n+1}(X_{n+1}) \mid X_n = x, a] \right\},$$

if we are maximizing expected returns, or

$$V_n(x) = \min_{a \in \mathcal{A}(x)} \left\{ r(x, a) + \gamma \mathbb{E}[V_{n+1}(X_{n+1}) \mid X_n = x, a] \right\},$$

for a minimization scenario. (For instance, in control problems, one may want to minimize cost.) From a forward perspective, X_{n+1} “emerges” once X_n and an action a are chosen. However, the *optimal* action at $X_n = x$ is usually computed by proceeding *backward* from time T , where $V_T(x)$ is specified by a terminal condition (often zero for a final cost), and iterating down to $n = 0$. This tension between forward Markov evolution and backward solution underscores the arrow-of-time concept in DP.

Connections to Earlier Material.

- **Sampling and Markov Chains (Sections 7.1–7.3)**

The same Markov property that enables efficient *forward* sampling — for instance, in Markov Chain Monte Carlo (MCMC) — also enables *backward* recursion. Once the final time’s distribution or cost is known, one can “unroll” transitions in reverse to compute conditional expectations or costs.

- **Stochastic Processes (Sections 7.1–7.3).**

Whether we deal with discrete-time Markov chains or continuous-time processes (e.g. Brownian motion), we typically know how $\{X_t : t \geq 0\}$ evolves forward. The arrow of time emerges when solving inference or optimization problems from $t = T$ backward.

- **Entropy and Information (Chapter 6)**

Directionality of information flow (conditioning on the past vs. conditioning on the future) affects how we measure uncertainty and mutual information. While we often talk about “predicting the future,” DP shows that some decisions can be computed more directly from a known future boundary condition, stepping backward.

- **Optimization Over Time (Chapter 3).**

Gradient-based methods for sequence models (like RNNs or Transformers) employ *backpropagation through time* — a continuous or algorithmic analog of the discrete DP approach. Once again, this is a backward pass, while the data (or physical system) evolves forward.

Example 7.4.2 (Shortest Path (Optimization)). *A classic example is the shortest-path problem from node s to node t in a graph, viewed as a dynamic programming task:*

1. *Label each node x by $V(x)$, the minimum cost to reach t from x .*
2. *At the terminal node t , set $V(t) = 0$.*
3. *For each node $x \neq t$,*

$$V(x) = \min_{(x \rightarrow y)} \{ c(x, y) + V(y) \},$$

where $c(x, y)$ is the edge cost from x to y .

4. *If there is a Markov-chain interpretation (e.g. each step picks an edge with some probability), then deterministic shortest paths become a special case of the more general DP recursion for expected cost.*

Even though the path is physically traversed forward from s to t , the dynamic program is typically solved backward (from t to all other nodes), illustrating the forward–backward duality.

Exercise 7.4.3 (Implementing a Markov-Chain DP). *Consider a simple Markov chain for routing in a small network, where each node x can transition to a set of neighbors $\{y_1, y_2, \dots\}$ with given probabilities. Suppose your goal is to find:*

- (a) *The distribution over states at the final time T , by marginalizing forward (summing over transitions).*
- (b) *The minimum (or maximum) expected cost to reach a target node t , by backward recursion in the sense of dynamic programming.*

Tasks:

1. *Write a short Python or Julia script that:*

- (a) *Takes an $n \times n$ transition matrix P for the Markov chain and a cost function $c(x)$ (or $c(x, y)$ for each edge).*

- (b) *Initializes either a distribution (for part (a)) or a value function $V(x)$ (for part (b)) at the terminal time T .*
 - (c) *Iterates forward (part (a)) or backward (part (b)) for all times $0, \dots, T$ to compute the final distribution (a) or the final cost $V_0(x)$ (b).*
2. *Experiment with a small example (e.g. a 4-node chain or grid) and visualize your results. Which states have the largest cost or probability mass at different times?*
 3. *Discuss how changing the terminal-time condition (e.g. T larger vs. smaller) affects the result.*

You may wish to incorporate the shortest path viewpoint (similar to Example 7.4.2) or the reinforcement learning setting (with a discount factor $\gamma < 1$). Provide commentary on how the forward vs. backward directions compare in terms of ease of implementation and insight.

7.5 Markov Chains Meet Sampling: MCMC

Markov Chain Monte Carlo (MCMC) methods constitute a family of algorithms that generate samples from complex probability distributions by constructing a Markov chain whose stationary distribution coincides with the target distribution. Among these methods, Gibbs sampling and the Metropolis–Hastings algorithm are two of the most prominent. In this section, we describe these methods and illustrate their application in an AI context using the representative example of the Ising model over a bi-partite graph – which is also called Restricted Boltzmann Machines (RBMs). RBMs played a pivotal role in AI because it enables learning and reconstruction under partial observability. (RBMs and their multi-layer generalizations, leading to Bayesian networks, will be discussed in more details the next chapter.)

A fundamental distinction between MCMC methods and direct sampling is that direct sampling generates independent and identically distributed (i.i.d.) samples from the target distribution, whereas MCMC methods produce a sequence of samples that are inherently correlated since each new sample is derived from the previous state. To mitigate this dependence, it is common practice to extract samples that are sufficiently separated – by at least the MC’s mixing time – so that they can be treated as approximately independent. Despite this need for the so-called "thinning", MCMC methods are celebrated for their simplicity and asymptotic exactness: as the number of samples tends to infinity, empirical estimates computed from the MCMC-generated sequence converge to the true expectations of the target distribution.

Gibbs Sampling

Gibbs sampling is a specialized MCMC technique where each variable is updated sequentially by sampling from its conditional distribution given the current states of all other variables. This approach circumvents the need to sample directly from the full joint distribution – a task that is often computationally prohibitive due to the high-dimensional integrals or sums involved.

Illustrative Example: The Ising Model Consider an Ising model defined on a graph, $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, with nodes $i \in \mathcal{V}$, where each node carries a spin $x_i \in \{-1, +1\}$, and (undirected) edges $(i, j) \in \mathcal{E}$. The energy function is given by

$$E(x) = - \sum_{(i,j) \in \mathcal{E}} J_{ij} x_i x_j - \sum_{i \in \mathcal{V}} h_i x_i,$$

with \mathcal{E} representing the set of edges, J_{ij} the coupling coefficients, and h_i the external fields. The corresponding Boltzmann distribution is

$$p(x) = \frac{1}{Z} \exp(-E(x)).$$

Thanks to the factorization properties inherent in the Ising model, the conditional probability for a single spin given its neighbors is straightforward to compute:

$$p(x_i | x_{\mathcal{N}(i)}) = \frac{1}{1 + \exp(-2x_i \left(\sum_{j \in \mathcal{N}(i)} J_{ij} x_j + h_i \right))},$$

where $\mathcal{N}(i) := \{j | (i, j) \in \mathcal{E}\}$. Algorithm 1 outlines a pseudo-code implementation of Gibbs sampling for the Ising model on an arbitrary graph.

Algorithm 1 Gibbs Sampling for the Ising Model

Require: Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with couplings $\{J_{ij}\}$ and external fields $\{h_i\}$; initial configuration $x_i^{(0)} = \{x_i^{(0)} \in \{-1, +1\} : i \in \mathcal{V}\}$; number of iterations T .

- 1: **for** $t = 0, 1, \dots, T - 1$ **do**
- 2: **for** each node $i \in \mathcal{V}$ (in any order or randomly) **do**
- 3: Compute the local field:

$$H_i = \sum_{j \in \mathcal{N}(i)} J_{ij} x_j^{(t)} + h_i.$$

- 4: Update the spin:

$$x_i^{(t+1)} = \begin{cases} +1, & \text{with probability } \frac{1}{1+\exp(-2H_i)}, \\ -1, & \text{with probability } \frac{\exp(-2H_i)}{1+\exp(-2H_i)}. \end{cases}$$

- 5: **end for**
 - 6: **end for**
 - 7: **return** Final configuration $s^{(T)}$.
-

Metropolis–Hastings Sampling

An alternative MCMC approach is provided by the Metropolis–Hastings (MH) algorithm, which relies on generating candidate states from a proposal distribution $q(\cdot | \cdot)$. Given the

current state x , a candidate x' is drawn from $q(x' | x)$ and is accepted with probability

$$\alpha(x, x') = \min\left\{1, \frac{p(x') q(x | x')}{p(x) q(x' | x)}\right\}.$$

If the candidate is rejected, the chain remains in state x . This acceptance/rejection mechanism is crucial as it enforces the detailed balance condition, ensuring that, over an infinite number of samples, the chain converges to the desired target distribution $p(x)$. A key advantage of the MH algorithm is that it circumvents the need for explicit computation of the partition function (i.e., the normalization constant) of the target distribution $p(\cdot)$. This is particularly beneficial when the partition function is intractable, allowing for effective sampling without requiring its direct evaluation.

Algorithm 2 presents a pseudo-code implementation of the MH algorithm for the Ising model.

Algorithm 2 Metropolis–Hastings Sampling for the Ising Model

Require: Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with couplings $\{J_{ij}\}$ and external fields $\{h_i\}$; initial configuration $x^{(0)}$; number of iterations T .

- 1: **for** $t = 0, 1, \dots, T - 1$ **do**
- 2: Randomly select a node $i \in \mathcal{V}$.
- 3: Propose a flip of its spin: $x'_i = -x_i^{(t)}$; let x' be the resulting configuration.
- 4: Compute the energy difference:

$$\Delta E = E(x') - E(x^{(t)}).$$

- 5: Accept x' with probability

$$\alpha = \min\left\{1, \exp(-\Delta E)\right\}.$$

- 6: **if** $U(0, 1) < \alpha$ **then**
 - 7: Set $x^{(t+1)} = x'$.
 - 8: **else**
 - 9: Set $x^{(t+1)} = x^{(t)}$.
 - 10: **end if**
 - 11: **end for**
 - 12: **return** Final configuration $x^{(T)}$.
-

Restricted Boltzmann Machines (RBMs)

Restricted Boltzmann Machines (RBMs) are energy-based models characterized by their bipartite structure, which comprises a visible layer v and a hidden layer h . The term “restricted” refers to the absence of intra-layer connections; only inter-layer interactions are

allowed. This design is particularly significant in AI, as it facilitates learning and reconstruction when training data are available only at the visible layer. The joint distribution of an RBM is defined as

$$P_\theta(v, h) = \frac{1}{Z_\theta} \exp(-E_\theta(v, h)),$$

with an energy function typically given by

$$E(v, h) = -b^\top v - c^\top h - v^\top Wh, \quad \theta := (W, c, b).$$

Due to the bipartite structure, the conditional distributions factorize neatly:

$$P_\theta(h | v) = \prod_{j=1}^m \sigma(c_j + (W^\top v)_j), \quad P_\theta(v | h) = \prod_{i=1}^d \sigma(b_i + (Wh)_i),$$

where $\sigma(x) = \frac{1}{1+\exp(-x)}$ is the sigmoid function. In the RBM framework, the Gibbs sampler alternates between sampling h given v and sampling v given h . Note that when the Gibbs sampling is used for learning, it is executed for only a few iterations before updating the model parameters (the learning update) – a strategy known as *Contrastive Divergence (CD)*. This truncated sampling process has been instrumental in enabling efficient learning in RBMs. In the next chapter – and specifically in Section 8.2.3 – we will delve deeper into RBMs in a learning context.

Combining Gibbs and Metropolis–Hastings: Glauber Dynamics

In practice, it can be advantageous to blend elements of both Gibbs sampling and the Metropolis–Hastings algorithm to design more effective sampling strategies. Glauber dynamics is one such hybrid method where the state of a single variable is updated (as in Gibbs sampling) while also incorporating an acceptance/rejection step (as in Metropolis–Hastings) to ensure the correct stationary distribution is achieved. This approach is particularly useful when the conditional probabilities are not available in closed form.

Example 7.5.1. *Comparison of Gibbs and Glauber Dynamics in RBMs* The accompanying notebook `RBM-MCMC.ipynb` compares two MCMC schemes:

- **Gibbs Sampling:** The function `gibbs_sampler` alternates between sampling the hidden and visible layers using their respective conditional distributions.
- **Local MH Sampler (Glauber Dynamics):** The function `mh_sampler_local` proposes a change by flipping a single randomly chosen spin (in either the visible or hidden layer). This leads to smaller, more localized moves, which typically result in a higher acceptance rate.
- **Cumulative Average Energy:** Both samplers track the running average of the instantaneous energy. After discarding an initial burn-in period and applying thinning, the cumulative average energy is plotted. When the chain has reached its stationary distribution, these averages should converge to the same constant.

In the observed results, the cumulative average energies do not align perfectly at first; however, alignment improves as the number of samples increases. Notably, the changes induced by the Glauber dynamics tend to be more erratic compared to the smoother transitions seen in the Gibbs sampler.

Exercise 7.5.1. MCMC Methods in RBMs: Gibbs vs. Metropolis–Hastings Consider the RBM setting and extend the `RBM-MCMC.ipynb` notebook by implementing a Metropolis–Hastings algorithm with a non-local proposal mechanism. Specifically:

- (a) Compare the performance of your new non-local MH algorithm with both Glauber dynamics and the standard Gibbs sampler.
- (b) Explore additional evaluation tasks, such as designing tests of convergence (e.g., using auto-correlation functions as a metric for mixing time) to assess the decay of correlations.

In summary, the methods presented in this section demonstrate how MCMC techniques – via both Gibbs sampling and the Metropolis–Hastings algorithm – serve as powerful tools for sampling from high-dimensional distributions in AI applications. Their use in models such as the Ising model and RBMs facilitates approximate inference and gradient estimation, paving the way for advanced learning methods that we will explore in the next chapter.

7.6 Beyond Markov via Auto-Regressive Modeling

In classical Markov chains the future state depends only on the current state, and the state space is fixed. In contrast, autoregressive models – such as those employed in modern transformers (see section 1.1.5) – generate sequences by predicting one token at a time. In these models, the state at time t is defined by the entire sequence of previously generated tokens:

$$s_t = [x_1, x_2, \dots, x_t].$$

Thus, the probability distribution for the next token is given by

$$P(x_{t+1} \mid s_t) = P(x_{t+1} \mid x_1, x_2, \dots, x_t),$$

and the joint probability of a sequence of T tokens factors as

$$P(x_1, x_2, \dots, x_T) = \prod_{t=1}^T P(x_t \mid x_1, \dots, x_{t-1}).$$

Unlike a standard Markov chain with a fixed state space, here the “state” grows with each additional token. In this sense, an autoregressive model is an unusual Markov process with a dynamically expanding state space.

7.6.1 Randomness in Next Token Generation

Although the model's prediction is inherently probabilistic, practical generation requires selecting a specific token. Several sampling strategies are commonly used to incorporate controlled randomness into the generation process:

1. **Greedy Sampling and Temperature Scaling:** The simplest strategy selects the token with the highest probability:

$$x_{t+1} = \arg \max_w P(w | s_t).$$

Alternatively, one can convert token probabilities into an energy function $E(w)$ by taking the negative logarithm, and then scale the energy by a temperature parameter $T > 0$ before applying the softmax. This scaling adjusts the sharpness of the resulting probability distribution:

$$P(x_{t+1} = w | s_t) = \frac{\exp(-E(w)/T)}{\sum_{w'} \exp(-E(w')/T)}.$$

A low temperature (i.e., $T \rightarrow 0$) leads to nearly deterministic (greedy) sampling, while a higher temperature yields more diverse outputs.

2. **Top- k Sampling:** In top- k sampling, only the k most likely tokens are considered. After truncating the distribution, sampling is performed over this reduced set:

$$P_{\text{top-}k}(x_{t+1} = w | s_t) = \frac{P(w | s_t) \mathbf{1}\{w \in \mathcal{K}_k\}}{\sum_{w' \in \mathcal{K}_k} P(w' | s_t)},$$

where \mathcal{K}_k is the set of k tokens with highest probability and $\mathbf{1}\{\cdot\}$ is the indicator function.

3. **Nucleus (Top- p) Sampling:** Rather than using a fixed k , one may choose the smallest set \mathcal{K}_p such that

$$\sum_{w \in \mathcal{K}_p} P(w | s_t) \geq p,$$

where $p \in (0, 1)$. Sampling is then performed over \mathcal{K}_p , allowing for a dynamic cutoff based on the confidence of the prediction.

Each of these strategies introduces stochasticity into the generation process, balancing between determinism and randomness, and enabling the model to produce coherent yet diverse sequences.

7.6.2 Auto-Regressive Modeling as an Expanding Markov Process

While the classical Markov property assumes a fixed state, in an autoregressive transformer the state s_t expands by one token at each step. This process can be described as a Markov process with a non-stationary state space:

- **State Evolution:** The state at time $t + 1$ is formed by appending the new token x_{t+1} to the previous state s_t :

$$s_{t+1} = s_t \oplus x_{t+1}.$$

- **Revisited Markov Property:** The next token depends solely on s_t (i.e., $P(x_{t+1} | s_t)$); however, the dimensionality of s_t increases with t . This “growing memory” is crucial for capturing long-range dependencies.

This interpretation connects with our earlier discussions on Markov chains (see Sections 7.4, 7.5) while highlighting the novel aspects of autoregression in deep generative models.

Example 7.6.1. *minGPT and Sampling Strategies* One practical illustration is the minGPT model by Karpathy [49], a minimal implementation of a GPT-style transformer. In minGPT, the next token is predicted by

$$P(x_{t+1} | s_t) = \text{softmax}(W \cdot h(s_t) + b),$$

where $h(s_t)$ is the hidden state derived from self-attention and feed-forward transformations applied to the input sequence s_t .

Sampling in minGPT: The sampling function in minGPT is typically implemented to support:

- **Temperature Scaling:** Modifying the sharpness of the output probability distribution.
- **Top- k Sampling:** Restricting token choices to the top k most likely candidates.
- **(Optional) Nucleus Sampling:** Extending the implementation to include top- p sampling.

These mechanisms help control the diversity of the generated text and provide practical means for exploring the trade-off between randomness and determinism.

Exercise 7.6.1. minGPT Sampling Exploration:

1. **Implement Sampling Variants:** Using the provided minGPT code, modify the sampling function to support:

- **Temperature Scaling:** Let the user specify a temperature T and sample using

$$P(x_{t+1} = w | s_t) = \frac{\exp(-E(w)/T)}{\sum_{w'} \exp(-E(w')/T)}.$$

- **Top- k Sampling:** Restrict the prediction to the k tokens with highest probabilities.
- **Nucleus (Top- p) Sampling:** Identify the smallest set \mathcal{K}_p such that

$$\sum_{w \in \mathcal{K}_p} P(w | s_t) \geq p,$$

and sample from \mathcal{K}_p .

2. Compare Outputs: Generate text samples using different values of T , k , and p . Analyze:

- How the diversity of generated text changes.
- The impact on coherence and fluency.

3. Reflect on the Markov Process: Write a brief explanation (in your notebook) on how the expanding state s_t enables the model to capture long-range dependencies and how the sampling strategies inject controlled randomness into this non-stationary Markov process.

This section bridges the classical theory of Markov processes with modern autoregressive generation techniques. By connecting with earlier discussions on transformers (Section 1.1.5) and Markov chains – Sections 7.4, 7.5 – and by providing concrete examples from minGPT, it offers both theoretical insight and practical exercises for further exploration.

Chapter 8

Energy Based (Graphical) Models

In this chapter we introduce a unifying framework for generative AI that centers on energy-based (graphical) models. At its core, the approach represents probability distributions in terms of an energy function – essentially, the (negative) logarithm of the unnormalized probability. This function encapsulates the interactions and dependencies among components of the model, often expressed through a factorization that can naturally be depicted as a graph. Such factorizations may mirror structures encountered in neural networks, physical constraints (e.g., equalities and inequalities), and inherent "physical" symmetries, offering a versatile way to design and interpret complex models.

In the context of AI and Machine Learning, the term energy borrows intuition from physics, where it typically quantifies the potential or configuration cost of a system. This connection was first introduced in our discussion of Classical Mechanics in Chapter 2, where the motion of particles was governed by Ordinary Differential Equations (ODEs) derived from energy principles, such as kinetic and potential energy. In AI, especially in statistical and probabilistic modeling, energy is used metaphorically to define a scalar function over configurations of interest — such as image pixels, binary variables, or node states in a graph — where lower energy corresponds to higher probability. This idea is formalized in Chapter 7, where we introduced Markov Chains (MC), Markov Chain Monte Carlo (MCMC) methods (over discrete spaces), and Stochastic ODEs (over continuous spaces), and derived the steady-state (Boltzmann–Gibbs) distribution from the Perron–Frobenius (ergodicity) theorem and the Langevin and Fokker–Planck formalisms. For instance, in the Ising model, configurations with aligned spins (i.e., lower energy) are exponentially more probable under the Boltzmann–Gibbs distribution. This dual role of energy – as both a physical and statistical construct – forms the foundation of energy-based models in AI, where inference and learning tasks are framed as optimization or sampling problems over such energy landscapes. These ideas also play a crucial role in modern generative modeling frameworks, including score-based diffusion models, which are explored in Chapter 9.

A central theme of our exposition in this Chapter is the duality between learning and inference. Conventionally, one might think of first training a model – learning its parameters from data – and then using it to make predictions or generate samples (inference). However, our perspective here is deliberately reversed: we begin by discussing inference, that is, how to use a model to sample, solve downstream tasks, or extract meaningful structure, and then turn to learning. This reversal is not merely a change in presentation but reflects a

deeper construction principle: the very design of the learning process is guided by the needs of inference. In other words, the tasks we expect the model to perform (inference) impose constraints and structure that the learning algorithm must respect.

Drawing on Chapter 11 of [50], we define *learning* as the process of constructing a model – integrating data with additional information or prior knowledge – so as to capture the underlying probability distribution of a system. In contrast, *inference* is the procedure by which this model is employed to generate samples, make predictions, or solve particular problems. Although training (learning) is executed before inference in practice, our reverse narrative here emphasizes that the efficacy of inference critically shapes the choice of model architecture and the formulation of the learning problem.

Energy-based models are especially compelling because they provide a natural language for representing probability distributions even when the normalization constant (the partition function) is unknown or computationally prohibitive. By expressing the model as an energy function (which is defined up to an additive constant), we can sidestep the need for exact normalization while still leveraging powerful factorization techniques. These techniques, when cast into a graphical framework, reveal the modular relationships among the state components – relationships that are central to both the interpretation and the efficient computation of the model.

In the sections that follow, we will explore the construction, interpretation, and application of energy-based (graphical) models in generative AI. We will demonstrate how inference procedures not only extract useful information from a model but also inform the learning process itself, leading to a more cohesive and effective design of AI systems.

8.1 Inference

In this section, we set the stage for what follows by addressing the dual challenges of achieving high-quality inference and managing its inherent computational complexity. In many practical applications, the primary goal is to obtain accurate samples, robust maximum likelihood estimates, and reliable expectation computations. However, most inference tasks in graphical models scale exponentially with the system size. This exponential growth makes exact inference computationally intractable in the majority of cases, forcing us to seek fast, albeit approximate, alternatives.

Overall, this section lays the foundation for understanding the trade-off between inference quality and computational complexity. We begin by reviewing graphical models and clarifying what we mean by inference in this context, and then we motivate the use of variational methods as a practical solution to otherwise intractable problems.

8.1.1 Graphical Models

To be more concrete about the nature of inference, we now briefly review the framework of Graphical Models (GM) – following Section 10.2 of [50]. GMs provide a systematic way to represent complex probability distributions through nodes and edges that encapsulate dependencies among variables. In this context, inference tasks typically include:

- **Sampling:** Generating representative instances from the model.

- **Maximum Likelihood Estimation:** Finding parameter values that maximize the likelihood of the observed data.
- **Expectation Computation:** Calculating statistical summaries such as means or variances.

The most famous graphical model – the Ising Model, and its particular case the Restricted Boltzmann Machine (RBM) – was discussed in Section 7.5 as an example illustrating MCMC approaches to sampling.

Several other classes of graphical models have become central in AI research and applications. Some of the most popular include:

- **Bayesian Networks (Directed Acyclic Graphs):** These models represent the joint probability distribution over a set of variables using directed edges to encode conditional dependencies. They are widely used for causal reasoning, decision making, and diagnostic applications.
- **Markov Random Fields (MRFs):** Also known as undirected graphical models, MRFs capture dependencies among variables via an undirected graph structure. They are extensively applied in image processing, computer vision, and spatial data analysis. In essence, the aforementioned Ising model is the simplest MRF.
- **Conditional Random Fields (CRFs):** These are a type of undirected graphical model, particularly popular in sequence modeling and labeling tasks, such as natural language processing and bioinformatics.
- **Factor Graphs:** These provide a bipartite representation that explicitly shows how a global function factors into a product of local functions. They serve as a unifying framework that underpins many inference algorithms (like belief propagation) used in error-correcting codes and probabilistic reasoning.
- **Hidden Markov Models (HMMs) and Dynamic Bayesian Networks (DBNs):** HMMs are a specialized form of Bayesian networks used for modeling temporal sequences with hidden states. DBNs extend HMMs to more complex, multidimensional temporal data and are widely used in speech recognition and time series analysis.

Each of these models provides a different perspective and set of tools for dealing with uncertainty and complex dependencies in high-dimensional data, which is why they continue to be a backbone in various AI applications.

In the remainder of this section, we focus on two specific classes: Bayesian Networks and Hidden Markov Models also with the reference to Dynamic Programming discussed earlier in Section 7.4.1, discussing their formulation, an example, and an exercise for each.

Bayesian Networks and Directed Acyclic Graphs

Bayesian networks are **Directed Acyclic Graphs (DAGs)** in which nodes represent random variables and directed edges encode conditional dependencies. The joint probability

distribution over the variables X_1, X_2, \dots, X_n factorizes according to the network structure as follows:

$$P(X_1, X_2, \dots, X_n) = \prod_{i=1}^n P(X_i | \text{Pa}(X_i)),$$

where $\text{Pa}(X_i)$ denotes the set of parent nodes of X_i . This factorization parallels the way the Ising model captures interactions via pairwise couplings and how the RBM exploits a bipartite structure, yet in Bayesian networks the directed nature of edges directly models causality and influences among variables.

Example 8.1.1. Consider a simple Bayesian network with three variables A , B , and C where A is a common cause for both B and C (i.e., $A \rightarrow B$ and $A \rightarrow C$). The joint distribution factorizes as:

$$P(A, B, C) = P(A) P(B | A) P(C | A).$$

For instance, if

$$\begin{aligned} P(A = 1) &= 0.3, & P(B = 1 | A = 1) &= 0.8, & P(B = 1 | A = 0) &= 0.2, \\ P(C = 1 | A = 1) &= 0.7, & P(C = 1 | A = 0) &= 0.4, \end{aligned}$$

then the joint probability $P(A = 1, B = 1, C = 1)$ is given by:

$$P(A = 1, B = 1, C = 1) = 0.3 \times 0.8 \times 0.7 = 0.168.$$

Exercise 8.1.1. 1. Extend the network by adding a fourth variable D that is influenced only by B (i.e., $B \rightarrow D$). Write down the new factorization for $P(A, B, C, D)$.

2. Using hypothetical numerical probabilities, compute $P(A = 1 | D = 1)$ via Bayes' rule.
3. Discuss how such a network might be used for causal reasoning in diagnostic systems.

Hidden Markov Models (HMMs)

Hidden Markov Models (HMMs) are statistical models for sequential data in which the system is assumed to follow a Markov process with unobserved (hidden) states. An HMM is defined by:

- A set of hidden states $\{S_t\}_{t=1}^T$ that satisfy the Markov property:

$$P(S_t | S_{t-1}, S_{t-2}, \dots) = P(S_t | S_{t-1}),$$

- A set of observations $\{O_t\}_{t=1}^T$ where each observation is generated from the current hidden state:

$$P(O_t | S_t),$$

- An initial hidden state distribution $P(S_1)$ and a hidden state transition matrix $P(S_t | S_{t-1})$.

Thus, the joint probability over the sequence of hidden states and observations is given by:

$$P(S_1, \dots, S_T, O_1, \dots, O_T) = P(S_1) P(O_1 | S_1) \prod_{t=2}^T P(S_t | S_{t-1}) P(O_t | S_t).$$

This structure is analogous to the layered representation in the RBM where the “hidden” units capture dependencies in the visible layer, but here the sequential aspect is emphasized and exploited for tasks such as speech recognition and time series analysis.

Example 8.1.2. *Imagine an HMM for weather prediction with two hidden states {Sunny, Rainy} and two possible observations {Umbrella, No Umbrella}. Suppose the model parameters are:*

$$P(S_1 = \text{Sunny}) = 0.6, \quad P(S_1 = \text{Rainy}) = 0.4,$$

with transition probabilities:

$$\begin{aligned} P(\text{Sunny} | \text{Sunny}) &= 0.8, & P(\text{Rainy} | \text{Sunny}) &= 0.2, \\ P(\text{Sunny} | \text{Rainy}) &= 0.4, & P(\text{Rainy} | \text{Rainy}) &= 0.6, \end{aligned}$$

and emission probabilities:

$$\begin{aligned} P(\text{Umbrella} | \text{Rainy}) &= 0.9, & P(\text{No Umbrella} | \text{Rainy}) &= 0.1, \\ P(\text{Umbrella} | \text{Sunny}) &= 0.3, & P(\text{No Umbrella} | \text{Sunny}) &= 0.7. \end{aligned}$$

One can compute the likelihood of a particular observation sequence (e.g., “Umbrella, No Umbrella, Umbrella”) using the above factorization.

Implementation of Bayesian Networks

While specialized libraries (e.g., Pyro, PyTorch Probability) exist for probabilistic modeling, understanding how to build a small Bayesian Network “by hand” in PyTorch offers valuable insight into the internal mechanics of:

1. *Parameterizing local conditional distributions* $p(X_i | \text{Pa}(X_i))$ via neural modules.
2. *Sampling*, i.e., forward simulation of the network.
3. *Inference*, i.e., querying probabilities such as $p(X_i | \mathbf{e})$, where \mathbf{e} is evidence for some observed variables.

Such an approach clarifies how we might combine the benefits of neural networks (as function approximators) with classical Bayesian inference strategies (e.g., enumerating or sampling from the underlying DAG).

Exercise 8.1.2 (Extending the Bayesian Network Snippet). *Using the `bn_inference.ipynb` script as your baseline, explore the following tasks:*

A small Bayesian Network with three variables – A, B, C – as well as functions for forward sampling, conditioned sampling, and simple exact inference by enumeration is presented in `bn_inference.ipynb`. Using the `bn_inference.ipynb` script as your baseline, explore the following tasks:

1. **Additional Parents.** Extend the Bayesian Network to include a fourth Bernoulli variable D that depends on (A, B, C) . Introduce suitable parameters $\text{thetaD}[a, b, c] = p(D = 1 \mid A = a, B = b, C = c)$. Implement forward sampling (`sample_forward`) and exact inference (`exact_inference`) for the updated BN.
2. **Custom Probability Distributions.** Instead of Bernoulli random variables, switch one of the variables (say C) to a discrete categorical variable taking $k > 2$ states. Adjust your sampling and inference functions to handle this generalized distribution.
3. **Inference with Partial Observations.** Currently, the BN snippet supports `sample_conditioned(given_dict)`. Extend the exact inference routine so that if some subset of variables is observed (e.g. $B = 1, D = 0$), it returns the conditional distribution for unobserved ones (e.g. $p(A, C \mid B = 1, D = 0)$). You might return a dictionary of joint distributions or a factorized representation.
4. **Comparison of Sampling vs. Exact Enumeration.** Compare the approximate conditional distributions obtained via `sample_conditioned` with the exact results from `exact_inference`. Use various sample sizes to see how quickly sampling approximations converge to the exact solution.
5. **Discussion and Scalability.** In a short discussion, comment on the scalability limitations of exact enumeration (which grows exponentially in the number of variables) vs. sampling-based methods. How might you combine the power of neural networks (for parameterizing complex conditional probabilities) with approximate inference techniques (e.g. MCMC or variational inference)?

8.1.2 Variational Methods

Having already discussed exact approaches — for instance, by leveraging the chain rule for exact sampling or by relying on asymptotically exact Markov Chain Monte Carlo (MCMC) methods — we now acknowledge that the majority of inference tasks in high-dimensional models are formidable. In such cases, exponential complexity often makes these exact approaches computationally prohibitive. To cope with these challenges, *variational inference* provides a practical route by recasting inference as an optimization task. Specifically, one chooses a tractable family of distributions and seeks to approximate the true (often intractable) posterior distribution by minimizing a divergence measure (such as KL divergence) between the two. This approach not only yields computational efficiency but also offers a systematic way to balance approximation quality with speed.

Below, we summarize the variational approaches following closely the material from [51], also reviewed in [50], specifically from Section 10.2.

From Posterior Inference to an Optimization Problem

Let \mathbf{x} denote the variables of interest (e.g. spins in an Ising model or bits in a decoding problem) and define the *posterior*

$$p(\mathbf{x}) = \frac{\tilde{p}(\mathbf{x})}{Z}, \quad Z = \sum_{\mathbf{x}} \tilde{p}(\mathbf{x}).$$

In Bayesian language the term *posterior* simply signals that this distribution already contains *all* information currently available – the prior couplings, external fields and any observations – whereas the *prior* would describe our beliefs before seeing those observations. We call $p(\mathbf{x})$ *intractable* because the partition function Z cannot be computed (or even tightly approximated) in polynomial time for realistic system sizes, making exact normalization or direct sampling infeasible and motivating approximate-inference techniques.

Variational inference posits a **surrogate** (variational) distribution $q(\mathbf{x}|\boldsymbol{\theta})$ – which we may also call **belief** distribution or just belief – selected from a tractable parametric family, and seeks to find the parameters $\boldsymbol{\theta}$ that minimize the Kullback–Leibler (KL) divergence

$$\text{KL}(q(\mathbf{x}|\boldsymbol{\theta}) \| p(\mathbf{x})) = \sum_{\mathbf{x}} q(\mathbf{x}|\boldsymbol{\theta}) \log \frac{q(\mathbf{x}|\boldsymbol{\theta})}{p(\mathbf{x})}, \quad (8.1)$$

or equivalently to maximize the corresponding **Evidence-Based Lower Bound** (ELBO). This setup recasts inference into a familiar optimization framework that can often be tackled via gradient-based or message-passing methods.

Evidence-Based Lower Bound (ELBO)

To clarify why minimizing (8.1) yields a lower bound on the log-partition function (or log-evidence), let us provide a concise proof of the relevant inequality rather than merely recalling it. Suppose we wish to evaluate

$$\log \sum_{\mathbf{x}} \tilde{p}(\mathbf{x}).$$

By multiplying and dividing by any other distribution $q(\mathbf{x})$, we can rewrite this as

$$\log \sum_{\mathbf{x}} \tilde{p}(\mathbf{x}) = \log \sum_{\mathbf{x}} q(\mathbf{x}) \frac{\tilde{p}(\mathbf{x})}{q(\mathbf{x})}.$$

Now let us introduce a distribution $q(\mathbf{x})$ and note that $\sum_{\mathbf{x}} q(\mathbf{x}) = 1$ by definition. Applying Jensen's inequality to the concave function $\log(\cdot)$, we obtain

$$\log \sum_{\mathbf{x}} q(\mathbf{x}) \frac{\tilde{p}(\mathbf{x})}{q(\mathbf{x})} \geq \sum_{\mathbf{x}} q(\mathbf{x}) \log \left[\frac{\tilde{p}(\mathbf{x})}{q(\mathbf{x})} \right]. \quad (8.2)$$

Proof by Jensen's Inequality. Let $\{w(\mathbf{x})\}$ be nonnegative weights such that $\sum_{\mathbf{x}} w(\mathbf{x}) = 1$. Define $f(z) = \log(z)$ as our concave function. Then, by Jensen's inequality,

$$f\left(\sum_{\mathbf{x}} w(\mathbf{x}) z_{\mathbf{x}}\right) \geq \sum_{\mathbf{x}} w(\mathbf{x}) f(z_{\mathbf{x}}).$$

Identifying $w(\mathbf{x}) = q(\mathbf{x})$ and $z_{\mathbf{x}} = \frac{\tilde{p}(\mathbf{x})}{q(\mathbf{x})}$, we see $z_{\mathbf{x}} \geq 0$ and $\sum_{\mathbf{x}} w(\mathbf{x}) z_{\mathbf{x}} = \sum_{\mathbf{x}} q(\mathbf{x}) \frac{\tilde{p}(\mathbf{x})}{q(\mathbf{x})} = \sum_{\mathbf{x}} \tilde{p}(\mathbf{x})$, which is precisely the quantity inside the log on the left-hand side. Hence the inequality (8.2) follows directly.

Rearranging to See the ELBO. From (8.2), rearrange terms to find that

$$\log \sum_{\mathbf{x}} \tilde{p}(\mathbf{x}) \geq \sum_{\mathbf{x}} q(\mathbf{x}) \log \tilde{p}(\mathbf{x}) - \sum_{\mathbf{x}} q(\mathbf{x}) \log q(\mathbf{x}).$$

Note that $\sum_{\mathbf{x}} q(\mathbf{x}) \log \tilde{p}(\mathbf{x}) - \sum_{\mathbf{x}} q(\mathbf{x}) \log q(\mathbf{x})$ can be rewritten using the definition of KL divergence:

$$\sum_{\mathbf{x}} q(\mathbf{x}) \log \tilde{p}(\mathbf{x}) - \sum_{\mathbf{x}} q(\mathbf{x}) \log q(\mathbf{x}) = -\text{KL}(q \parallel \tilde{p}) + (\text{constant}),$$

where the constant depends on the normalization constant for $\tilde{p}(\mathbf{x})$ – that is the partition function, Z . Consequently, minimizing $\text{KL}(q \parallel p)$ is equivalent to maximizing the right-hand side, which we identify as the *evidence-based lower bound* (ELBO). In other words, $-\text{KL}(q \parallel \tilde{p})$ is a lower bound to the log-partition function, and the distribution $q(\mathbf{x})$ that attains the minimum KL divergence provides the best such lower bound within the chosen variational family.

Thus, the objective $\text{KL}(q(\mathbf{x}) \parallel \tilde{p}(\mathbf{x}))$ represents a practical target for approximate inference in situations where directly computing or maximizing $Z = \sum_{\mathbf{x}} \tilde{p}(\mathbf{x})$ (the partition function) is infeasible.

Mean-Field Approximation on the Ising Model

Let us now illustrate Eq. (8.1) by specializing the variational surrogate $q(\mathbf{x})$ to a specific choice known as the *mean-field* (MF) ansatz.

Why “Mean-Field”? The name “mean-field” traces back to physics, where the original intuition emerged from treating each spin or particle as if it experiences only an *average* (mean) effect from all other particles instead of fully capturing detailed fluctuations or correlations. In statistical mechanics, this approximation often becomes asymptotically exact in the so-called thermodynamic limit, where the size of the system grows unbounded and collective behavior can be effectively captured by averaged interactions.

Consider the Ising model – introduced earlier in Section 7.5) on a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where each node $i \in \mathcal{V}$ has a spin variable $x_i \in \{-1, +1\}$. The joint distribution, up to a normalization constant, is given by

$$p(\mathbf{x}) \propto \exp\left(\sum_{(i,j) \in E} J_{ij} x_i x_j + \sum_{i \in V} h_i x_i\right). \quad (8.3)$$

A particularly simple variational ansatz is the *mean-field* approximation:

$$q(\mathbf{x}) = \prod_{i \in V} q_i(x_i), \quad (8.4)$$

where each spin is treated as if it were statistically independent, having its own one-site marginal $q_i(x_i)$. In practice, one often parameterizes $q_i(x_i)$ in terms of the *magnetization* $m_i \in [-1, 1]$, via

$$q_i(x_i = +1) = \frac{1+m_i}{2}, \quad q_i(x_i = -1) = \frac{1-m_i}{2}.$$

KL Divergence under Mean-Field. Substituting (8.4) into the KL-divergence objective (8.1) and recalling that $p(\mathbf{x}) \propto \exp(\sum_{(i,j)} J_{ij} x_i x_j + \sum_i h_i x_i)$, one obtains

$$\text{KL}(q \| p) = \sum_{\mathbf{x}} \left(\prod_i q_i(x_i) \right) \log \left[\frac{\prod_i q_i(x_i)}{\exp(\sum_{(i,j)} J_{ij} x_i x_j + \sum_i h_i x_i)} \right] + (\text{const}).$$

Rewriting in terms of expectations over the single-site marginals $q_i(x_i)$ yields the well-known *mean-field self-consistency equations*:

$$m_i = \tanh(h_i + \sum_{j \in \partial i} J_{ij} m_j), \quad \forall i \in V, \quad (8.5)$$

where ∂i denotes the set of neighbors of node i in the graph. Solving (8.5) numerically provides a stationary point in the variational parameters $\{m_i\}$, hence yielding the mean-field approximation $q(\mathbf{x})$. Because this approximation factorizes all spins, it typically underestimates true correlations $\langle x_i x_j \rangle$, especially when the couplings J_{ij} are large or the graph contains loops.

Exactness and Thermodynamic Limit. In finite-size systems, mean-field is rarely exact, since real-world (or simulated) spins exhibit correlations that the product ansatz omits. However, in certain limiting regimes – for instance, as the system size $|V| \rightarrow \infty$ with suitably weak or dense couplings – the mean-field description can become asymptotically exact. Theoretical aspects of such regimes are discussed extensively in the literature on statistical physics, e.g., [52] and [51]. In practice, mean-field often serves as a fast baseline or initialization method before more refined approaches are used.

In what follows, we will see how one can improve upon this factorized approximation by allowing small subsets of variables (e.g., pairs) to be jointly modeled, leading to Belief Propagation and the Bethe approximation.

Belief Propagation (BP): Factorization via Marginals

A more refined variational method is Belief Propagation (BP), sometimes called the Bethe approximation. Instead of factorizing over single nodes, one factorizes over small subsets (e.g., edges) in a factor graph. For the Ising model in (8.3), one treats each edge $(i, j) \in E$ as a two-site factor, thereby incorporating pairwise correlations:

$$q(\mathbf{x}) = \prod_{(i,j) \in E} q_{ij}(x_i, x_j) / \prod_{i \in V} q_i(x_i)^{d_i - 1}, \quad (8.6)$$

where $q_{ij}(x_i, x_j)$ is the two-site marginal and $q_i(x_i)$ is the single-site marginal, with d_i the degree of node i . The exponents $(d_i - 1)$ in the denominator compensate for multiple over-counting in the product of pairs.

Exactness on a Tree. When the underlying graph G is a tree (i.e. loop-free), the Bethe or BP factorization (8.6) is *exact*, yielding the true posterior. A simple way to see this is by inductively constructing the solution from smaller trees:

1. *Base case (single node).* If $|V| = 1$, there are no edges. In this trivial graph, the Bethe factorization reduces to a single-site marginal $q_i(x_i)$ that must match the exact posterior. No overcounting arises, so exactness is manifest.
2. *Inductive step (adding a leaf).* Assume exactness for any tree with $n - 1$ nodes. Consider a tree with n nodes and $n - 1$ edges. Pick any leaf node ℓ whose only neighbor is k . In the BP factorization, the edge term $q_{\ell k}(x_\ell, x_k)$ couples only these two sites, while all other edges remain unaffected. One can remove ℓ and its edge to reduce the problem to a tree with $n - 1$ nodes. By the induction hypothesis, the factorization there is exact. Restoring ℓ and its single edge then amounts to adding a single factor $q_{\ell k}(x_\ell, x_k)$ and the single-site term $q_\ell(x_\ell)$. A counting argument shows that ℓ 's contribution is accounted for *exactly once* in the product of pairwise marginals, once the single-site marginal $q_\ell(x_\ell)$ is included with exponent $(d_\ell - 1) = 0$. Thus consistency with the exact posterior is preserved.
3. *Consequence.* Repeating this leaf-removal argument from base up to a tree of any size $|V|$ confirms that the Bethe factorization perfectly reconstructs the exact distribution. In more physical terms, there is no “overcounting” on a loop-free graph, and so the Bethe free-energy functional coincides with the true free energy.

Consequently, on a tree, solving for q_{ij} and q_i that minimize the Bethe free energy yields the unique exact solution for $p(\mathbf{x})$. This underlies why belief propagation (BP) is guaranteed to converge to the exact marginals on trees.

BP Equations on the Ising Model: Message-Passing View. The resulting BP algorithm – sometimes also called the *sum-product* algorithm – can be expressed in terms of messages that pass between nodes along edges. For an edge (i, j) , let $m_{i \rightarrow j}(x_j)$ be the (unnormalized) *message* from node i to j , capturing i 's beliefs about x_j . One may derive the following iterative updates:

$$m_{i \rightarrow j}(x_j) \leftarrow \sum_{x_i \in \{-1, +1\}} \exp\left(J_{ij} x_i x_j + h_i x_i\right) \prod_{k \in \partial i \setminus j} m_{k \rightarrow i}(x_i), \quad (8.7)$$

where $\partial i \setminus j$ denotes neighbors k of i except for j . Once these messages converge, one obtains the single-site marginals via

$$q_i(x_i) \propto \exp(h_i x_i) \prod_{k \in \partial i} m_{k \rightarrow i}(x_i),$$

and the pairwise marginals from $q_i(x_i)q_j(x_j)$ times an additional factor from $J_{ij}x_i x_j$. As shown above when the graph is a tree, convergence to the unique exact marginals is guaranteed. However for graphs with loops, BP – in this context often called **loopy belief propagation** as applied to graphs with loops – typically provides a good approximation. The BP can also be corrected systematically by accounting for contributions of loops – see [51] and references therein for more details.

8.1.3 Neural Decoding of Low-Density Parity-Check Codes

In this subsection, we first develop a variational formulation of the belief propagation (BP) approach for decoding Low-Density Parity-Check (LDPC) codes. This leads to an optimization over bit-wise and check-wise beliefs conditioned on noisy channel outputs. Departing from the conventional message-passing scheme used to solve the optimization iteratively, we then describe an alternative approach in which beliefs are approximated using Neural Networks (NNs). The resulting objective, augmented by regularization terms enforcing consistency between bit and check beliefs, is used as a loss function for training the NNs based on samples of (\mathbf{x}, \mathbf{y}) pairs.

Setup: LDPC Code and Parity-Check Matrix

An LDPC code of length n is defined by a binary $m \times n$ parity-check matrix H . A binary vector $\mathbf{x} \in \{0, 1\}^n$ is a valid codeword if it satisfies

$$H\mathbf{x} = \mathbf{0} \pmod{2},$$

where arithmetic is over GF(2). Each row of H corresponds to a parity constraint involving a subset of bits. The sparsity of H enables efficient representation via a bipartite factor graph.

AWGN Channel Model

A codeword \mathbf{x} , mapped to BPSK symbols $\mu_{x_i} \in \{-1, +1\}$, is transmitted over an Additive White Gaussian Noise (AWGN) channel:

$$y_i = \mu_{x_i} + z_i, \quad z_i \sim \mathcal{N}(0, \sigma^2).$$

Typically, $\mu_0 = -1$ and $\mu_1 = +1$.

Posterior Distribution $p(\mathbf{x} | \mathbf{y})$

Given observations \mathbf{y} , the posterior distribution over codewords \mathbf{x} is

$$p(\mathbf{x} | \mathbf{y}) \propto \prod_{c=1}^m \mathbf{1}(H_{c,:}\mathbf{x} = 0 \pmod{2}) \cdot \prod_{i=1}^n \exp\left(-\frac{(y_i - \mu_{x_i})^2}{2\sigma^2}\right).$$

Here, the indicator enforces the parity constraints, while the Gaussian terms correspond to the likelihood.

BP Approximate Posterior

Since exact decoding is infeasible for large n , we approximate the posterior using a factorized variational distribution:

$$q^{(\text{bp})}(\mathbf{x}) = \prod_{c=1}^m q_c(\mathbf{x}_c) \cdot \prod_{i=1}^n [q_i(x_i)]^{\alpha_i}, \quad (8.8)$$

where $q_c^{(\text{bp})}$ denotes marginal beliefs over the bits involved in check c , $q_i^{(\text{bp})}$ denotes marginal beliefs for bit i , and $\alpha_i = 1 - d_i$ accounts for overcounting due to multiple constraints involving bit i .

KL Divergence Objective

The optimal variational distribution minimizes the KL divergence from $q^{(\text{bp})}$ to the true posterior:

$$\text{KL}(q^{(\text{bp})} \| p) = \sum_{\mathbf{x}} q^{(\text{bp})}(\mathbf{x}) \log \frac{q^{(\text{bp})}(\mathbf{x})}{p(\mathbf{x} | \mathbf{y})},$$

subject to local consistency between bit-wise and check-wise beliefs.

NN-Based Belief Approximation

Rather than re-solving this optimization for each channel output \mathbf{y} , we approximate the beliefs using NNs:

$$q_i(x_i | y_i) = \text{NN}_{\theta}^{(\text{bit } i)}(x_i, y_i), \quad q_c(\mathbf{x}_c | \mathbf{y}_c) = \text{NN}_{\theta}^{(\text{check } c)}(\mathbf{x}_c, \mathbf{y}_c),$$

where θ denotes shared trainable parameters. Each bit or check network learns to predict beliefs based on the corresponding channel outputs.

Enforcing Consistency via Penalty

To ensure that the NN-based beliefs are approximately consistent, we introduce soft constraints:

$$\sum_{\mathbf{x}_c \setminus x_i} q_c(\mathbf{x}_c | \mathbf{y}_c) \approx q_i(x_i | y_i), \quad \sum_{\mathbf{x}_c} q_c(\mathbf{x}_c | \mathbf{y}_c) = 1, \quad \sum_{x_i} q_i(x_i | y_i) = 1.$$

These are enforced via an L_1 penalty:

$$\min_{\theta} \left[\text{KL}(q_{\theta} \| p) + \lambda \sum_c \sum_{i \in \text{vars}(c)} \left\| \sum_{\mathbf{x}_c \setminus x_i} q_c(\mathbf{x}_c | \mathbf{y}_c) - q_i(x_i | y_i) \right\|_1 \right].$$

Global Surrogate vs. Factorized NN

Alternatively, one can define a global surrogate $q_{\theta}(\mathbf{x} | \mathbf{y})$ using a single NN that maps (\mathbf{x}, \mathbf{y}) to a probability. This is tractable only for small n due to the exponential size of \mathbf{x} . The BP-style NN decomposition scales better and generalizes to larger blocklengths.

Implementation: NN-BP-LDPC.ipynb

The notebook NN-BP-LDPC.ipynb implements both approaches – global and BP – for a toy LDPC code with 8 codewords. The BP surrogate uses small sub-networks for each bit and check factor: `BitMarginalNN` and `CheckMarginalNN`, respectively.

We conclude this subsection by referencing [53], which proposes a neural extension of belief propagation for LDPC decoding. The method incorporates a Graph Neural Network (GNN) architecture, and is reported to outperform standard BP in decoding accuracy.

8.1.4 Variational Auto-Encoders

Earlier in this section, we explored how inference in graphical models (GMs) can be formulated as an optimization involving probability distributions and KL divergences. We now turn to *Variational Auto-Encoders (VAEs)* – a powerful class of models introduced by Kingma and Welling in [54] that merges the ideas of variational inference from Bayesian statistics with deep neural network architectures. VAEs have become one of the key building blocks in modern generative modeling, alongside approaches such as normalizing flows and diffusion-based models (discussed later). They illustrate how we can efficiently learn latent-variable generative models, even when the posterior distribution is intractable.

Motivation and Setup

Recall that in a latent-variable generative model we often assume:

$$p_\theta(x, z) = p_\theta(z) p_\theta(x | z),$$

where

- $z \in \mathbb{R}^m$ (or a higher-dimensional space) is the latent variable,
- $x \in \mathbb{R}^d$ is the observed data,
- θ are the model parameters describing both the prior $p_\theta(z)$ and the conditional likelihood $p_\theta(x | z)$.

When training, we want to fit θ so that the *marginal likelihood* of the data,

$$p_\theta(x) = \int p_\theta(x, z) dz = \int p_\theta(z) p_\theta(x | z) dz,$$

is maximized. However, the integral $\int p_\theta(z) p_\theta(x | z) dz$ is generally intractable. Moreover, the posterior $p_\theta(z | x)$ is also often impossible to compute in closed form – this is precisely the challenge in complex graphical models that we have been discussing.

A *variational auto-encoder* attacks this problem by introducing an approximate posterior $q_\phi(z | x)$ – also called the *encoder* – and jointly learning both:

1. The generative parameters θ , which define $p_\theta(z) p_\theta(x | z)$.
2. The variational parameters ϕ , which define a learned, tractable distribution $q_\phi(z | x)$ that approximates the true posterior $p_\theta(z | x)$.

The ELBO Objective. From our earlier discussion of variational inference and KL divergence, recall that:

$$\log p_\theta(x) = \mathcal{L}(\theta, \phi; x) + \text{KL}\left(q_\phi(z | x) \parallel p_\theta(z | x)\right),$$

where

$$\mathcal{L}(\theta, \phi; x) = \mathbb{E}_{q_\phi(z|x)} [\log p_\theta(x | z)] - \text{KL}\left(q_\phi(z | x) \parallel p_\theta(z)\right).$$

Because $\text{KL} \geq 0$, we have

$$\mathcal{L}(\theta, \phi; x) \leq \log p_\theta(x).$$

Hence, $\mathcal{L}(\theta, \phi; x)$ is the *Evidence Lower BOund* (ELBO) discussed earlier in Section 8.1.2. Maximizing \mathcal{L} w.r.t. both θ and ϕ increases the log-likelihood $\log p_\theta(x)$ and simultaneously reduces the KL between $q_\phi(z | x)$ and $p_\theta(z | x)$. In other words, ϕ is learned to make the approximate posterior close to the true posterior, and θ is learned so that the model $p_\theta(x | z)$ is good at reconstructing the data from latent codes z .

Neural Network Parameterization. *Encoder (Recognition Model).* In a VAE, $q_\phi(z | x)$ is often chosen to be Gaussian:

$$q_\phi(z | x) = \mathcal{N}(z; \mu_\phi(x), \Sigma_\phi(x)),$$

where $\mu_\phi(x)$ and $\Sigma_\phi(x)$ (or a diagonal approximation) are given by *Neural Networks* (NN) of parameters ϕ . These networks transform the observed data x into a latent code distribution. Because the dimension of z is typically much smaller than x (e.g., a 2D or 64D latent space for images), we get a compressed representation of the data in z .

Decoder (Generative Model): Simultaneously, $p_\theta(x | z)$ is modeled by another NN (decoder) that takes a latent variable z as input and outputs the parameters of the distribution of x . For example:

- If x represents real-valued data (e.g. pixel intensities in $[0, 1]$), one might let $p_\theta(x | z) = \mathcal{N}(\mu_\theta(z), \Sigma_\theta(z))$.
- Alternatively, for grayscale or color images in $[0, 1]$, a Bernoulli or Beta distribution can be used to reflect the bounded pixel range.

These *encoder* and *decoder* NNs are jointly optimized to maximize the ELBO. In practice, we split the ELBO into:

$$\max_{\theta, \phi} \mathcal{L}(\theta, \phi) = \underbrace{\mathbb{E}_{q_\phi(z|x)} [\log p_\theta(x | z)]}_{\text{(reconstruction term)}} - \underbrace{\text{KL}\left(q_\phi(z | x) \parallel p_\theta(z)\right)}_{\text{(regularization term)}},$$

where:

- *Reconstruction term* encourages the decoder network $p_\theta(x | z)$ to produce samples close to the observed data x , given latent code z .
- *KL regularization term* encourages the approximate posterior to stay close to the prior $p_\theta(z)$. A common choice is a standard Gaussian prior $p_\theta(z) = \mathcal{N}(0, I)$.

Example 8.1.3. 2D Ring Let us consider a **2D toy dataset** consisting of points $x = (x_1, x_2)$ shaped in a ring (e.g., radius 2). We want to learn a 1D latent variable $z \in \mathbb{R}$ that can produce these data.

1. **Data Generation (Synthetic):** Sample angles θ from $[0, 2\pi)$ with some noise, then let

$$x_1 = 2 \cos(\theta) + \varepsilon_1, \quad x_2 = 2 \sin(\theta) + \varepsilon_2.$$

In code, we might create 5000 data points like this and shuffle.

2. **Encoder** $q_\phi(z \mid x)$: A small fully connected net with layers mapping $(x_1, x_2) \rightarrow (\mu_\phi(x), \sigma_\phi(x))$. We treat the approximate posterior as $\mathcal{N}(\mu_\phi(x), \sigma_\phi(x)^2)$.
3. **Decoder** $p_\theta(x \mid z)$: Another small MLP that takes z and outputs $\hat{x} \approx (x_1, x_2)$. We interpret \hat{x} as the mean of a Gaussian likelihood.
4. **Training:**

- We sample a mini-batch $\{x^{(i)}\}$.
- For each $x^{(i)}$, the encoder outputs $\mu_\phi(x^{(i)})$, $\sigma_\phi(x^{(i)})$.
- We sample $z^{(i)} \sim \mathcal{N}(\mu_\phi(x^{(i)}), \sigma_\phi(x^{(i)})^2)$ using the reparameterization trick:

$$z^{(i)} = \mu_\phi(x^{(i)}) + \sigma_\phi(x^{(i)}) \epsilon, \quad \epsilon \sim \mathcal{N}(0, I).$$

- The decoder then predicts $\hat{x}^{(i)} = D_\theta(z^{(i)})$.
- We compute the reconstruction term $\log p_\theta(x^{(i)} \mid z^{(i)})$ and the KL term $\text{KL}\left(q_\phi(z \mid x^{(i)}) \parallel p_\theta(z)\right)$.
- We backpropagate the negative of their sum (which is $-\text{ELBO}$).

After training, we can sample from the learned generative model by drawing latent codes $z \sim p_\theta(z)$ (often $\mathcal{N}(0, I)$) and pushing them through the decoder to get new \hat{x} . On a ring-like dataset, a well-trained VAE can produce samples that lie approximately on the ring.

Exercise 8.1.3. 1D-Latent VAE for a Ring Dataset

- **Data Creation.** Generate $\{x^{(i)}\}_{i=1}^N$ in \mathbb{R}^2 by sampling θ uniformly in $[0, 2\pi)$:

$$x_1^{(i)} = R \cos(\theta_i) + \varepsilon_1, \quad x_2^{(i)} = R \sin(\theta_i) + \varepsilon_2,$$

with $R = 2$ and $\varepsilon_1, \varepsilon_2$ small Gaussian noise.

- **Encoder.** Construct $q_\phi(z \mid x)$ with a two-layer Multi-Layer-Perceptron (MLP). The output is $\mu_\phi(x)$ and $\log \sigma_\phi(x)$.
- **Decoder.** Construct $p_\theta(x \mid z)$ with another two-layer MLP. Output the mean of $\mathcal{N}(\hat{x}, \mathbf{I})$.
- **Training.** Write (or adapt) a short script:

1. Implement the reparameterization trick: $z = \mu_\phi(x) + \exp(\log \sigma_\phi(x)) \cdot \epsilon$, $\epsilon \sim \mathcal{N}(0, 1)$.

2. Compute $\log p_\theta(x | z)$ using an L2 penalty $\|x - \hat{x}\|^2$ for reconstruction.
 3. Compute the $\text{KL}(q_\phi(z | x) || \mathcal{N}(0, 1))$ term in closed form (since both are Gaussians).
 4. Minimize the negative ELBO (i.e. sum of reconstruction loss plus KL) over mini-batches.
- **Visualization.** Plot final reconstructions \hat{x} for random samples from the ring. Also plot new points generated by decoding from random latent codes $z \sim \mathcal{N}(0, 1)$. Discuss how well the learned model captures the geometry of the ring and how the 1D latent variable “walks” around the ring.

Connection to Bayesian Inference and Information Theory:

1. *Bayesian Perspective.* The approximate posterior $q_\phi(z | x)$ stands in for the true posterior $p_\theta(z | x)$. Through the reparameterization trick and gradient-based learning, the VAE simultaneously fits a generative model $p_\theta(x | z)$ and a powerful inference network q_ϕ . This ties closely to the Bayesian mindset introduced earlier: we have a prior $p_\theta(z)$, a likelihood $p_\theta(x | z)$, and an (intractable) posterior replaced by a learned approximation.
2. *KL Divergence.* VAEs place KL divergences front-and-center: they impose a KL penalty $\text{KL}(q_\phi(z | x) || p_\theta(z))$ to keep the latent encoding “close” to the prior and to avoid degenerate solutions.
3. *ELBO and Mutual Information.* From the standpoint of mutual information, the ELBO can be interpreted as controlling the flow of information about x that gets “bottlenecked” through z . By adjusting the KL term’s weighting (often done by a β -VAE), one can trade off the reconstruction fidelity vs. how “compressive” the latent codes become.

Bridge to Next Topics: Learning, Diffusion, and Beyond. Variational auto-encoders have profoundly influenced *how* we learn generative models of data. In what follows we will explore:

- *Learning (next section):* More advanced aspects of maximizing likelihood and dealing with partial observability, including Expectation-Maximization and contrastive divergence in energy-based models.
- *Synthesis (next, final chapter):* How the ideas of learning latent variables extend to modern diffusion-based models and Schrödinger bridge formulations in statistical mechanics. There, we will see how VAEs can be combined with diffusion processes or used as building blocks in advanced architectures.

VAEs thus stand at the intersection of Bayesian inference, NN expressivity, and the variational ideas we have been studying. They demonstrate that a carefully chosen combination of KL-based regularization and neural approximation can yield a highly flexible but still principled generative model, paving the way to deeper theories of generative AI.

PyTorch Code Snippet for a 1D-Latent VAE on a Ring Dataset

A minimal PyTorch script (`ring_vae.ipynb`) is available. It illustrates data creation, building the encoder/decoder, and training via the ELBO objective. This snippet can be adapted to match the exercise instructions above.

Remarks on the Snippet.

- We treat the decoder output as the *mean* of a Gaussian distribution (with unit covariance), and use an L2 (MSE) loss for reconstruction. In practice, one could model the covariance as well.
- The KL term for $q_\phi(z \mid x) = \mathcal{N}(\mu, \sigma^2)$ vs. the standard normal $p_\theta(z) = \mathcal{N}(0, I)$ is computed in closed form:

$$\text{KL} = \frac{1}{2} \sum_{i=1}^d (\mu_i^2 + \sigma_i^2 - \log \sigma_i^2 - 1).$$

- Although here we do a single update per loop, in a real application we might iterate over multiple batches per epoch, shuffle data, etc.
- After training, one can visualize:
 1. The learned reconstructions by passing actual x through the encoder and decoding the resulting z .
 2. Random new samples from $z \sim \mathcal{N}(0, I)$, decoded to x space.

8.2 Learning

This section delves into how one can infer or learn an underlying Graphical Model (GM) directly from data. We focus on methods relevant to *Energy-Based Models (EBMs)*, touching on both conceptual underpinnings and efficient algorithms. The exposition closely mirrors discussions in Sections 10.3 of [50] and 9.2 of [51], as well as the notes introduced in earlier sections of this text. The goal is to reveal how statistical learning, i.e. constructing the GM from data, is conceptually intertwined with inference (partition function, marginals, sampling) and can inherit its computational difficulties. We outline both the core theory and the common approximations crucial for modern AI.

8.2.1 Likelihood

On the Sufficiency of Empirical Moments in the Exponential Family

Building on the general framework of likelihood-based learning, we now explore the principle of *sufficiency of empirical moments* within the exponential family of distributions. While the Ising model over binary variables provides a concrete and widely studied case, the underlying insights generalize seamlessly to broader settings—including models over categorical, count, and continuous variables.

Exponential-Family Formulation: A family of probability distributions over variables $x = (x_1, \dots, x_N) \in \mathcal{X}^N$, where \mathcal{X} may be discrete or continuous, is said to belong to the exponential family if the probability density (or mass) function can be written in the canonical form:

$$P_\theta(x) = h(x) \exp(\theta^\top T(x) - A(\theta)), \quad (8.9)$$

where:

- $\theta \in \mathbb{R}^d$ is the vector of natural (canonical) parameters,
- $T(x) \in \mathbb{R}^d$ is the vector of sufficient statistics,
- $h(x)$ is the base measure (e.g., Lebesgue measure density or a discrete counting term),
- $A(\theta) = \log \int h(x) \exp(\theta^\top T(x)) dx = \log Z(\theta)$ is the log-partition function ensuring normalization.

This framework encompasses a wide variety of common distributions – e.g., Bernoulli, categorical, Poisson, exponential, Gaussian, and their multivariate extensions.

Sufficiency of Empirical Moments: Suppose we are given an i.i.d. dataset $\{x^{(\ell)}\}_{\ell=1}^M$, each sample drawn from a distribution $P_\theta(x)$ in the form (8.9). The log-likelihood of the parameter θ given the dataset is:

$$\log \mathcal{L}(\theta) = \sum_{\ell=1}^M \log P_\theta(x^{(\ell)}) = \sum_{\ell=1}^M (\theta^\top T(x^{(\ell)}) - A(\theta) + \log h(x^{(\ell)})). \quad (8.10)$$

Since $h(x^{(\ell)})$ is independent of θ , maximizing the likelihood reduces to:

$$\max_{\theta} \log \mathcal{L}(\theta) = \theta^\top \left(\sum_{\ell=1}^M T(x^{(\ell)}) \right) - M A(\theta). \quad (8.11)$$

This form makes clear that the entire dependence on the dataset enters through the aggregated sufficient statistics:

$$\hat{T}_M = \frac{1}{M} \sum_{\ell=1}^M T(x^{(\ell)}). \quad (8.12)$$

These empirical moments \hat{T}_M are thus sufficient for maximum likelihood estimation (MLE): once they are computed, the raw dataset $\{x^{(\ell)}\}$ is no longer needed for likelihood-based inference or optimization.

Example 8.2.1. • In the Ising model (binary $x_i \in \{0, 1\}$ or $\{\pm 1\}$), $T(x)$ includes one- and two-variable terms such as x_a and $x_a x_b$.

- In a multivariate Gaussian, the sufficient statistics are x and xx^\top .
- In categorical models, $T(x)$ typically encodes indicator functions of variable configurations.

In all cases, the log-likelihood depends on θ only through inner products with \hat{T}_M and the log-partition function $A(\theta)$.

Implications for Learning and Inference: This structure offers major practical advantages:

- **Data compression:** Storing \widehat{T}_M reduces memory cost and enables repeated optimization (e.g., hyperparameter tuning or initialization restarts) without reloading raw data.
- **Gradient computation:** The gradient of the log-likelihood is:

$$\nabla_{\theta} \log \mathcal{L}(\theta) = M \left(\widehat{T}_M - \mathbb{E}_{\theta}[T(x)] \right),$$

which highlights the moment-matching principle at the heart of exponential-family learning.

Partition Function Challenges: Despite the elegance of the exponential family, its practical use in high dimensions is hindered by the difficulty of computing the partition function $Z(\theta) = \exp(A(\theta))$. This function – and its gradient – requires integrating or summing over all possible configurations $x \in \mathcal{X}^N$, which is exponential in N for most interesting models. Thus:

- Maximum likelihood learning is often intractable in general due to the cost of computing $A(\theta)$ and $\nabla_{\theta}A(\theta)$.
- This is the same obstacle encountered in probabilistic inference—e.g., when computing marginals or conditionals under $P_{\theta}(x)$.

Contrast with Traditional AI Models: This computational barrier distinguishes exponential-family models with unnormalized densities from many classical parametric models in machine learning:

- For example, logistic regression or NNs parameterize the conditional $P(y | x)$ directly in normalized form. Training them does not require computing a partition function.
- In contrast, Energy-Based Models (EBMs) that define only unnormalized log-densities (e.g., $E_{\theta}(x) = -\theta^{\top}T(x)$) must learn while handling normalization implicitly – via surrogates or approximations.

Approximations and Algorithmic Complexity: To sidestep exact computation of $A(\theta)$, practitioners rely on:

- Sampling methods (e.g., MCMC) to estimate expectations under $P_{\theta}(x)$,
- Variational methods or mean-field approximations to upper-bound $A(\theta)$,
- Pseudo-likelihood or contrastive divergence to avoid partition functions during learning,
- Score-matching or noise-contrastive estimation, which minimize alternative objectives that do not require $Z(\theta)$.

Pointers to Generative AI (more in the next chapter): While exponential-family models were originally developed with an emphasis on statistical sufficiency, interpretability, and likelihood-based learning, they also underpin many ideas central to modern generative AI. In particular, models that define a probability distribution through an unnormalized energy function—often referred to as *Energy-Based Models* (EBMs)—form a conceptual bridge between classical statistics and contemporary generative frameworks.

- **Score-based models** aim to learn the gradient of the log-density function, known as the *score function*, $\nabla_x \log P(x)$. For energy-based models of the form $P_\theta(x) \propto \exp(-E_\theta(x))$, this score becomes $-\nabla_x E_\theta(x)$. This formulation avoids explicit computation of the partition function, aligning well with score-matching-based training.
- The term *score function* originates in classical statistics (see e.g. [55] and references therein), where it denotes the gradient of the log-likelihood with respect to parameters or inputs. In generative modeling, it refers to the gradient of the log-probability with respect to the input variables x .
- **Diffusion models** (e.g., [56], [57]) generate data by reversing a stochastic corruption process – such as a Gaussian noise diffusion – using a learned approximation of the score function at various noise levels. This reverse-time evolution mimics Langevin sampling from an unnormalized energy model, grounding diffusion models in the same energy-based intuition.
- Thus, despite architectural and algorithmic advances, many modern generative AI models rest on energy-based or unnormalized probabilistic foundations. These approaches often bypass the need to compute the partition function through dynamic, stochastic, or iterative sampling.

In summary, the exponential family provides a principled and flexible framework where sufficient statistics – typically low-dimensional empirical moments – encapsulate all necessary data information for likelihood-based learning. This elegant statistical property enables efficient data summarization and reuse. However, practical deployment – especially for high-dimensional models – faces serious computational barriers due to the intractability of the log-partition function. Approximate inference and sampling techniques remain essential for connecting the theoretical foundations of exponential-family models to scalable learning and generative modeling.

Example 8.2.2 (Score Matching for Energy-Based Models). Consider a synthetic 2D dataset composed of samples from a mixture of Gaussians. We wish to fit an unnormalized energy-based model of the form

$$P_\theta(x) \propto \exp(-E_\theta(x)),$$

where $E_\theta(x)$ is a shallow neural network parameterizing the energy landscape.

To avoid computing the intractable partition function, we train the model using score matching [55], which minimizes the expected squared difference between the model's score function and the data's true score:

$$\mathbb{E}_{x \sim \text{data}} \left[\frac{1}{2} \|\nabla_x E_\theta(x)\|^2 + \Delta_x E_\theta(x) \right].$$

This objective is derived by expanding the Fisher divergence $\|\nabla_x \log P_\theta(x) - \nabla_x \log P_{\text{data}}(x)\|^2$, and it bypasses the partition function entirely. Notably, the gradient $-\nabla_x E_\theta(x)$ approximates the direction in which data samples concentrate under the modeled distribution.

This form of learning is deeply connected to the Langevin equation (see Section 7.3.3), which describes the dynamics of sampling from an unnormalized distribution via stochastic differential equations. Specifically, Langevin dynamics simulate trajectories governed by the score function:

$$dx_t = -\nabla_x E_\theta(x_t) dt + \sqrt{2} dW_t,$$

providing a mechanism for sample generation directly from $P_\theta(x)$ using the learned energy function.

The Jupyter/PyTorch notebook `ScoreMatchingEnergy.ipynb` provides a working implementation that fits such a model to a toy 2D Gaussian mixture.

Exercise 8.2.1 (Score-Based Generation and Denoising Extensions). This exercise extends the score-matching example into the realm of sampling and generative modeling. Your tasks span both classical Langevin-based sampling and modern denoising-score learning.

- (A) **Sampling via Langevin Dynamics.** Implement Langevin dynamics to sample from the learned energy model $E_\theta(x)$. Start from isotropic Gaussian noise and use:

$$x_{k+1} = x_k - \eta \nabla_x E_\theta(x_k) + \sqrt{2\eta} \cdot \xi_k, \quad \xi_k \sim \mathcal{N}(0, I).$$

Visualize the resulting samples. Compare the generated distribution to the original data using scatter plots and kernel density estimates. How does sample quality vary with step size and number of iterations?

- (B) **Denoising Score Matching (DSM).** Train a separate model $s_\phi(\tilde{x})$ to learn the score of noisy data using the DSM objective:

$$\mathcal{L}_{\text{DSM}} = \mathbb{E}_{x, \varepsilon} \left[\left\| s_\phi(x + \sigma \varepsilon) + \frac{\varepsilon}{\sigma} \right\|^2 \right],$$

where $\varepsilon \sim \mathcal{N}(0, I)$, and σ is a fixed noise level.

Use the same Gaussian mixture dataset, and explore how training behaves as σ is varied.

- (C) **Generative Sampling with Denoising Scores.** Use the trained denoising score model to implement a simple stochastic sampling process (e.g., annealed Langevin dynamics). For example:

$$x_{k+1} = x_k + \epsilon_k \cdot s_\phi(x_k) + \sqrt{2\epsilon_k} \cdot \xi_k,$$

where ϵ_k is a step schedule and $\xi_k \sim \mathcal{N}(0, I)$.

Generate samples and compare them with both:

- Samples from the original mixture of Gaussians.

- Samples from the Langevin dynamics in part (A).

(D) **Compare and Reflect.** Discuss:

- Differences in sample quality, mode coverage, and convergence behavior.
- Challenges in training energy vs. score-based models.
- Which method is more stable or scalable in higher dimensions?

Optional: Quantify distributional similarity using kernel two-sample tests or compute metrics like Maximum Mean Discrepancy (MMD).

8.2.2 Local Methods: Pseudo-Log-Likelihood and Interaction Screening

When learning the structure or parameters of large-scale graphical models, direct likelihood maximization quickly becomes computationally intractable due to the need to evaluate the global partition function or its gradient. In this subsection, we explore local approximation techniques that offer efficient and also asymptotically exact alternatives by focusing on conditional or neighborhood-based objectives. These include the Pseudo-Log-Likelihood (PLL) and Interaction Screening (IS) approaches – see [58, 59, 51] and references therein – both of which avoid costly global computations while retaining theoretical soundness in many practical settings.

Motivation for Approximate Learning: For Graphical Models (GM) over N variables, computing or differentiating the log-likelihood $\ln P(\text{data} \mid \theta)$ requires evaluating the partition function $Z(\theta)$ or its gradient. This step is computationally prohibitive for most nontrivial models (e.g., Ising models), where summing over 2^N configurations is infeasible. Approximate local objectives, which sidestep global normalization, offer scalable alternatives.

Pseudo-Log-Likelihood: Instead of maximizing the joint log-likelihood, the pseudo-log-likelihood approach maximizes the sum of local conditional log-probabilities:

$$\mathcal{L}_{\text{PLL}}(\theta) = \sum_{a \in \mathcal{V}} \sum_{s=1}^S \log P_\theta(x_a^{(s)} \mid x_{\setminus a}^{(s)}), \quad (8.13)$$

where $x_{\setminus a}^{(s)}$ denotes the observed values at all nodes which may be neighbors of the node $a \in \mathcal{V}$. For exponential family models the conditional distributions $P(x_a \mid x_{\setminus a})$ have a closed-form representation. For example, in the case of the Ising model

$$P(x_a = 1 \mid x_{\setminus a}) = \frac{\exp(\theta_a + \sum_{b \in \mathcal{N}(a)} \theta_{ab} x_b)}{1 + \exp(\theta_a + \sum_{b \in \setminus a} \theta_{ab} x_b)}.$$

Thus, maximizing \mathcal{L}_{PLL} reduces to solving $|\mathcal{V}|$ independent logistic regression problems, one per node, where each node's value is regressed against its neighbors.

Interaction Screening: This class of methods assumes that local structure can be inferred independently by inspecting correlations between a node and its neighborhood. In the case of the Ising model, this amounts to estimating local statistics—magnetizations and pairwise correlations—and solving node-wise regression problems to infer edges.

For instance, one can minimize a surrogate objective for each node a :

$$\min_{\theta_a, \theta_{ab}} \mathbb{E}_{\text{empirical}} \left[\left(x_a - \sigma \left(\theta_a + \sum_{b \in \setminus a} \theta_{ab} x_b \right) \right)^2 \right] + \lambda \sum_b |\theta_{ab}|,$$

where $\sigma(\cdot)$ is the sigmoid function and λ promotes sparsity. Efficient implementations via coordinate descent or greedy methods make this strategy well-suited for high-dimensional graphs.

Theoretical Guarantees and Comparisons: Under certain conditions (e.g., incoherence, sufficient data), both pseudo-likelihood and interaction screening methods are consistent estimators of the true underlying graph structure. In particular, [58, 59, 51] discusses a variety of theoretical regimes under which these estimators are provably accurate, even in regimes where global MLE is computationally infeasible. Moreover, these methods can be shown to exhibit sparsistency (correct recovery of zero vs. nonzero parameters) and fast convergence rates.

Example 8.2.3 (Node-Wise Logistic Regression on Synthetic Ising Data.). We consider a synthetic example mimicking a local Ising-like interaction. Each sample consists of 3 binary variables (neighbors) $x_b \in \{-1, 1\}$ and a target variable x_a generated deterministically via

$$x_a = \text{sign}(1.5x_1 - 2x_2 + 2x_3).$$

We then convert x_a into a binary label $y_a = 1$ if $x_a > 0$ and 0 otherwise. The logistic regression model is then trained to recover this mapping:

$$\min_{\theta_a, \theta_{ab}} -\frac{1}{M} \sum_{s=1}^S \log \sigma \left(y_a^{(s)} \left(\theta_a + \sum_{b=1}^3 \theta_{ab} x_b^{(s)} \right) \right).$$

The PyTorch notebook `Node-wise-logistic.ipynb` implements this synthetic data generation, regression, and training loop, showing convergence to the ground-truth weights and illustrating the interpretability of pseudo-likelihood estimation when the neighborhood structure is known.

Exercise 8.2.2 (Scalable Structure Learning with Local Objectives). Simulate an Ising model over $N = 50$ binary variables arranged in a 2D grid or random graph topology. Generate $M = 10^4$ i.i.d. samples and perform structure learning via both pseudo-likelihood and interaction screening.

- (a) Extend the PyTorch snippet in the example to estimate all N local conditional distributions using independent logistic regressions (one per node).

- (b) Recover the edge structure by thresholding the learned weights: $|\theta_{ab}| > \epsilon$ for some $\epsilon > 0$.
- (c) Compare the learned graph with the true one using precision, recall, and F1 score.
- (d) Analyze how recovery quality depends on number of samples M and regularization λ .
- (e) (Advanced) Implement interaction screening using node-wise ℓ_1 -regularized squared loss instead of logistic. Compare performance.
- (f) (Open-ended) Can you reduce the number of samples needed to reach 95% edge recovery? Try: (i) smarter initialization, (ii) pruning candidate edges before training, (iii) multi-node training using shared features.

In summary – local approximation methods such as pseudo-log-likelihood and interaction screening provide practical alternatives to full likelihood-based learning in graphical models. By exploiting conditional independence and locality, they allow for scalable structure estimation even when global partition functions are intractable. Their relevance is especially pronounced in high-dimensional learning and foundational to modern developments in energy-based modeling.

8.2.3 Restricted Boltzmann Machine

Structure and Motivation: A Restricted Boltzmann Machine (RBM) is an undirected, bipartite graphical model that defines a joint energy-based probability distribution over two layers of binary random variables: **visible units** $v \in \{0, 1\}^n$ and **hidden units** $h \in \{0, 1\}^m$. The absence of intra-layer connections (no v - v or h - h edges) makes inference and sampling more tractable. Obviously the RBM is a particular case of the Ising model, introduced earlier in Section 7.5, over a bi-partite graph.

We recall that the energy function of the RBM is defined as:

$$E_\theta(v, h) = -v^\top Wh - b^\top v - c^\top h, \quad \theta = (W, b, c), \quad (8.14)$$

where $W \in \mathbb{R}^{n \times m}$ is the weight matrix, and b, c are the bias vectors for the visible (observed) and hidden units, respectively.

The joint distribution of v and h is given by the Gibbs distribution:

$$P_\theta(v, h) = \frac{1}{Z_\theta} \exp(-E_\theta(v, h)), \quad \text{where } Z_\theta = \sum_{v, h} \exp(-E_\theta(v, h)). \quad (8.15)$$

The RBM thus defines a marginal distribution over the visible variables:

$$P_\theta(v) = \frac{1}{Z_\theta} \sum_h \exp(-E_\theta(v, h)), \quad (8.16)$$

which makes it a powerful generative model despite its simple structure.

Learning via Contrastive Divergence (CD): Learning the parameters $\theta = (W, b, c)$ is typically performed by (approximate) gradient ascent on the log-likelihood of the data:

$$\nabla_\theta \log P(v) = \mathbb{E}_{P_\theta(h|v)}[\nabla_\theta E_\theta(v, h)] - \mathbb{E}_{P_\theta(v, h)}[\nabla_\theta E_\theta(v, h)], \quad (8.17)$$

where the second term requires averaging over the model distribution, which is intractable for large systems.

Contrastive Divergence (CD- k) for an RBM. Hinton's *Contrastive Divergence* algorithm [60] trades an intractable expectation over the model distribution for a short, data-anchored Gibbs sampling/walk (see Section 7.5). For an RBM with energy (8.14) CD proceeds as follows (one update per training vector):

1. **Clamp data.** Take a training example $v^{(0)}$ ("visible layer").
2. **Sample hidden units.** Draw $h^{(0)} \sim P_\theta(h | v^{(0)})$, where each hidden unit is independent: $P(h_j = 1 | v) = \sigma((W^\top v + c)_j)$.
3. **k alternating Gibbs steps.** For $t = 0, \dots, k - 1$ do

$$v^{(t+1)} \sim P_\theta(v | h^{(t)}), \quad h^{(t+1)} \sim P_\theta(h | v^{(t+1)}).$$

Each visible unit is sampled via $P(v_i = 1 | h) = \sigma((Wh + b)_i)$. After k full *visible* \rightarrow *hidden* \rightarrow *visible* cycles we obtain the *negative-phase* pair $(v^{(k)}, h^{(k)})$.

4. **Contrastive-divergence gradient.** Replace the intractable model expectation in the true log-likelihood gradient by the empirical average over the single "negative-phase" sample:

$$\nabla_\theta \mathcal{L}_{\text{CD-}k} \approx \underbrace{\mathbb{E}_{P_\theta(h|v^{(0)})} [\nabla_\theta E_\theta(v^{(0)}, h)]}_{\text{positive phase}} - \underbrace{\mathbb{E}_{P_\theta(h|v^{(k)})} [\nabla_\theta E_\theta(v^{(k)}, h)]}_{\text{negative phase}}.$$

Concretely, $\Delta W \propto v^{(0)}h^{(0)\top} - v^{(k)}h^{(k)\top}$, and analogous formulas hold for b and c .

Although this gradient is *biased* for small k (CD-1 is common), it works well in practice because starting the chain at real data keeps $v^{(k)}$ close to $v^{(0)}$ when the model is near convergence; the resulting "contrast" between the two expectations still points in a useful ascent direction while costing only two conditional passes through the network.

Expectation Maximization (EM) Perspective: The RBM's hidden layer introduces latent variables h , suggesting a connection with the Expectation-Maximization (EM) framework. EM proceeds by alternating:

- **E-step:** Estimate the expected sufficient statistics of the hidden units conditioned on visible data;
- **M-step:** Maximize the complete-data log-likelihood using these expectations.

However, unlike Gaussian mixture models or Hidden Markov Models, the "M-step" here involves computing the model expectation over $P(v, h)$, which is again intractable — hence the resort to MCMC approximations as in CD.

Challenges and Modern Role: RBMs played a foundational role in the early days of deep learning:

- They enabled unsupervised pre-training of Deep Belief Networks (DBNs) and Deep Boltzmann Machines (DBMs);
- Their bipartite structure facilitates block Gibbs sampling and learning via CD;

- They form a conceptual bridge between Graphical Models (GM) and modern generative networks.

Today, RBMs have largely been supplanted by more flexible generative architectures such as VAEs and diffusion models. However, RBMs remain valuable as pedagogical tools and as building blocks in hybrid models where interpretable latent structure is beneficial.

Example 8.2.4 (Small RBM with Binary Units). Consider an RBM with $n = 2$ visible and $m = 1$ hidden units, all binary. Let $v = (v_1, v_2) \in \{0, 1\}^2$, and $h \in \{0, 1\}$. Let the parameters be:

$$W = \begin{bmatrix} 1 \\ -1 \end{bmatrix}, \quad b = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad c = 0.$$

Then the energy is:

$$E(v, h) = -v_1 \cdot 1 \cdot h - v_2 \cdot (-1) \cdot h = -h(v_1 - v_2).$$

The conditional distributions are:

$$P(h = 1|v) = \sigma(v_1 - v_2), \quad P(v_i = 1|h) = \sigma(W_{i1}h),$$

where $\sigma(x) = 1/(1 + e^{-x})$ is the sigmoid function.

This example can be computed exhaustively by enumerating the $2^3 = 8$ possible (v, h) configurations.

Exercise 8.2.3 (Manual and Visual Exploration of Tiny RBMs). This exercise explores two small-scale Restricted Boltzmann Machines (RBMs) — one with fully binary units, and one with Gaussian visible units and binary hidden units. These models allow for exact enumeration, manual computation, and visual interpretation of key quantities.

Part I: Binary-Binary RBM

Consider the RBM in Example 8.2.4, where $v = (v_1, v_2) \in \{0, 1\}^2$, $h \in \{0, 1\}$, and parameters are:

$$W = \begin{bmatrix} 1 \\ -1 \end{bmatrix}, \quad b = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad c = 0.$$

1. Compute the energy $E(v, h)$ and the unnormalized probability $\exp(-E)$ for all $2^3 = 8$ possible (v, h) configurations.
2. Normalize to compute the joint probability $P(v, h)$.
3. Compute the marginal distribution $P(v)$ and compare the values to those obtained via explicit summation.
4. Compute the conditional probability $P(h = 1|v)$ and verify that it matches $\sigma(v_1 - v_2)$.
5. Simulate one full step of block Gibbs sampling starting from $v = (1, 0)$. That is:
 - Sample $h \sim P(h|v)$;
 - Sample $v' \sim P(v|h)$.

What is the typical direction of transition? Interpret the result in terms of energy preference.

Part II: Gaussian-Bernoulli RBM

Now consider a Gaussian-Bernoulli RBM with:

$$v \in \mathbb{R}^2, \quad h \in \{0, 1\}, \quad W = \begin{bmatrix} 1 \\ -1 \end{bmatrix}, \quad b = \mathbf{0}, \quad c = 0, \quad \sigma = (1, 1).$$

The energy becomes:

$$E(v, h) = \frac{1}{2} \sum_{i=1}^2 (v_i - b_i)^2 - h(v_1 - v_2).$$

1. Derive the conditional distributions:

- $P(h = 1|v) = \sigma((v_1 - v_2));$
- $P(v|h)$ is a product of Gaussians. Write explicit expressions for the means and variances.

2. Plot the function $P(h = 1|v)$ over a grid of $v \in [-4, 4]^2$ and interpret its symmetry and decision boundary.

3. Simulate one full Gibbs sampling step starting from a visible configuration $v_0 = (2, -2)$. Sample:

- $h \sim P(h|v_0);$
- $v_1 \sim P(v|h).$

Comment on the resulting sample and how h partitions the visible space.

This exercise illustrates how RBMs — both binary and hybrid — model joint distributions and how learning proceeds through alternating conditionals. You are encouraged to experiment numerically using the accompanying Jupyter notebook: `rbm+gaussian-rbm-toy.ipynb`.

Beyond RBM: Deep Belief Networks (DBNs): RBMs are composable — multiple RBMs can be stacked to form a **Deep Belief Network (DBN)**. The key insight is that the hidden layer of one RBM can serve as the visible layer of the next. Training proceeds greedily:

1. Train the first RBM using CD to model the data distribution $P_1(v)$.
2. Sample $h_1 \sim P(h_1|v)$ to create synthetic data for layer 2.
3. Train a second RBM to model $P_2(h_1)$, and repeat for deeper layers.

This unsupervised pre-training initializes a deep generative model that can be fine-tuned with backpropagation. DBNs were popular in early deep learning and were used for digit recognition, speech models, and collaborative filtering.

Gaussian-Bernoulli RBMs: Hybrid Units. While the original RBM is binary-binary, it can be extended to model real-valued data (e.g., pixel intensities) using Gaussian-Bernoulli units:

- **Visible units** $v \in \mathbb{R}^n$ are modeled as conditionally Gaussian.
- **Hidden units** $h \in \{0, 1\}^m$ remain binary.

The energy function becomes:

$$E(v, h) = \sum_i \frac{(v_i - b_i)^2}{2\sigma_i^2} - \sum_j c_j h_j - \sum_{i,j} \frac{v_i}{\sigma_i} W_{ij} h_j,$$

where σ_i is the standard deviation for visible unit v_i .

This formulation is especially useful for image and speech data. Hybrid RBMs serve as input layers for DBNs or as standalone generative models.

Exercise 8.2.4 (Contrastive Divergence for Gaussian RBM). *Given a Gaussian-Bernoulli RBM:*

1. Derive the conditional $P(h_j = 1 | v)$.
2. Derive the conditional $P(v_i | h)$.
3. Implement one step of Contrastive Divergence and simulate data from a real-valued image patch.

Use a small dataset (e.g., 8×8 grayscale images) to illustrate training.

8.2.4 From Graphical Models to Graph Neural Networks

Graphical Models (GMs) and Graph Neural Networks (GNNs) both rely on graph-based structures to represent dependencies, but they arise from different traditions. GMs are probabilistic models capturing structured distributions and enabling inference over latent variables. GNNs are differentiable architectures used in deep learning to process data structured as graphs, focusing primarily on representation learning and prediction. In this section, we bridge these two paradigms, emphasizing how GNNs can be seen as learned message-passing schemes inspired by inference in GMs.

Physics-Informed Machine Learning. In many domains, the structure of the data is not arbitrary but arises from physical or relational constraints, often represented naturally as graphs: molecules, sensor networks, power grids, social networks, and more. In this context, specifying a graph is a form of domain knowledge – an encoding of spatial, temporal, or functional adjacency. Graphical models explicitly encode this via energy terms or conditional dependencies, while GNNs use it to define the flow of information during learning.

Graphical Convolutions and Embeddings. The architecture of a GNN is constructed around the principle of *message passing*, where the features of each node are updated by aggregating transformed features from its neighbors. This operation, at a high level, mirrors variational inference or belief propagation in GMs, where nodes iteratively share information to compute marginals or conditionals, as described in Section 8.1.2.

Let $G = (V, E)$ be an undirected graph with $|V| = n$ nodes and edges $(i, j) \in E$. A typical GNN update layer takes the form:

$$h_i^{(t+1)} = \sigma \left(W_0 h_i^{(t)} + \sum_{j \in \mathcal{N}(i)} W_1 h_j^{(t)} \right),$$

where $h_i^{(t)} \in \mathbb{R}^d$ is the embedding of node i at layer t , $\mathcal{N}(i)$ is the neighborhood of node i , and σ is a nonlinear activation function. This can be interpreted as a learned approximation to a local marginal update, similar in spirit to how the mean-field Eqs. (8.5) are solved iteratively in the variational inference framework, or how message passing described by Eq. (8.7) works in the case of the belief propagation algorithm.

Ising Model-Inspired GNN. Suppose we have a binary node label $y_i \in \{-1, +1\}$ defined over a graph, with features $\mathbf{x}_i \in \mathbb{R}^d$ and pairwise energy:

$$E(\mathbf{y}) = - \sum_{(i,j) \in E} J_{ij} y_i y_j - \sum_i h_i y_i,$$

as in the Ising model. A GNN may be trained to predict y_i by learning embeddings h_i that mimic the effect of the energy minimization. One can construct a GNN layer where

$$h_i^{(t+1)} = \sigma \left(W_1 h_i^{(t)} + \sum_{j \in \mathcal{N}(i)} J_{ij} W_2 h_j^{(t)} + h_i \right),$$

with trainable weights W_1, W_2 and learnable or fixed J_{ij} . This is structurally analogous to one iteration of mean-field inference. Therefore, stacking multiple layers of GNN is somehow similar to running respective number of the mean-field Eqs. (8.5) iterations.

Energy Minimization and GNN Iteration. Interpreting GNN layers as steps in a variational update, one can also view the learned embeddings as approximate minimizers of an energy function. This is especially explicit in recent works that define neural architectures from energy-based loss functionals, including PDE-informed GNNs, where spatial derivatives are replaced by graph Laplacians [61, 62]. The training objective incorporates terms like

$$\mathcal{L}(\theta) = \sum_{i \in V} \ell(f_\theta(h_i), y_i) + \lambda \sum_{(i,j) \in E} \|h_i - h_j\|^2,$$

encouraging smoothness over the graph – again echoing regularization terms in GM-based inference.

From Inference to Representation: GMs \rightarrow GNNs. Whereas GM inference often seeks explicit beliefs or marginals, GNNs focus on learning an embedding h_i that is predictive of the label or useful for downstream tasks. The key similarity is structural: both propagate local updates via the graph. The key difference is functional: GMs propagate messages for probabilistic consistency; GNNs learn update rules to optimize a supervised or unsupervised loss.

Back to Generative AI. Generative models over graphs—e.g., molecular graph generation, generative modeling for point clouds, or simulation of physical systems—demand respecting topological and physical constraints. The synergy of GMs and GNNs enables this: GNNs can encode structural priors in learned transformations; GMs can provide probabilistic scaffolding for sampling or scoring graph-structured data. This hybrid view informs emerging architectures in structured generation, graph diffusion models, and geometric deep learning.

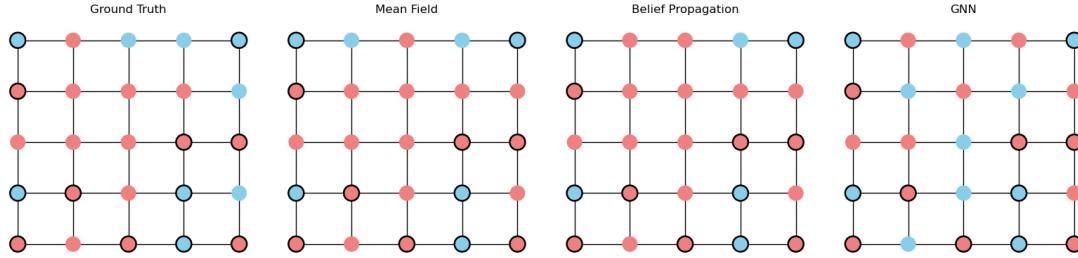


Figure 8.1: **Inference on a Single Ising Sample from a 5×5 Grid.** The ground truth configuration was generated using Gibbs sampling from the Ising model with interaction parameter β and no external field ($h_i = 0$), conditioned on a random subset of visible nodes. Three inference methods are compared: (1) **Mean-Field (MF)** approximates marginals iteratively assuming independence; (2) **Belief Propagation (BP)** uses max-product message passing to estimate MAP configuration; and (3) **Graph Neural Network (GNN)** is trained only on visible nodes with one-hot node identity features and uniform neighbor weights. Observed nodes are marked with black outlines. In this case: Mean Field Accuracy is 0.692, Belief Propagation Accuracy is 0.769, GNN Accuracy is 0.385.

Example 8.2.5 (Comparing Inference Methods on a Grid). Consider a single sample from a binary labeling problem on a $n \times n$ grid graph. Each node i has a label $y_i \in \{-1, +1\}$ representing a spin in an Ising model with no external field ($h_i = 0$). The similarity parameter $\beta > 0$ and the percentage of visible (observed) nodes are treated as hyperparameters. A subset of the nodes is selected uniformly at random and their labels are revealed. The rest are to be inferred.

The full ground truth configuration is generated via Gibbs sampling from the Ising model distribution,

$$P(y) \propto \exp \left(\beta \sum_{\langle i,j \rangle} y_i y_j \right)$$

conditioned on the observed labels being fixed throughout the sampling process.

We compare three inference methods on this single sampled instance:

1. **Mean-Field Inference (MF):** Iteratively updates approximate marginals assuming independence between nodes. The update rule is:

$$p_i^{(t+1)} = \sigma \left(2 \sum_{j \in \mathcal{N}(i)} (2p_j^{(t)} - 1) \right)$$

where $\sigma(z) = 1/(1 + e^{-z})$. Observed nodes are clamped to their known labels.

2. **Belief Propagation (BP)**: Performs iterative max-product style message passing over the grid, approximating MAP configurations. Messages are exchanged over edges and aggregated into local beliefs. Observed nodes are fixed during message updates.
3. **Graph Neural Network (GNN)**: A two-layer message-passing neural network is trained on the observed nodes using one-hot encoded node identity features. All edges are treated equally, and the GNN is trained only on this single sampled instance using the fixed grid adjacency structure.

Outcome: All three methods are able to propagate information from the visible nodes and recover much of the hidden label structure. The quality of inference depends on β and the number of labeled nodes. Visualization (see figure below) highlights differences in behavior across the three methods. Implementation is available in `gnn_vs_gm_grid.py`.

Exercise 8.2.5 (Noisy and Sparse Labeling on Larger Grids). Extend the above setup to a more challenging inference problem involving adding noise to the ground truth sample. Evaluate and report accuracy on the hidden nodes, the cross-entropy loss, and the disagreement between each method and the ground truth. Study how disagreement scales with the hyper-parameters. Modify the grid model by assigning random Gaussian weights to edges. Retrain the GNN using this weighted graph. Does it approximate MF or BP more closely?

Recommended Additional Readings. Connections between energy minimization, graphical models, and Graph Neural Networks (GNNs) have been explored across several research directions. The foundational Graph Convolutional Network (GCN) model by Kipf and Welling [61] makes a direct connection to label propagation in GMs, with Laplacian smoothing acting as a form of variational regularization. Physics-informed neural operators, such as the Fourier Neural Operator (FNO) [62], extend this perspective to continuous domains by linking deep learning with partial differential equations (PDEs), where graphs and graphical models arise from spatio-temporal discretizations. Input Convex Neural Networks (ICNNs) [63] further develop architectures that are explicitly derived from energy-based principles, enforcing convexity in the learned energy landscape and enabling efficient inference via convex optimization.

Chapter 9

Synthesis

Over the past decade, generative modeling has undergone a series of transformative breakthroughs. It began with frameworks like Variational Autoencoders (VAEs) and Generative Adversarial Networks (GANs), and has more recently culminated in the emergence of Diffusion Models (DMs), with Score-Based Diffusions (SBDs) currently representing the state of the art. While these early approaches differ in formulation and motivation, a striking insight has emerged: many pre-diffusion generative models can now be reinterpreted as special or limiting cases within the broader diffusion framework.

This realization motivates us to begin the chapter with a detailed discussion of Score-Based Diffusion Models (SBDs), presented in Section 9.1. In this framework, two stochastic processes are constructed: a forward-time process that incrementally corrupts ground-truth (GT) samples by adding noise, and a reverse-time process that reconstructs data by gradually removing noise – effectively “denoising” and generating new samples. The learned component of the model is the score function, approximated by a NN, which governs the drift of the reverse process.

We then turn to a complementary class of models – Bridge Diffusions (BDs) – introduced in Section 9.1.1. These models build upon the theory of optimal transport and Schrödinger bridges. Like SBDs, BDs begin with a forward stochastic process that diffuses GT data by adding noise over time. However, the reverse process in BDs is fundamentally different: it is a conditioned, “pinned” process that connects a known initial distribution to a specified final configuration, which can be a sample or a distribution. The result is a stochastic bridge that transports samples from a tractable prior (often Gaussian) to a complex target distribution represented by empirical data. Like in the case of the SBD the bridge is represented by a stochastic differential equation with the drift (score) function represented via a NN.

By introducing Score-Based and Bridge Diffusion models early in this chapter, we set the stage for a unified perspective on generative modeling. In Section 9.2, we return to earlier approaches – such as GANs and VAEs – and reinterpret them through the lens of diffusion, revealing them as special cases within this more general and geometrically grounded framework. This synthesis not only clarifies theoretical connections between models, but also highlights new opportunities for hybrid generative techniques.

In Section 9.3. We probe *nonequilibrium learning dynamics* through the lens of statistical-physics phase transitions. After revisiting the “U-turn” diffusion model of [64] that captures memorization-forgetting trade-offs, we show how high-dimensional generative diffusions can

undergo sharp transitions in sample quality and mode coverage, and we list open mathematical questions that connect these phenomena to spin-glass theory.

Section 9.4 is a fast primer on *Markov Decision Processes* as the calculus of sequential decision making. We review value–policy duality, Bellman operators, and entropy-regularised RL, then discuss how physics-inspired priors (e.g. control as inference) enrich classical RL and how modern Gen-AI models already embed RL-style objectives during large-scale fine-tuning.

In Section 9.5 we marry the two worlds covered in the previous sections. Starting with Stochastic Optimal Control (SOC) and *Path-Integral Diffusion* (PID), we follow [65] and identify three increasingly “integrable” regimes where score-based diffusion can be recast as an RL problem – or, conversely, where RL admits a diffusion interpretation – thereby furnishing new algorithms that inherit the strengths of both paradigms.

In Section 9.6 we reinterpret *Generative Flow Networks* (GFNs) of [66] as samplers over decision trajectories rather than over raw data, introduce “Decision Flow” of [67] as an integrable, diffusion-like extension of GFNs, and explain why choosing the *time axis* (artificial vs. physical) is itself a modeling decision that affects sample diversity, credit assignment and compute cost. Finally, the chapter concludes in Section 9.7 with a survey of ongoing projects (space–time and PID diffusions, decision flows) and a “grand-unification” outlook that positions diffusion, autoregression/transfomers and RL as three limit cases of a single mathematical framework. Here, we also highlight open problems and promising downstream applications – from controllable molecule generation to adaptive scientific simulators – that the author (and readers may) wish to pursue in future work.

9.1 Score-Based Diffusion Models

The fundamental mechanics of Artificial Intelligence (AI) involve a three-step cycle: acquiring ground truth (GT) data, constructing a model of this data, and using the model to predict, infer, or generate synthetic data. In generative AI, the last step—data generation—is central, and score-based diffusion (SBD) models have emerged as a particularly powerful class of generative frameworks.

The success of a generative model hinges on how effectively it captures and reconstructs the structure of data. SBD models achieve this by introducing an auxiliary time parameter and leveraging the dynamics of stochastic differential equations (SDEs). They feature a forward-time stochastic process that incrementally corrupts data by injecting noise, and a reverse-time process that learns to undo this corruption. What distinguishes SBD from earlier methods is the use of the score function (i.e., the gradient of the log probability density) to drive the reverse process.

SBD methods unify two major ideas in the literature: the denoising diffusion probabilistic models (DDPM) of [56, 69], and score matching with Langevin dynamics [68]. The resulting framework is most naturally described using SDEs, providing a foundation to invoke Anderson’s theorem [70], which formally connects the forward and reverse dynamics.

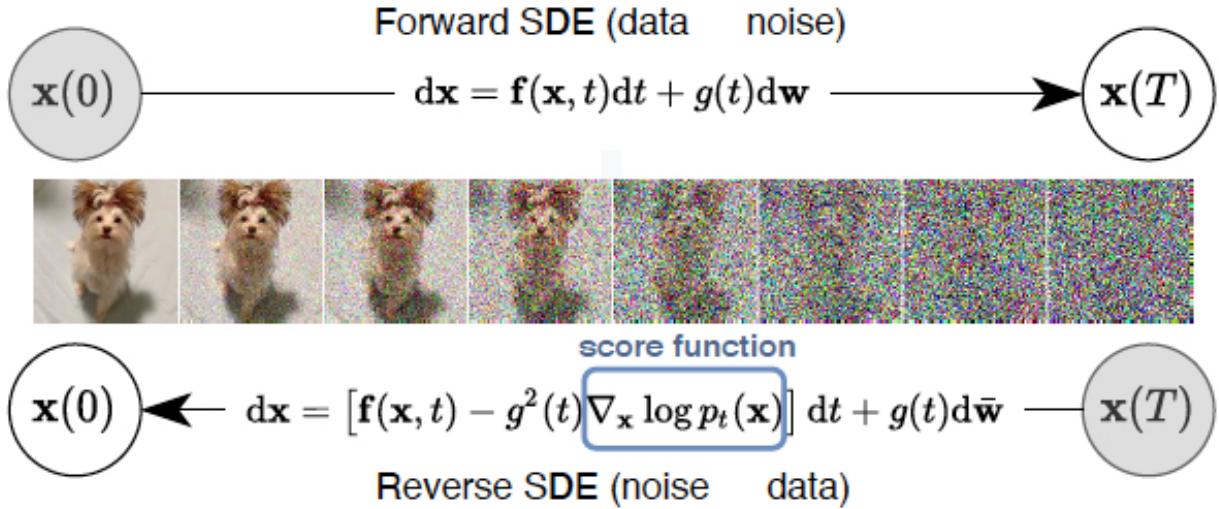


Figure 9.1: Now famous figure – borrowed from [68] – illustrating the core principles of score-based diffusion modeling: a forward (noising) process that gradually corrupts data, a neural network trained to estimate the score function at each noise level, and a reverse process that generates new samples by iteratively denoising.

Forward and Reverse SDEs

Following [57], we define the forward and reverse SDEs for time-indexed vectors $\mathbf{x}_t, \mathbf{y}_t \in \mathbb{R}^d$:

$$\text{Forward: } d\mathbf{x}_t = \mathbf{f}(\mathbf{x}_t, t) dt + \mathbf{g}(\mathbf{x}_t, t) d\mathbf{w}_t, \quad t \in [0, T], \quad (9.1)$$

$$\text{Reverse: } d\mathbf{y}_t = (\mathbf{f}(\mathbf{y}_t, t) - \nabla \cdot \mathbf{G}(\mathbf{y}_t, t) - \mathbf{G}(\mathbf{y}_t, t) s(\mathbf{y}_t, t)) dt + \mathbf{g}(\mathbf{y}_t, t) d\bar{\mathbf{w}}_t, \quad (9.2)$$

where \mathbf{f} is a drift vector field, \mathbf{g} is a diffusion matrix, and $\mathbf{G} = \mathbf{g}\mathbf{g}^\top$. The function $s(\mathbf{x}_t, t) := \nabla_{\mathbf{x}_t} \log p(\mathbf{x}_t, t)$ is the *score function* of the marginal distribution of the forward process (9.1). The noise terms \mathbf{w}_t and $\bar{\mathbf{w}}_t$ are standard Wiener processes in forward and reverse time, respectively.

The forward process maps the data distribution p_{data} to a tractable noise distribution p_T (typically Gaussian) by progressively adding noise. Importantly, the dynamics \mathbf{f}, \mathbf{g} are often chosen independently of data. The reverse process is constructed to invert this trajectory by using the score function $s(\mathbf{x}_t, t)$ as a guiding drift.

Theoretical Foundation: Fokker–Planck and Anderson’s Theorem

To understand the equivalence of marginals between the forward and reverse processes, one can follow the approach from Chapter 7. Applying the Fokker–Planck formalism to both SDEs yields the following forward and backward evolution equations for the probability density $p(\mathbf{x}, t)$:

$$\partial_t p + \nabla_i (f_i p) = \frac{1}{2} \nabla_i \nabla_j (G_{ij} p), \quad (9.3)$$

$$\partial_t p + \nabla_i (f_i p - p \nabla_j G_{ij} - G_{ij} s_j p) = -\frac{1}{2} \nabla_i \nabla_j (G_{ij} p). \quad (9.4)$$

By imposing that both dynamics describe the same time-marginal $p(\mathbf{x}, t)$ (for all t), one sees that the two Fokker–Planck equations are consistent if and only if the reverse drift in (9.2) is as given. This derivation, originally due to Anderson [70], provides the theoretical justification for constructing the reverse process.

Asymptotic Validity. The exact equivalence between the forward and reverse marginals holds in the limit $T \rightarrow \infty$, when the forward diffusion fully converges to the terminal distribution p_T (typically standard normal). In practice, T is taken to be large but finite, and the approximation remains effective. We return to this issue in Section 9.3.1, where we explore finite-time corrections using the U-turn diffusion model of [64].

Beyond Marginals: Pathwise Equivalence. In the classical SBD setting, only marginal distributions of forward and reverse processes match. However, the U-turn model goes further: it constructs a looped process where the full path distributions (transition kernels) of forward and backward segments match. This generalizes the notion of detailed balance (DB), discussed in Chapter 7, to non-equilibrium and time-dependent settings.

Deterministic Alternatives. Matching marginals does not require matching path measures. This observation leads to alternative approaches such as the *probability flow ODE* [57], where the reverse dynamics are deterministic. Although DB is violated at the level of transitions, the marginal distributions remain correctly reproduced.

Time-Dependent Brownian Diffusion

A particularly simple but effective instance of the SBD framework uses Brownian motion with time-varying noise. In this case:

$$\mathbf{f}(\mathbf{x}_t, t) = \mathbf{0}, \quad \mathbf{g}(\mathbf{x}_t, t) = \sqrt{2\beta_t} \mathbf{I}. \quad (9.5)$$

The forward process is then an isotropic Brownian motion with time-dependent diffusion strength. The corresponding score function has a closed-form expression:

$$\mathbf{s}(\mathbf{x}_t, t) = \nabla_{\mathbf{x}_t} \log \left(\sum_{n=1}^N \mathcal{N} \left(\mathbf{x}_t \middle| \mathbf{x}^{(n)}, 2\mathbf{I} \int_0^t \beta_\tau d\tau \right) \right), \quad (9.6)$$

where $\{\mathbf{x}^{(n)}\}_{n=1}^N$ is the empirical dataset. This closed-form score bypasses the need to simulate the forward process, greatly simplifying training and analysis.

Discretization and β -Protocols

In practice, continuous-time SDEs are implemented via time discretization. The interval $[0, T]$ is divided into a finite number of steps, and the noise schedule is governed by the so-called β -protocol, i.e., the function β_t . This function determines how aggressively noise is injected (forward) or removed (reverse).

Space–Time Diffusion

Choosing an elementary forward dynamics – for instance the Brownian motion in Eq. (9.5) – is attractive because the forward path need not be simulated and the score can be written in closed form via Eq. (9.6). Yet, in many imaging tasks one benefits from weaving problem-specific structure into the forward process. Remarkably, this can be done *without* forfeiting analytical tractability. A prototypical construction, introduced in [71], augments the Brownian drift by a linear term

$$f_i(\mathbf{x}_t, t) = \sum_j L_{ij} x_{j,t}, \quad \text{with } \mathbf{L} \text{ the graph Laplacian of the image grid.}$$

Because the Laplacian encodes the geometry of the underlying pixel graph, the modified forward diffusion better aligns with the data manifold; empirically, this yields sharper and more faithful samples while the score remains available in closed form.

Training

The score network \mathbf{s}_θ is trained via the *denoising score matching* (DSM) objective:

$$\mathcal{L}_{\text{DSM}}(\theta) = \mathbb{E}_{t \sim U(0,T), \mathbf{x}_0 \sim p_{\text{data}}, \mathbf{x}_t \sim p(\mathbf{x}_t | \mathbf{x}_0)} \left[\frac{\lambda(t)}{2} \|\nabla_{\mathbf{x}_t} \log p(\mathbf{x}_t | \mathbf{x}_0) - \mathbf{s}_\theta(\mathbf{x}_t, t)\|^2 \right],$$

where $\lambda(t)$ is a weighting function over time. This loss encourages the NN to predict the conditional score of noisy samples given clean ones. In practice, this training is done using finite samples from the dataset and simulated noise injections.

Inference

Generating new samples amounts to simulating the reverse SDE (9.2) from $t = T$ to $t = 0$. Initialization is done by sampling from a known reference distribution (typically Gaussian), which approximates the final distribution p_T of the forward process. To guide this reverse-time evolution, the score function $s(\mathbf{x}_t, t)$ must be estimated.

As computing the exact score function is generally intractable, it is approximated by a neural network:

$$\mathbf{s}_\theta(\mathbf{x}_t, t) \approx \nabla_{\mathbf{x}_t} \log p(\mathbf{x}_t, t),$$

where θ denotes trainable parameters. The learned score network enables efficient and high-quality generation in complex data domains.

Example 9.1.1. Score-Based Diffusion on a 3×3 Grid Graph *Building on the theory of Score-Based Generative Modeling (SGM) developed earlier in this section, we now demonstrate its implementation on a synthetic 3×3 grid dataset. This toy example allows us to concretely explore the relationship between forward stochastic differential equations (SDEs), their time-reversed counterparts, and learned score functions.*

Our setup closely follows and extends a <https://mfkasim1.github.io/2022/07/01/sgm-1/>, modified in the companion notebook `02-SGM-with-SDE-9grid.ipynb`. The goals of this computational experiment are to:

- Simulate the forward process using the Euler-Maruyama scheme on grid-structured data;
- Learn the score function $\nabla_x \log p_t(x)$ via denoising score matching;
- Use the learned score to simulate the reverse-time SDE and generate samples;
- Investigate how diffusion parameters and discretization impact sample quality.

The notebook provides implementation details and visualizations of both noisy trajectories and reverse-time generations under different regimes. The example serves as a springboard for the exercise that follows.

Exercise 9.1.1 (Exploring the Role of Reverse Process Noise in Score-Based Diffusion). Let the target distribution be a mixture of Gaussians as introduced in the example above (refer to the Jupyter notebook `02-SGM-with-SDE-9grid.ipynb`). Consider a forward noising process experimented with in the notebook. In this exercise, we explore the effect of modifying the diffusion coefficient in the reverse process.

- (a) **Generalized Reverse Fokker-Planck Equation.** Starting from the standard Fokker-Planck formulation of the reverse process (Eq. (9.4)), re-derive the expression assuming the diffusion coefficient of the reverse process, denoted $\tilde{\mathbf{G}}(\mathbf{y}_t, t)$, is allowed to differ from the forward process diffusion coefficient $\mathbf{G}(\mathbf{x}_t, t)$.
- Derive the modified expression for the score function $\nabla_x \log p_t(x)$ under this more general setting, requiring that the marginal distributions in the forward and reverse processes coincide.
 - Provide a corresponding empirical estimate of the score function assuming that $p_t(x)$ is approximated by a sample-based distribution.
- (b) **Sampling Quality vs. Reverse Noise Level.** Now assume the reverse diffusion coefficient is a constant $\tilde{\mathbf{G}}(\mathbf{y}_t, t) = \varepsilon \mathbf{1}$ (i.e., independent of time and space, and uniform). Experiment with varying values of ε in the reverse SDE while keeping the forward diffusion fixed. For each choice of ε :
- Evaluate the sampling quality empirically. Suggested metrics include visual comparison to the target distribution and (optionally) KL divergence between empirical and ground-truth distributions.
 - Study how the number of reverse-time steps (i.e., time discretization granularity) and number of samples impact the quality of approximation, especially for small and large ε .

Discuss the trade-off between noise strength and stability of the reverse dynamics. What trends emerge in terms of accuracy and convergence?

9.1.1 Bridge Diffusion

Standard (score based) diffusion described above is formally exact only at $T \rightarrow \infty$ when statistics of $\mathbf{x}(T)$ becomes independent on the pre-history. Can we keep T finite (say to $T = 1$) while also making statistics of $\mathbf{x}(T)$ fixed – independent on the pre-history?

Schrödinger Bridge, which we about to discuss next, is a modification to the forward process which makes $\mathbf{x}(T)$ not only independent on the pre-history, but moreover fixed/pinned to a pre-defined value.

Schrödinger Bridges

Consider the following general SDE:

$$d\mathbf{x}(t) = \mathbf{f}(t; \mathbf{x}(t)) dt + \sqrt{\mathbf{G}(t; \mathbf{x}(t))} d\mathbf{w}_t, \quad (9.7)$$

with forward and backward Fokker-Planck operators defined respectively by

$$\left(\partial_t - \hat{\mathcal{L}}_t^* \right) p(\mathbf{x}(t) | \mathbf{x}(0)) = 0, \quad \hat{\mathcal{L}}_t^* = -\nabla \cdot \mathbf{f} + \frac{1}{2} \nabla_i \nabla_j G_{ij}, \quad (9.8)$$

$$\left(\partial_t + \hat{\mathcal{L}}_t \right) p(\mathbf{x}(1) | \mathbf{x}(t)) = 0, \quad \hat{\mathcal{L}}_t = \mathbf{f} \cdot \nabla + \frac{1}{2} G_{ij} \partial_i \partial_j. \quad (9.9)$$

The Schrödinger bridge process conditions the SDE to hit a terminal point $\mathbf{x}(1)$, modifying the dynamics via a Doob h -transform [72]:

$$d\mathbf{x}(t) = (\mathbf{f}(t; \mathbf{x}) + \mathbf{G} \nabla_{\mathbf{x}} \log p(\mathbf{x}(1) | \mathbf{x}(t))) dt + \sqrt{\mathbf{G}} d\mathbf{w}_t. \quad (9.10)$$

Bridge Score. We can now replace the standard forward process – governed by Eqs. (9.1, 9.3, 9.5) – by Eq. (9.10) with $\mathbf{x}(1)$ sampled from a tractable distribution, for example a Gaussian. See [71] and references there for more details.

9.2 A Unified View: Generative Models as Diffusions

This section offers a unified synthesis by showing how several foundational generative paradigms – such as VAEs and GANs – can be viewed as special cases of diffusion-based models. These connections emerge either in the deterministic zero-noise limit or through mechanisms like conditioning and entropic regularization. Fig. (9.2) visualizes the landscape of generative modeling, highlighting how core diffusion models – Score-Based Diffusions and Diffusion Bridges, as introduced in the previous section – relate to and generalize their predecessors, which we examine in the subsections that follow.

9.2.1 GANs as Implicit Diffusion Models

Generative Adversarial Networks (GANs), introduced by Goodfellow et al. [73], consist of two competing neural networks:

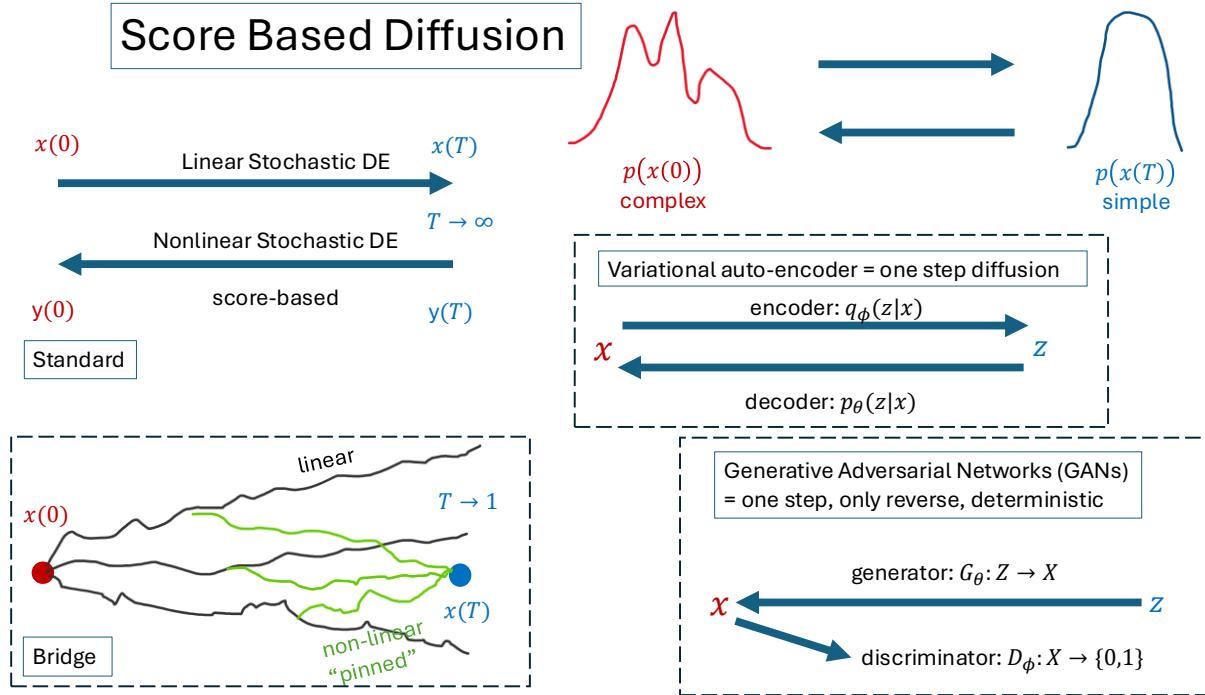


Figure 9.2: Illustration of main concepts behind diffusion models.

- A **generator** $G_\theta : \mathcal{Z} \rightarrow \mathcal{X}$, which maps latent variables $z \sim \mathcal{N}(0, \mathbf{I})$ to the data space;
- A **discriminator** $D_\phi : \mathcal{X} \rightarrow [0, 1]$, which estimates the probability that a given sample is drawn from the true data distribution.

The networks are trained via a min-max optimization objective:

$$\min_{\theta} \max_{\phi} \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}} [\log D_\phi(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p(\mathbf{z})} [\log(1 - D_\phi(G_\theta(\mathbf{z})))]. \quad (9.11)$$

GANs as Deterministic One-Step Reverse Diffusions. While GANs lack an explicit forward stochastic process, the generator G_θ can be interpreted as implementing a degenerate reverse-time stochastic differential equation (SDE):

$$\mathbf{z} \sim \mathcal{N}(0, \mathbf{I}) \quad \mapsto \quad \mathbf{x} = G_\theta(\mathbf{z}),$$

which transports noise to data in a single, deterministic step. This is analogous to a reverse diffusion with horizon $T = 1$ and zero noise, contrasting with the many-step stochastic processes of typical score-based models.

GANs as Approximate Score Matchers. Recent work [74] reveals that under certain conditions, GANs implicitly estimate score functions. For example, when trained with Integral Probability Metrics (IPMs) such as Maximum Mean Discrepancy (MMD) or Wasserstein distance, the learned generator implicitly aligns gradients of the log densities:

$$\nabla_{\mathbf{x}} \log p_G(\mathbf{x}) \approx \nabla_{\mathbf{x}} \log p_{\text{data}}(\mathbf{x}),$$

paralleling objectives found in denoising score matching for training score-based diffusion models (SBDs).

Example 9.2.1 (Linear Generator Matching Gaussian Data). Let the GAN generator be linear: $G_\theta(\mathbf{z}) = W\mathbf{z} + b$, where $W \in \mathbb{R}^{d \times d}$ and $b \in \mathbb{R}^d$. Suppose the latent prior is $\mathbf{z} \sim \mathcal{N}(0, I)$, and the real data is distributed as $\mathbf{x} \sim \mathcal{N}(\mu, \Sigma)$.

We aim to match the generated distribution p_G with p_{data} . Since affine transformations of Gaussians are Gaussian:

$$G_\theta(\mathbf{z}) \sim \mathcal{N}(b, WW^\top).$$

Matching the mean and covariance with the data distribution implies:

$$b = \mu, \quad WW^\top = \Sigma.$$

Thus, even this simple linear case demonstrates the generator as a deterministic transport map from prior to data, designed to align second-order statistics. It serves as an explicit one-step instantiation of a (deterministic) reverse diffusion.

GANs as Zero-Noise Schrödinger Bridges. From the perspective of stochastic interpolation [75], GANs can be seen as a limiting case of Schrödinger bridges. These bridges define stochastic flows between prior and target distributions, regularized by entropy. As the noise level $\varepsilon \rightarrow 0$, the bridge converges to a deterministic map – precisely the behavior implemented by a GAN generator. This interpretation connects GANs to diffusion-based models through the lens of optimal stochastic control and entropy minimization.

Exercise 9.2.1. Consider the SDE:

$$d\mathbf{x}(t) = \mathbf{f}(t; \mathbf{x}(t)) dt + \sqrt{\varepsilon \mathbf{I}} d\mathbf{w}_t, \quad \mathbf{x}(0) \sim p_0,$$

where \mathbf{f} is a drift term and $\varepsilon > 0$ is a small noise parameter. The Schrödinger bridge modifies this dynamics so that the process is conditioned to hit a target distribution p_1 at time $t = 1$:

$$d\mathbf{x}(t) = (\mathbf{f}(t; \mathbf{x}) + \varepsilon \nabla_{\mathbf{x}} \log p(\mathbf{x}(1) | \mathbf{x}(t))) dt + \sqrt{\varepsilon} d\mathbf{w}_t.$$

1. Show that as $\varepsilon \rightarrow 0$, the modified dynamics converge (formally) to a deterministic transport map $\mathbf{x}(0) \mapsto \mathbf{x}(1)$ that pushes forward p_0 to p_1 .
2. Using the result of the preceding example, interpret a linear generator $G_\theta(\mathbf{z}) = W\mathbf{z} + b$ as the zero-noise limit of a Schrödinger bridge where both p_0 and p_1 are Gaussian.
3. (**Conceptual**) In the GAN framework, the generator learns such a deterministic map from noise to data. The discriminator, which measures divergence between the generated and real distributions, implicitly enforces the terminal constraint $p_G \approx p_{\text{data}}$.

Compare this to the role of the correction term $\nabla \log p(\mathbf{x}(1) | \mathbf{x}(t))$ in the bridge formulation: in both cases, there is an adaptation of a prior-driven stochastic process to fit an endpoint constraint. Discuss how this analogy helps explain the adversarial (min-max) structure of GAN training.

Hint: In the limit $\varepsilon \rightarrow 0$, stochastic paths concentrate along the most likely trajectories—those minimizing the action. This is closely related to optimal transport. The GAN generator can be interpreted as learning one such path, while the discriminator ensures that the pushforward distribution matches p_1 .

9.2.2 Variational Autoencoders as Diffusion Models

As introduced in Section 8.1.4, Variational Autoencoders (VAEs) [54, 76] combine latent-variable graphical models with deep neural networks for inference and generation. The posterior $p_\theta(z|x)$ is approximated by a tractable encoder $q_\phi(z|x)$, optimized using the variational ELBO objective:

$$\mathcal{L}(\theta, \phi; x) = \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)] - D_{\text{KL}}(q_\phi(z|x) \| p(z)). \quad (9.12)$$

A standard VAE specifies:

- A prior: $p(z) = \mathcal{N}(0, I)$;
- A decoder: $p_\theta(x | z)$;
- An encoder: $q_\phi(z | x)$.

VAEs as One-Step Diffusions. A VAE defines two stochastic maps:

$$x \xrightarrow{q_\phi} z \sim \mathcal{N}(\mu_\phi(x), \sigma_\phi^2(x)) \quad z \xrightarrow{p_\theta} x \sim \mathcal{N}(\mu_\theta(z), \sigma^2 I),$$

interpretable as a one-step forward (encoding) and reverse (decoding) diffusion. Unlike score-based models, the encoder and decoder are learned jointly.

Hierarchical VAEs and Discrete Diffusions. Ladder VAEs [77] extend this to T latent layers:

$$z_T \sim p(z_T), \quad z_{t-1} \sim p_\theta(z_{t-1} | z_t), \quad x \sim p_\theta(x | z_0),$$

with encoder transitions:

$$q_\phi(z_1 | x), \quad q_\phi(z_2 | z_1), \quad \dots, \quad q_\phi(z_T | z_{T-1}).$$

This structure discretizes forward and reverse stochastic processes in latent space.

Continuous-Time Limit. As $T \rightarrow \infty$, the latent path becomes a diffusion:

$$dz_t = f_\theta(t, z_t) dt + \sqrt{\Sigma_\theta(t, z_t)} dw_t.$$

The decoder models the reverse-time SDE, and the encoder approximates the forward process.

Example 9.2.2 (Latent Diffusion via Two-Layer VAE on a Ring). This example, based on the `ring_vae_latent_diffusion_comparison.ipynb` notebook, compares a classical VAE with a two-layer decoder model on a synthetic ring dataset.

In the baseline VAE, we learn:

$$q_\phi(z | x) = \mathcal{N}(z; \mu_\phi(x), \sigma_\phi^2(x)), \quad p_\theta(x | z) = \mathcal{N}(x; \mu_\theta(z), \sigma^2 I).$$

Trained using the ELBO objective, the VAE reconstructs the ring accurately and learns a latent ring structure, but samples from the prior yield a narrow distribution with low variance. To improve generative diversity, we freeze the VAE encoder and extend the decoder to two steps:

$$z_2 \sim \mathcal{N}(0, I), \quad z_1 = f_\theta(z_2) + \epsilon, \quad x = g_\theta(z_1) + \eta,$$

with $\epsilon \sim \mathcal{N}(0, \beta I)$ and β annealed (i.e. grow in time) during training. The decoder networks f_θ and g_θ are trained from scratch; KL regularization is applied to z_1 .

This setup mimics the reverse dynamics of score-based models: the latent variable z_2 is progressively denoised toward the VAE manifold. Compared to the single-layer VAE, this model produces samples with higher variability and more accurate thickness, while maintaining competitive MSE and FID. It forms a concrete bridge between amortized inference and learned latent diffusion.

Exercise 9.2.2 (Exploring Tradeoffs in Latent Diffusion on a Ring). This exercise invites you to analyze and extend the models introduced above.

1. Visualize and compare latent encodings $z_1 = \mu_\phi(x)$:

- How does the geometry of z_1 differ between the models?
- Is the ring structure preserved under the noise-to-denoise mapping $z_2 \rightarrow z_1$?

2. Analyze the tradeoff between MSE and FID:

- Plot reconstruction error vs. angle and radius;
- Examine which modes or regions are better captured by the two-layer model;
- Compare pairwise distance histograms across samples.

3. Explore the impact of the noising level β :

- Sweep over different fixed β values;
- Plot training dynamics (e.g., KL loss vs. epoch);
- Evaluate diversity vs. stability.

4. Replace fixed β with a learned noise schedule:

$$\beta = h_\psi(z_2) \quad \text{or} \quad \beta_t \quad (\text{trainable scalar}).$$

Visualize how β varies over latent space and how it impacts FID.

5. (Optional) Add a third decoder layer: $z_3 \rightarrow z_2 \rightarrow z_1 \rightarrow x$. Does increased depth improve flexibility or introduce instability?

9.3 Diffusion Models and Dynamic Phase Transitions

The dynamics of score-based diffusion models, especially when examined through the lens of a truncated scheme, reveal a rich structure of dynamical regimes that can be understood through a combination of empirical insights and theoretical tools from statistical physics. In this section, we synthesize recent findings from the U-Turn Diffusion model [64] and statistical mechanics analyses of standard diffusion processes [78, 79] to develop a unified conceptual and mathematical framework for dynamic phase transitions in generative diffusion.

9.3.1 U-Turn Diffusion and Memorization Dynamics

In U-Turn Diffusion [64], a pre-trained score-based diffusion model is modified by terminating the forward noising process at an intermediate time $T_u < 1$, and initializing the reverse process at that same point using the sample \mathbf{x}_{T_u} obtained from the forward process. This construction uncovers several key phenomena:

- A *memorization time* T_m , beyond which the reverse-generated samples begin to deviate from the specific Ground Truth (GT) sample that seeded the forward process.
- A *speciation time* $T_s > T_m$, beyond which generated samples transition into different semantic classes, indicating a qualitative phase transition in generative behavior. In multi-class settings, this may correspond to multiple speciation events.
- The effective non-linearity of the score function $s_t(\mathbf{x})$ also evolves: it is strongly non-linear for $t < T_m$, approximately affine in the interval $T_m < t < T_s$, and nearly affine beyond T_s .

These transitions are measurable through quantities such as sample auto-correlation functions, divergence from GT trajectories, and structural changes in the score field.

Varying T_u allows us to probe different dynamical regimes:

1. For $T_u < T_m$, the model output remains close to the original GT, with only local latent perturbations.
2. For $T_m < T_u < T_s$, the dynamics enter a metastable interpolation zone where sample diversity grows and the score becomes effectively linear.
3. For $T_u > T_s$, the model transitions to a different generative basin, producing samples from other classes or semantic regions.

9.3.2 Dynamic Phase Transitions in High Dimensions

The theoretical framework developed in [78, 79] provides foundational support for these empirical observations. In their high-dimensional asymptotic setting:

- The **speciation transition** corresponds to a symmetry-breaking phenomenon, whereby diffusion trajectories begin to resolve global structure in the data manifold.

- The **collapse transition** (aligned with the memorization transition) marks the onset of trajectory attraction to specific data points—analogous to condensation into metastable states seen in glassy systems.
- These transitions can be diagnosed via spectral analysis of the data correlation matrix and via information-theoretic quantities like excess entropy.

9.3.3 Open Problems

The U-Turn formulation can be applied to any pre-trained diffusion model, offering a versatile tool for exploring and analyzing model dynamics. Harnessing this formulation to improve training, sampling, and interpretability remains an open area of research.

Exercise 9.3.1 (Experiments with U-Turn Diffusion).** Consider a simple multi-class generative setting such as the 9-mode Gaussian mixture in the notebook `02-SGM-with-SDE-9grid.ipynb`, and study the following:

1. How does the distribution of outputs $\mathbf{y}(0)$ evolve with increasing T_u ? How many distinct transitions are observed? How sharp are they?
2. How does the dynamic hierarchy of phase transitions change when altering the forward protocol or modifying the reverse-time diffusion coefficient?
3. Can sampling of specific modes or combinations of modes be encouraged or discouraged by tuning T_u or other hyperparameters?
4. Can we turn the information collected about the hierarchy of the dynamic transition with the U-turn trick to self-classify samples?
5. Add your own experimental idea or theoretical analysis related to dynamic regimes in U-Turn diffusion.

9.4 From Markov Decision Processes to Reinforcement Learning

In the previous sections, we examined stochastic processes and their Markovian specializations, where the future evolution depends only on the present state (the Markov property). We now introduce *Markov Decision Processes (MDPs)*, a powerful framework for modeling *controlled* Markov processes – i.e., scenarios in which an agent can take actions that influence both state transitions and associated costs or rewards. Afterward, we bridge to *Reinforcement Learning (RL)*, which provides data-driven, and often model-free, methods for solving MDPs in complex environments. This section offers only a succinct introduction and thus we direct the reader to specialized monographs such as [48, 80] for details and a more comprehensive treatment of advanced topics.

9.4.1 Markov Decision Processes

An MDP generalizes a Markov chain by allowing an *action* a to be chosen at each step. Formally, an MDP is defined by:

- **States:** A set of possible states \mathcal{S} .
- **Actions:** A set of possible actions \mathcal{A} . Depending on the problem, \mathcal{A} may be discrete (*e.g.*, move left/right) or continuous (*e.g.*, real control inputs).
- **Transition probabilities:** For each state $s \in \mathcal{S}$ and action $a \in \mathcal{A}$, a distribution $p(s' | s, a)$ governing the probability of transitioning from state s to next state s' .
- **Reward function:** $r(s, a)$, specifying the immediate reward (or negative cost) after performing action a in state s . Some formulations assign reward to the tuple (s, a, s') instead.

A trajectory of length T in an MDP is a sequence

$$s_0, a_0, s_1, a_1, \dots, s_{T-1}, a_{T-1}, s_T,$$

where a_t is chosen by the agent in state s_t according to a *policy* π . A *policy* π is a (possibly stochastic) mapping from states to actions:

$$\pi(a | s) = \text{Prob}(\text{take action } a | \text{current state } s).$$

Goal: Optimal Policy. Given an MDP, one typically seeks a policy π^* that maximizes the *expected cumulative reward* (or minimizes cumulative cost). A common objective is the *discounted reward*:

$$J(\pi) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) \right], \quad (9.13)$$

where $\gamma \in [0, 1]$ is a discount factor, and thus

$$\pi^* = \arg\min_\pi J(\pi). \quad (9.14)$$

Value Functions and the Bellman Equations. A central concept is the *value function* $V^\pi(s)$, defined as the expected return (cumulative reward) starting in state s and following policy π :

$$V^\pi(s) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) \right] \quad \text{where } s_0 = s.$$

One can show that V^π satisfies the *Bellman equation*:

$$V^\pi(s) = r(s, \pi(s)) + \gamma \sum_{s'} p(s' | s, \pi(s)) V^\pi(s'). \quad (9.15)$$

The optimal value function V^* (corresponding to some optimal policy π^*) satisfies the *Bellman optimality equation*:

$$V^*(s) = \max_{a \in \mathcal{A}} \left\{ r(s, a) + \gamma \sum_{s'} p(s' | s, a) V^*(s') \right\}.$$

Solving these equations – either via *dynamic programming* when the model $p(s' | s, a)$ and $r(s, a)$ is known, or via data-driven methods – gives a systematic way to find or approximate the best action in each state.

Historical Note: Bellman, Kantorovich, and the Birth of Dynamic Programming. Richard Bellman introduced the principle of optimality and formulated many fundamental ideas in what he called “Dynamic Programming” [81], working at the RAND Corporation in the 1950s. Almost in parallel (though independently, and somewhat earlier), Leonid Kantorovich in the Soviet Union developed related methods for optimizing multi-stage resource allocation problems [82, 83]. Their approaches converge around the idea that complex decision-making tasks can be decomposed into simpler subproblems, each solvable recursively. Bellman’s pioneering insight about the “recursive structure” of optimal solutions (captured by what we now call the Bellman equations) and Kantorovich’s frameworks for sequential planning together laid the groundwork for modern Dynamic Programming and, by extension, Markov Decision Process (MDP) theory in control and AI.

Example 9.4.1 (Bellman’s DP for a Simple Two-State MDP.). Consider an MDP with two states, S_1 and S_2 , and two possible actions available in each state. Denote these actions by:

$$\mathcal{A}(S_1) = \{\text{Action A, Action B}\}, \quad \mathcal{A}(S_2) = \{\text{Action C, Action D}\}.$$

Assume a discount factor $\gamma \in (0, 1)$ and the following immediate rewards and transition probabilities:

- **State S_1 :**

$$\text{Action A: } r(S_1, A) = 1,$$

$$P(S_1 | S_1, A) = 0.5, \quad P(S_2 | S_1, A) = 0.5.$$

$$\text{Action B: } r(S_1, B) = 0,$$

$$P(S_2 | S_1, B) = 1.0.$$

- **State S_2 :** (Example choice – any two actions with different payoffs/probabilities)

$$-\text{ Action C: } r(S_2, C) = 2,$$

$$P(S_1 | S_2, C) = 0.6, \quad P(S_2 | S_2, C) = 0.4;$$

$$-\text{ Action D: } r(S_2, D) = 0,$$

$$P(S_1 | S_2, D) = 0.2, \quad P(S_2 | S_2, D) = 0.8.$$

We seek the optimal value function V^* , which satisfies the Bellman optimality equations:

$$\begin{aligned} V^*(S_1) &= \max \left\{ 1 + \gamma [0.5 V^*(S_1) + 0.5 V^*(S_2)], 0 + \gamma [1.0 V^*(S_2)] \right\}, \\ V^*(S_2) &= \max \left\{ 2 + \gamma [0.6 V^*(S_1) + 0.4 V^*(S_2)], 0 + \gamma [0.2 V^*(S_1) + 0.8 V^*(S_2)] \right\}. \end{aligned}$$

The first line represents choosing the better of two actions in S_1 , while the second line captures choosing between the two actions in S_2 . By rearranging (for each state) and solving for $V^*(S_1)$ and $V^*(S_2)$, one obtains the optimal values.

Solution Sketch (Value Iteration):

1. Initialize $V_0(S_1)$ and $V_0(S_2)$ arbitrarily (say 0).

2. At iteration $k = 0, 1, 2, \dots$ update:

$$\begin{aligned} V_{k+1}(S_1) &= \max \left\{ 1 + \gamma [0.5 V_k(S_1) + 0.5 V_k(S_2)], \gamma V_k(S_2) \right\}, \\ V_{k+1}(S_2) &= \max \left\{ 2 + \gamma [0.6 V_k(S_1) + 0.4 V_k(S_2)], \gamma [0.2 V_k(S_1) + 0.8 V_k(S_2)] \right\}. \end{aligned}$$

3. Repeat until $|V_{k+1}(S_i) - V_k(S_i)| < \varepsilon$ for $i = 1, 2$ (for some small tolerance ε).

The result is $V^*(S_1) = \lim_{k \rightarrow \infty} V_k(S_1)$ and $V^*(S_2) = \lim_{k \rightarrow \infty} V_k(S_2)$. With these values in hand, the optimal policy π^* is simply the action in each state that achieves the maximum in the Bellman equations. Hence, even in this tiny example, Bellman's DP (here, instantiated as Value Iteration) systematically yields the optimal decision rule for each state, effectively enumerating the possible actions and selecting those that maximize long-term discounted reward.

9.4.2 Reinforcement Learning

In *Reinforcement Learning (RL)*, we still adopt the MDP framework but often do *not* assume prior knowledge of $p(s' | s, a)$ or $r(s, a)$ in closed form. Instead, an *agent* explores the environment, collecting trajectory data $\{(s_t, a_t, r_t, s_{t+1})\}$, and from these experiences *learns* a policy π to maximize cumulative rewards. RL has emerged as a leading AI paradigm for tasks such as robotic control, game-playing, and resource management. See, for instance, [48, 84, 85] for historical developments and a survey of core RL methods.

Historical Note: Early Origins of Reinforcement Learning. Although Bellman's Dynamic Programming (DP) ideas paved the way, the term "Reinforcement Learning" came into prominence in the late 1980s and early 1990s. Pioneering work by researchers such as Sutton, Barto, and Watkins introduced foundational concepts like temporal-difference learning and Q-learning. The community drew inspiration from psychology (animal learning) and control theory, creating a distinctive subfield bridging trial-and-error learning and optimal control.

Value-based Methods: Q-Learning. A fundamental RL algorithm is *Q-learning*, which learns an approximation \hat{Q} of the *state-action value function*, $Q^\pi(s, a)$. The quantity $Q^\pi(s, a)$ represents the expected discounted return if the agent takes action a in state s and subsequently follows policy π . For an optimal policy, Q^* satisfies the Bellman optimality equation:

$$Q^*(s, a) = r(s, a) + \gamma \sum_{s'} p(s' | s, a) \max_{a' \in \mathcal{A}} Q^*(s', a').$$

The Q-learning algorithm updates an estimate \hat{Q} iteratively via experience tuples (s, a, r, s') :

$$\hat{Q}_{k+1}(s, a) \leftarrow \hat{Q}_k(s, a) + \eta \left[r + \gamma \max_{a' \in \mathcal{A}} \hat{Q}_k(s', a') - \hat{Q}_k(s, a) \right],$$

where η is a learning rate. Provided all state-action pairs are explored sufficiently often, $\hat{Q}_k \rightarrow Q^*$ in the limit. Once \hat{Q} converges, the policy π is taken as

$$\pi(s) = \arg \max_{a \in \mathcal{A}} \hat{Q}(s, a).$$

Policy-based Methods. Instead of directly learning a value function, *policy-based methods* parameterize $\pi_\theta(a | s)$ and optimize the parameters θ to improve returns. A classic example is the *policy gradient*:

$$\nabla_\theta J(\theta) = \mathbb{E} \left[\nabla_\theta \log \pi_\theta(a | s) (R - \text{baseline}(s)) \right],$$

where J is defined in Eq. (9.13) – with π parameterized by θ – and R is the (empirical) return from states visited under π_θ . In modern RL (*e.g.*, proximal policy optimization, actor-critic methods), neural networks approximate both π_θ and relevant value functions. The synergy between MDP theory, function approximation, and large-scale optimization drives state-of-the-art RL.

Example 9.4.2 (Gridworld Meets Q-Learning). Consider a 4×4 grid-world. The agent's state is the location (x, y) on the grid. The agent can take actions {left, right, up, down}, unless blocked by walls or boundaries. Some squares yield negative rewards, and one “goal” square yields a large positive reward.

1. **Initialization:** Start with $\hat{Q}(s, a) = 0$ for all (s, a) .
2. **Exploration:** Use an ε -greedy policy: pick random actions with probability ε , or pick $\arg \max_{a \in \mathcal{A}} \hat{Q}(s, a)$ otherwise.
3. **Update:** For each transition (s, a, r, s') , update

$$\hat{Q}(s, a) \leftarrow \hat{Q}(s, a) + \eta \left[r + \gamma \max_{a'} \hat{Q}(s', a') - \hat{Q}(s, a) \right].$$

4. **Convergence:** After sufficient iterations, $\hat{Q} \approx Q^*$. The policy that chooses $\arg \max_a \hat{Q}(s, a)$ yields near-optimal navigation to the goal.

Exercise 9.4.1 (MDP and RL Basics). 1. *Refresher.* Show that for an MDP with finite \mathcal{S} and \mathcal{A} , there exists at least one optimal policy (though not necessarily unique). Hints: argue the existence of a solution to the Bellman optimality equations, or convert the problem into an equivalent linear program if you wish.

2. *Tabular Q-Learning.* Implement a basic Q-learning routine for the 4×4 grid-world in Example 9.4.2. Plot or tabulate the learned Q-values for each state-action pair, then visualize or output the final policy.
3. *Stochastic Rewards.* Modify your environment so that each state-action yields a reward drawn from a distribution with noise (e.g., Normal or Bernoulli). How does this affect convergence speed and policy performance?

Key Takeaways from “Classic” MDP and RL:

- **MDPs** capture Markovian environments where an agent selects actions that influence both state transitions *and* rewards.
- **Reinforcement Learning (RL)** methods like Q-learning and policy gradients enable learning near-optimal policies even in the absence of explicit transition models.

The AI revolution has extended significance of MDP and RL in a number of ways. First of all, it has re-enforced the classic MDP and RL methods by **combining MDP theory with function approximation** via Neural Nets, and thus has led to breakthroughs in AI: superhuman board-game play, robotic control, and guiding large language models with *reinforcement signals*. In a broader context – while standard RL frameworks assume an interactive environment, generative AI tasks re-purpose these concepts for optimizing high-level objectives or preferences, often driven by data and human feedback. These approaches are still very much state of the art – under very impressive and cutting edge development. See, in particular, recent tutorials and surveys detail the intersection of RL with generative modeling and large language models: [86, 87].

Another cutting-edge topic, somewhat less explored, is about reinforcing control tasks using the latest AI tools within environments described by physics-informed or, more broadly, science-informed models. A compelling example is elaborated in the next subsection.

9.4.3 Physics-Informed and AI-enabled Reinforcement Learning

This subsection builds on insights from [88], focusing on a novel Physics-Informed Reinforcement Learning (PIRL) strategy for controlling particle dynamics in turbulent flows. Reinforcement Learning (RL) traditionally leverages interaction-based learning to optimize control strategies under uncertainty. Standard Actor-Critic (AC) methods use neural networks (NNs) to represent both the control policy (actor) and the value function (critic). However, [88] propose an innovative modification termed the Actor-Physicist (AP), replacing the NN-based critic with a physics-derived analytical heuristic termed the “physicist”—see Fig. (9.3) for illustration.

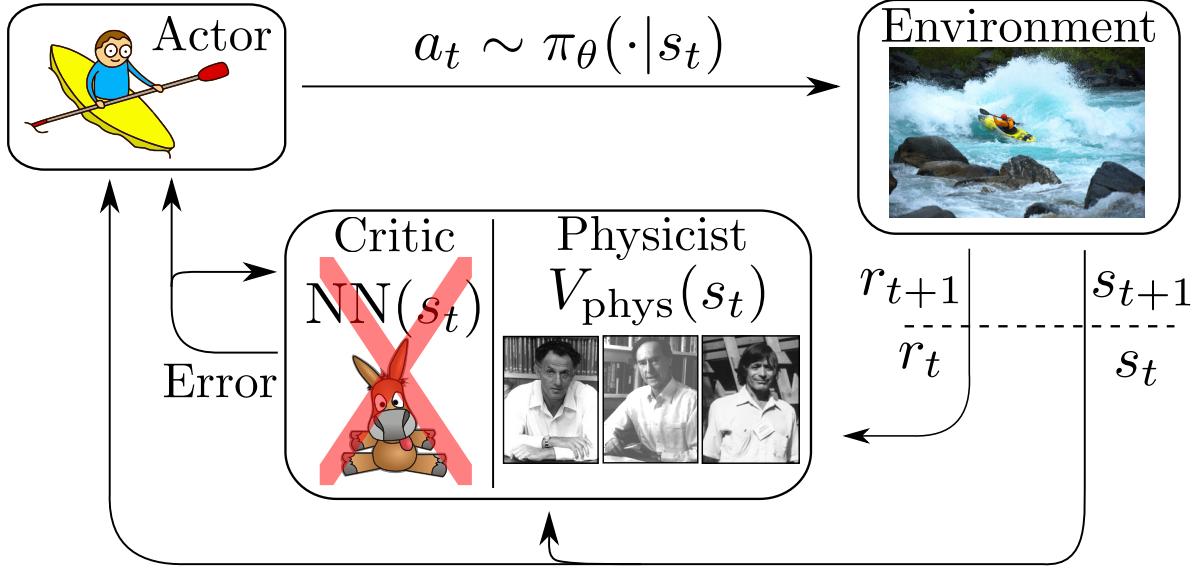


Figure 9.3: Actor-Physicist (AP) diagram from [88], illustrating the main idea: substituting a physics-derived expression in place of the standard neural network-based critic.

The primary motivation arises from the challenge of controlling the proximity of an actively swimming particle to a passive counterpart in turbulent environments, frequently encountered in natural and engineered systems. The governing equation for the separation vector \mathbf{s} between active and passive particles in a turbulent flow is modeled by the following stochastic differential equation (SDE):

$$\frac{d\mathbf{s}}{dt} - \mathbf{v}(t, \mathbf{s}) = \frac{-\mathbf{a}(t) + \boldsymbol{\xi}(t)}{\tau},$$

where τ is a frictional coefficient, $\mathbf{a}(t)$ is the control force exerted by the active particle, and $\boldsymbol{\xi}(t)$ is a stochastic term representing environmental noise.

The AP algorithm, illustrated schematically in Fig. (9.4) from [88], integrates physical insights derived analytically from stochastic optimal control (SOC) theory into the RL framework. The insights specifically leverage studies of idealized stochastic Batchelor-Kraichnan flows.

Next, let us briefly revisit the actor-critic framework of RL, adapting notation consistent with previous sections of the book.

Actor-Critique Algorithm

As already mentioned above RL is a data-driven version of Stochastic Optimal Control (SOC) which in its continuous time formulation with finite time horizon (episode) can be stated as follows:

$$\max_{\pi} \int_0^T dt e^{-\nu t} \mathbb{E}_{a \sim \pi} [r(s(t), a(t))], \quad (9.16)$$

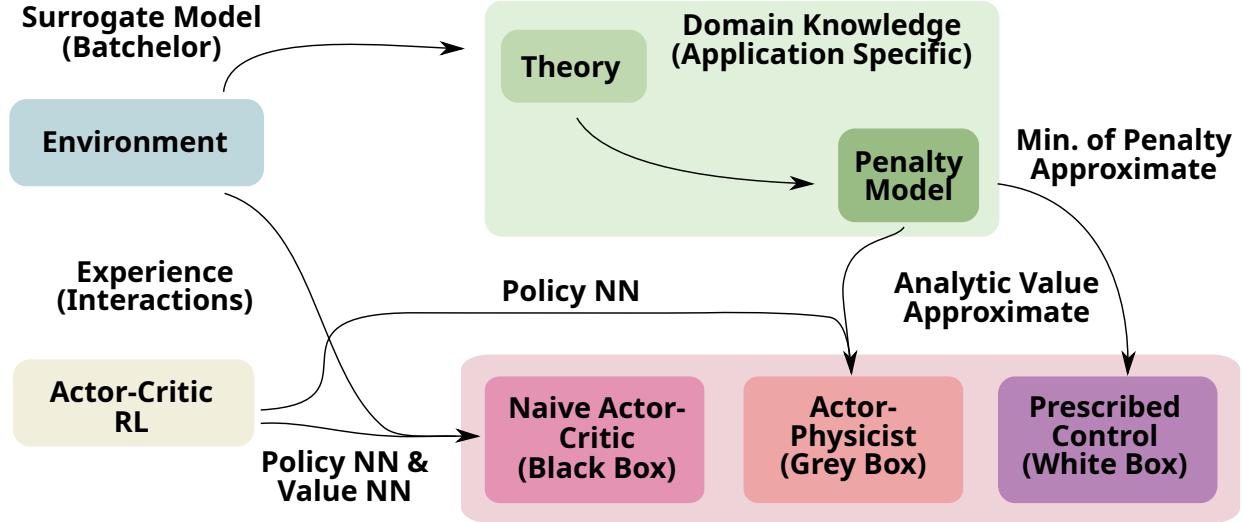


Figure 9.4: Flowchart from [88] illustrating interactions among RL, the environment, theoretical turbulence phenomenology, and different control scheme components.

where the $s(t)$ and $a(t)$ are state observed and action taken according to the policy π at the moment of time t , ν is the discount factor that tells the agent whether to prioritize short-term or long-term rewards, and $r(s(t), a(t))$ is the reward encouraging or penalizing actions of the agents.

In RL, which should be considered as a data driven version of SOC, an agent interacts with an environment, which evolves in discrete time steps. Let us now explain SOC (9.16) on its discrete time RL version. Both in SOC and RL the environment is uncertain, and the next state s' is sampled based on the current state s and the agent's action a : $s' \sim p(\cdot|s, a)$, where the probability distribution is unknown (or at least not fully known). Then Eq. (9.16) turns to

$$\mathcal{G}_\pi(T) = \Delta \mathbb{E}_{\substack{s_0 \rightarrow N \\ a_0 \rightarrow N}} \left[\sum_{n=0}^N \gamma^n r(s_n, a_n) \right], \quad (9.17)$$

where $\pi(a_n|s_n)$ is the policy which defines the probability for the agent to select an action $a_n = a(t_n)$ conditioned to the current state $s_n = s(t_n)$. Here, $t_n = n\Delta = nT/N$, $n = 0, \dots, N \rightarrow \infty$, and $0 < \gamma \leq 1$ is the discrete time version of the discount factor ν in Eq. (9.16). In Eq. (9.17), $s_{n \rightarrow N}$ and $a_{n \rightarrow N}$ are shorthand notations for $(s_{k+1} \sim p(\cdot|s_k, a_k)|k = n, \dots, N-1)$ and $(a_k \sim \pi(\cdot|s_k)|k = n, \dots, N-1)$, respectively.

It is also custom in the field to define the state-action and state value functions:

$$Q_\pi(s_n, a_n) = \Delta \mathbb{E}_{\substack{s_{n \rightarrow N} \\ a_{n \rightarrow N}}} \left[\sum_{k=n}^N \gamma^{k-n} r(s_k, a_k) \mid s_n, a_n \right], \quad (9.18)$$

$$V_\pi(s_n) = \mathbb{E}_{a_n \sim \pi(\cdot|s_n)} [Q_\pi(s_n, a_n)], \quad (9.19)$$

which describe the expected reward accumulated from the current time $t_n = nT/N$ until the end of the episode, under the given policy and conditioned on the current state and action

(s_n and a_n in Eq. (9.18)) and the current state (s_n in Eq. (9.19)).

A classic RL approach derives recursive equations for the value functions under a given policy, followed by policy optimization in a greedy, dynamic programming manner.

A policy gradient-based method, that focuses on maximizing the value function over the policy, was adopted in [88]. As now custom (at the age of AI) the policy function is parameterized via a Neural Network (NN) as $\pi_\theta(\cdot)$, where θ represents the parameters of the NN. The objective is to find the maximum of $V_{\pi_\theta}(s)$ by evaluating its gradient with respect to θ and iteratively updating θ to make the gradient zero in the limit:

$$\mathbb{E}_{a \sim \pi_\theta(\cdot|s)} [Q_\pi(s, a) \nabla_\theta \log \pi_\theta(a|s)] = 0. \quad (9.20)$$

However, this method tends to have high variance, resulting in slow convergence and unreliable estimates. To address this issue, and as now custom in basic RL – we replace the state-action value function in Eq. (9.20) with the so-called *advantage function* [89]:

$$A_\pi(s, a) = Q_\pi(s, a) - b(s),$$

where the *baseline* $b(s)$ depends only on the state and not on the action. This ensures that:

$$\mathbb{E}_{a \sim \pi_\theta(\cdot|s)} [A_\pi(s, a) \nabla_\theta \log \pi_\theta(a|s)] = 0, \quad (9.21)$$

assuming Eq. (9.20) is valid, due to the fact that $\mathbb{E}_{a \sim \pi_\theta(\cdot|s)} [\nabla_\theta \log \pi_\theta(a|s)] = 0$ for any s . Although $b(s)$ can be any function of s , a common choice is $b(s) = V_\pi(s)$.

This modification – from Q to A with $b = V$ – results in the widely-used *actor-critic* methods, where in addition to the policy function (the *actor*), the state-value function $V(\cdot)$ (the *critic*) is also approximated by a NN.

This actor-critic modification of the vanilla policy gradient (where only an actor is present, without a critic) raises the question: how should we interpret this baseline? Formally, it is a degree of freedom used to reduce the variance in the value function's gradient. Informally, the baseline serves as a benchmark for comparison.

Though the use of V_π as a baseline is powerful, it has its limitations. The main drawback is that the training objective changes significantly with each NN update, resulting in continued variations in the gradient estimates.

Physics-Informed Actor-Critic

The proposal of [88] was to use a physics-informed remedy – the concept of the baseline/critic/benchmark was elevated by using a physically derived estimate for the state-dependent value function, rather than relying on a NN as in standard physics-agnostic actor-critic RL approaches. As demonstrated in [88] under certain simplifying assumptions about system dynamics and control, an explicit analytical expression for the baseline, as a function of the state, can be derived for a pair of particles placed in a large-scale turbulent flow.

Table 9.1 presents the mapping of general RL notations to the specific problem of controlling the relative separation between two particles. In this context, the agent is represented by the active particle, whose state is the separation distance from the passive particle. The action corresponds to the swimming efforts of the active particle, while the transition probability models the stochastic evolution influenced by relative velocity and Brownian forces.

General RL	PIRL for Particles
Agent	Active particle
State, s	Separation between active and passive particles, \mathbf{s}
Action, $a \sim \pi(\cdot s)$	Swimming efforts of the active swimmer
Transition probability $p(s' s, a)$	Stochastic evolution according to relative turbulent velocity between swimmers and Brownian force.
Baseline, $b(s)$	State value function, $V_\phi(\mathbf{s})$, estimated following assumptions of the theory.

Table 9.1: Mapping of general RL notations to the specific problem of controlling the relative separation of two particles.

In conclusion – a comparative numerical study conducted in [88] validates the effectiveness of the AP approach against traditional AC strategies (Advantage Actor-Critic (A2C) and Proximal Policy Optimization (PPO)) and a baseline proportional control strategy. The comparison highlights the superior performance of the AP algorithm in environments modeled by synthetic Batchelor-Kraichnan and realistic Arnold-Beltrami-Childress turbulent flows. The superiority of AP is attributed to the physicist’s explicit knowledge of flow dynamics encapsulated in the critic’s heuristic function.

Exercise 9.4.2 (1D swimmer). Consider a simplified 1D stochastic model:

$$\frac{ds}{dt} = -\gamma s + a(t) + \xi(t).$$

Suggest a tractable cost-to-go which introduces a plausible stochastic model for flows in the uncertain/stochastic environment. Explicitly derive and simulate the AP optimal control $a^*(t)$, comparing its performance with standard RL methods. Explore robustness to parameter variations.

As a brief summary – we have introduced MDPs, connected them to RL and illustrated how AI, and specifically illustrated how NN-based function approximation, can reinforce physics-informed exploration of the environment.

Future Challenges: Despite the early advantages – demonstrated in [88] of the physics-informed RL approach – several exciting and challenging research directions remain open:

- Generalization of the Physics Informed Reinforcement Learning methodology to more complex and non-linear flow environments.
- Development of adaptive physics heuristics capable of updating based on real-time data from the environment.

- Investigation of multi-agent control scenarios, where interactions among multiple actively controlled particles are governed by collective physics-informed policies.
- Integration of more sophisticated stochastic optimal control solutions within the physicist (critic) component to enhance the predictive accuracy of the AP algorithm.

In the next subsection, we start to explore how Reinforcement Learning (RL) and Stochastic Optimal Control (SOC) can be systematically integrated into the framework of Generative AI, and vice versa.

9.4.4 RL and Generative AI

In *Generative AI* – particularly under the umbrella of diffusion modeling – Reinforcement Learning (RL) can be embedded to achieve higher-level objectives or to incorporate user feedback. This turns generative modeling into a sequential decision-making process. Conversely, generative models can enhance RL by synthesizing instructive states, trajectories, or even entire environments.

These two directions – **RL → better generation** and **generation → better RL** – form a rapidly evolving feedback loop at the forefront of research. Below, we present two didactic mini-projects that illustrate each direction on a small, manageable scale. Readers are encouraged to implement these exercises. The section concludes with a curated list of state-of-the-art references in the *Further Reading* paragraph.

Exercise 9.4.3 (Reinforcement Learning for Better Energy-Model Sampling). **Motivation.** Suppose we wish to sample low-energy states from a Boltzmann distribution $p(s) \propto \exp(-E(s))$ over discrete configurations s . Naïve MCMC techniques—such as Metropolis–Hastings or Gibbs sampling—can struggle with multimodal distributions due to long trapping times in local minima. An RL agent that learns to actively choose spin flips can potentially accelerate convergence.

Setup. Consider an Ising model with n spins (see Section 7.5 for details).

RL formulation.

- States: spin configuration $s \in \{-1, +1\}^n$.
- Actions: flip one of the n spins.
- Transitions: deterministic, $s \mapsto s'$.
- Reward: $r = -E(s')$; larger reward for lower energy.

With small $n \leq 6$, a tabular Q-learner suffices. Compare its efficiency with Metropolis sampling under identical wall-clock constraints.

1. **Q-learning.** Implement the update:

$$Q(s, a) \leftarrow Q(s, a) + \eta \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right],$$

and monitor convergence of the lowest energy found.

2. **Baseline.** Plot energy histograms from a random-flip Metropolis sampler run for the same number of steps.
3. **Optional:** add a Gibbs sampler for further comparison.
4. **Analysis.** Does the learned policy become deterministic? How does performance scale with n ?

Exercise 9.4.4 (Generative AI for Curriculum Maze Creation in RL). **Motivation.** Let us now reverse the interaction: instead of improving a generator with RL, we use a generative model to create a dynamic curriculum of tasks tailored to the agent’s skill level. The canonical test-bed is an 8×8 grid-world maze.

Environment.

- Generator:
 - (a) **CNN-based Sampler:** a lightweight convolutional net maps uniform noise to a binary solvable maze. Fast but deterministic.
 - (b) **Diffusion-based Generator:** treat the maze as an 8×8 binary image. A denoising model θ_t performs T reverse steps starting from i.i.d. Bernoulli noise, mimicking the diffusion generation process.
- RL episode: the agent is placed in a start cell; reward is +1 at goal and -0.01 per step otherwise.

Curriculum Loop. Every N_c episodes, the generator is updated based on the agent’s recent success rate. If the agent struggles, easier mazes are produced (e.g., fewer junctions). If it performs well, harder mazes are synthesized. In the diffusion variant, this curriculum is implemented by conditioning the denoising network θ_t on a difficulty label d_t .

Notation. We use the term **DQN (Deep Q-Network)** for the neural approximation $Q_\phi(s, a)$ of the Q -function.

1. **Fixed Maze Baseline:** Train DQN on a single static maze; log episode length and return over training.
2. **Random Maze:** Train DQN using fresh samples from the CNN-based maze generator.
3. **Diffusion Curriculum:** Replace the CNN sampler with the diffusion model, conditioned on the difficulty schedule.
4. **Evaluation:** Freeze the trained DQN and test on 100 unseen hard mazes. Compare performance across all training regimes.

Further Reading

RL → Better Generation. The energy-model exercise above showcases a toy application of RL within a generative process. But RL’s role in generative modeling – especially in diffusion – is deeper. One notable example is *Denoising Diffusion Policy Optimization (DDPO)* introduced in [90], which re-frames denoising in diffusion models as a sequential decision-making task.

Recent advances along this line include:

- [91]: scales DDPO to millions of images and multi-objective settings (e.g., fairness, alignment).
- [92]: adapts DDPO to one-step *consistency models*, enabling rapid inference after RL-style fine-tuning.
- [93]: introduces *D3PO* (Direct Preference Policy Optimization), which avoids training a separate reward model by directly fine-tuning from human feedback.

These approaches suggest that RL-guided fine-tuning of generative models may offer robust, interpretable control over outputs – beyond heuristic prompt engineering.

Generation → Better RL. The maze exercise, though toyish, hints at a more general role of diffusion as a data synthesizer in RL. A broader review is offered in [94], which discusses:

- *Diffusion as a Planner*: generative diffusion models produce action plans via autoregressive view of the task.
- *Diffusion Policies*: models directly parameterize the agent’s policy via diffusion mechanisms, replacing conventional policy nets.
- *Curriculum Learning*: task generation adapts over time, as in our maze example, to match agent progress – a concept also used in reward shaping and active task sampling.

The interplay between data synthesis and decision making in such settings is an exciting frontier for both Generative AI and Control.

9.5 Synthesis of Diffusion and Reinforcement Learning: Path Integral Diffusion

Why fuse diffusion with RL? The preceding section illustrated *two* complementary directions of recent research: (i) how score-based diffusion models benefit from Reinforcement Learning (RL) ideas and (ii) how RL itself adopts continuous time stochastic-dynamical viewpoints that first appeared in diffusion-based generative modeling. The natural next step – developed in this section – is to *erase the boundary altogether* and view the two paradigms through a single optimal-control lens. The resulting framework is called *Path-Integral Diffusion (PID)* [65] and can be read simultaneously as an *integrable* subclass of Stochastic Optimal Control (SOC), and a diffusion model whose score (and hence sampler) admits a *closed-form expression*, removing the need for a Neural Network (NN) *score-function oracle* (at least in some settings).

9.5.1 A short pre-history of “integrable” SOC

The idea that some SOC problems become *linear* after a non-linear change of variables goes back to the work of Fleming [95] and Mitter [96] (early 1980s) on the Hopf–Cole transformation of the Hamilton–Jacobi–Bellman (HJB) equation of SOC. Kappen’s *Path-Integral Control* (PIC) [97] showed that when the control cost is quadratic, space-dependent term in the control’s cost function is arbitrary and the control enters the dynamics additively and with the same covariance as the noise, the HJB reduces to a backward heat equation in a potential (which can also be viewed as a Schrödinger equation in imaginary time) whose fundamental solution (i.e. Green function) fully characterizes the optimal policy. Todorov then generalized the construction to discrete time/space, introducing *Linearly-Solvable MDPs* (LS-MDPs) [98]. In generative modeling, Tzen and Raginsky [99] employed the simplest case of the integrable SOC structure (the PIC structure without potential, known from Fleming [95]) to prove convergence guarantees for *latent diffusions* [99].

9.5.2 From SOC to Path-Integral Diffusion (PID)

We follow the recent formulation of [65] which augments Path-Integral Control (PIC), which has already contained an *external potential* $V(t, \mathbf{x})$, with

- An *arbitrary* drift $\mathbf{f}(t, \mathbf{x})$,
- And a *vector potential* $\mathbf{A}(t, \mathbf{x})$ (a “gauge field”).

The stochastic dynamics is

$$t \in [0, 1] : \quad d\mathbf{x}_t = \mathbf{f}(t; \mathbf{x}_t + \mathbf{u}(t; \mathbf{x}_t)dt + d\mathbf{W}_t, \quad \mathbf{x}(0) = \mathbf{0}, \quad (9.22)$$

where we introduce the control vector field \mathbf{u} to enforce that the (probability distribution) density at $t = 1$ fits the target distribution:

$$t = 1 : \quad p(\mathbf{x}_1) = p_{\text{target}}(\mathbf{x}_1). \quad (9.23)$$

It was shown in [65] that the goal can be achieved in many ways. Moreover, the desired controlled solution can be framed as resolving the following SOC formulation:

$$\min_{\mathbf{u}(\cdot; \mathbf{x}(\cdot))} J(t = 0, \mathbf{x}_{t=0}), \quad (9.24)$$

$$\begin{aligned} \tau \in [0, 1] : \quad J(\tau; \mathbf{x}_\tau) := \mathbb{E} \Big[\int_\tau^1 dt \left(\frac{|\mathbf{u}(t; \mathbf{x}_t)|^2}{2} + V(t; \mathbf{x}_t) + \dot{\mathbf{x}}_t \mathbf{A}(t; \mathbf{x}_t) \right) \\ + \phi(\mathbf{x}_1) \Big| \text{Eqs. (9.22,9.23)} \Big], \end{aligned} \quad (9.25)$$

where ϕ -function defines the terminal cost consistent with the terminal (optimal transport) condition, set by Eq. (9.23); and the J -function is the so-called cost-go, which satisfies the Hamilton–Jacobi–Bellman (HJB) equation (which is just a continuous-time-space version of the Bellman Eq. (9.15) we have encountered earlier in the Markov Decision Process setting):

$$t \in [1 \rightarrow 0] : \quad -\partial_t J = V + \frac{1}{2} (\nabla^T (\nabla J + \mathbf{A}) - |\nabla J + \mathbf{A}|^2) + \mathbf{f}^T (\nabla J + \mathbf{A}), \quad (9.26)$$

which we solve backwards in time, assuming that $J(t = 1, \mathbf{x}_{t=1}) = \phi(\mathbf{x}_1)$. Once the HJB Eq. (9.26) we can then use the resulting cost-to-go function, J to find the optimal control

$$t \in [0, 1] : \quad \mathbf{u}^*(t; \mathbf{x}_t) = -\nabla_{\mathbf{x}} J(t, \mathbf{x}_t) - \mathbf{A}(t, \mathbf{x}_t). \quad (9.27)$$

Applying the Hopf–Cole substitution – $J(t; \mathbf{x}) = -\log \psi(t; \mathbf{x})$ – transforms the nonlinear HJB Eq. (9.26) into a *linear* backward Kolmogorov equation

$$\begin{aligned} -\partial_t \psi + \tilde{V} \psi + \tilde{\mathbf{A}}^T \nabla \psi &= \frac{1}{2} \Delta \psi, \quad \psi(1; \mathbf{x}) = \exp(-\phi(\mathbf{x})), \\ \tilde{V} &:= V + \frac{1}{2} \nabla^T \mathbf{A} + \mathbf{f}^T \mathbf{A} - \frac{1}{2} |\mathbf{A}|^2, \quad \tilde{\mathbf{A}} := \mathbf{A} - \mathbf{f}, \end{aligned} \quad (9.28)$$

and then the forward Kolmogorov-Fokker-Planck equation for the optimal (probability distribution) density is

$$\partial_t p^* + \nabla^T \left(p^* (\nabla \log \psi - \tilde{\mathbf{A}}) \right) = \frac{1}{2} \Delta p^*, \quad p^*(0; \mathbf{x}) = \delta(\mathbf{x}), \quad p^*(1; \mathbf{x}) = p_{\text{target}}(\mathbf{x}). \quad (9.29)$$

The resulting expression for the optimal control vector field becomes

$$\mathbf{u}^*(t; \mathbf{x}_t) = \nabla_{\mathbf{x}} \log \left(\int d\mathbf{y} \, p_{\text{target}}(\mathbf{y}) \frac{G_-(t; \mathbf{x}_t; \mathbf{y})}{G_+(1; \mathbf{y}; \mathbf{0})} \right), \quad (9.30)$$

where G_{\mp} -functions are the Green functions of the backward and forward Kolmogorov equations:

$$t \in [1 \rightarrow 0] : \quad -\partial_t G_- + \tilde{V}(\mathbf{x}; t) G_- + \tilde{\mathbf{A}}^T \nabla G_- = \frac{1}{2} \Delta G_-, \quad G_-(1; \mathbf{x}) = \delta(\mathbf{x} - \mathbf{y}), \quad (9.31)$$

$$t \in [0 \rightarrow 1] : \quad \partial_t G_+ + \tilde{V}(\mathbf{x}; t) G_+ - \nabla^T (\tilde{\mathbf{A}} G_+) = \frac{1}{2} \Delta G_+, \quad G_+(0; \mathbf{x}) = \delta(\mathbf{x} - \mathbf{y}). \quad (9.32)$$

Therefore, Eq. (9.30) expresses the control – which in the diffusion interpretation is nothing but the *score function* – as a convolution of p_{target} with a kernel expressed via the Green functions.

9.5.3 Three levels of integrability

(i) Top level: arbitrary fields. All terms $(V, \mathbf{f}, \mathbf{A})$ are retained; assuming the Green functions can be found, sampling reduced to the SDE (9.22) – with the control field, $\mathbf{u}(\bullet; \bullet)$ substituted by its optimal value, $\mathbf{u}^*(\bullet; \bullet)$, governed by Eq. (9.30). Moreover the sampling requires *no NN*.

(ii) Mid level: Harmonic PID. If V is *quadratic* in \mathbf{x} and both \mathbf{f} and \mathbf{A} are *affine* in \mathbf{x} , then the Green functions, G^{\pm} , are multivariate Gaussians. The optimal control becomes

$$\mathbf{u}^*(t, \mathbf{x}) = a(t)\mathbf{x} - b(t)\hat{\mathbf{x}}(t; \mathbf{x}), \quad \hat{\mathbf{x}}(t; \mathbf{x}) := \mathbb{E}_{\mathbf{y} \sim w(\bullet|t; \mathbf{x})} [\mathbf{y}], \quad w(\mathbf{y}|t; \mathbf{x}) \propto p_{\text{target}}(\mathbf{y}) \frac{G_-(t; \mathbf{x}; \mathbf{y})}{G_+(1; \mathbf{y}; \mathbf{0})},$$

where the *weight* $w(\mathbf{y}|t, \mathbf{x}) \propto p_{\text{target}}(\mathbf{y}) \frac{G_-(t, \mathbf{x}; \mathbf{y})}{G_+(1, \mathbf{y}; 0)}$ is a bona-fide probability density.

(iii) **Low level: uniform quadratic potential.** Setting $f = \mathbf{A} = \mathbf{0}$ and $V(\mathbf{x}) = \beta\|\mathbf{x}\|^2/2$ yields the *Harmonic PID* (H-PID) sampler implemented in [65]. Analytic expressions for the Green functions and for \mathbf{u}^* follow from elementary Gaussian identity manipulation.

Example 9.5.1 (Harmonic PID sampler). For $V(\mathbf{x}) = \beta\|\mathbf{x}\|^2/2$ we have

$$\begin{aligned} u^*(t; \mathbf{x}) &= \frac{\sqrt{\beta}}{\sinh((1-t)\sqrt{\beta})} \left(\hat{\mathbf{x}}(t; \mathbf{x}) - \mathbf{x} \cosh((1-t)\sqrt{\beta}) \right), \\ \frac{G_-(t; \mathbf{x}; \mathbf{y})}{G_+(1; \mathbf{y}; \mathbf{0})} &= \sqrt{\frac{\sinh(\sqrt{\beta})}{\sinh((1-t)\sqrt{\beta})}} \exp \left(-\frac{\sqrt{\beta}}{2} \left((\mathbf{x}^2 + \mathbf{y}^2) \coth((1-t)\sqrt{\beta}) \right. \right. \\ &\quad \left. \left. - \mathbf{y}^2 \coth(\sqrt{\beta}) - \frac{2(\mathbf{x}^T \mathbf{y})}{\sinh((1-t)\sqrt{\beta})} \right) \right). \end{aligned}$$

It was reported in [65] that the random vector $\hat{\mathbf{x}}$ plays the role of an order parameter: as β crosses the critical value the PID dynamics undergoes a dynamic phase transition – akin to one discussed in Section 9.3.

Exercise 9.5.1 (Harmonic PID and dynamic phase transition).

Fix a one-dimensional two-modal target density $p_{\text{target}}(y) = \frac{1}{2}[\mathcal{N}(y|-\mu, \sigma^2) + \mathcal{N}(y|\mu, \sigma^2)]$ and analyze H-PID dynamics at different β .

9.5.4 PID viewed from Reinforcement Learning

PID may be viewed as offering via the potential V and the vector potential \mathbf{A} terms a closed-form “critic” while the *actor* becomes the deterministic mapping $\mathbf{u}^*(t, \mathbf{x})$. The PID-learning reduces to fixing critic and estimating (or approximating) the optimal control (score function).

Open question: It may be interesting to analyze if and how standard RL algorithms can be improved/accelerated – possibly providing physics/application guided prior in terms of the scalar and vector potentials, V and \mathbf{A} and the drift term f .

Take-aways

- **One equation to rule them all.** A Hopf–Cole transform linearises the HJB; the resulting Green functions encode both diffusion models and RL policies.
- **Three flavors of exact solvability.** “Top” (general linear PDEs), “mid” (Gaussian kernels) and “low” (closed analytic sampler) tiers allow a smooth trade-off between realism and mathematical tractability.
- **Neural-network-free score.** In PID the score is an explicit convolution, providing a rare example of a *provably convergent, provably unique* diffusion sampler without a trainable NN.

- **Bridge to RL and Transformers.** PID aligns continuous-time diffusion with RL, while discrete LS-MDPs underlie *Generative Flow Networks*, thereby placing diffusion, RL and auto-regressive Transformers *under one umbrella* – and this is the topic we are about to discuss in the next section.

9.6 Diffusion, Reinforcement Learning and Transformers under one Umbrella: *Sampling Decisions*

9.6.1 From *Artificial* to *Physical* Time

The continuous *Path–Integral Diffusion* (PID) formulation interprets a score-based diffusion model as an *integrable* Schrödinger bridge: a Stochastic Optimal Control (SOC) problem defined on the artificial “noise–scale” $\tau \in [0, 1]$ that smoothly transports the point-source $x(0) = 0$ to a Gibbs target p_{target} at $\tau = 1$ [65]. The parameter τ , however, is purely algorithmic — it is not the clock that governs the *physical* assembly of the sample.

Decision Flow (DF) re-establishes such a physical chronology by (a) discretizing time into T *growth steps*, and (b) equipping each step with an explicit action that adds exactly one new degree of freedom to the partial configuration. In doing so, DF inherits PID’s linear solvability while working in genuine, causal time, exposing the auto-regressive structure familiar from transformers.

9.6.2 Generative Flow Networks in a Nutshell

A *Generative Flow Network* (GFN) [66, 100] produces samples by successively *growing* an object, rather than drawing it in one shot. Let $\sigma = (\sigma_{a_1}, \dots, \sigma_{a_T}) \in \mathcal{X}$ be the final object we wish to draw with probability $\pi_T(\sigma) \propto R(\sigma) = \exp(-E(\sigma))$. A GFN introduces a growth trajectory

$$s_0 = \emptyset \rightarrow s_1 \rightarrow \dots \rightarrow s_T = \sigma, \quad s_t = (a_1, \dots, a_t) \quad (9.33)$$

where s_t records the first t moves, with size gradually increasing in t — like starting from a white image and coloring it pixel-by-pixel, such that $s_t \in \mathcal{X}_t$ and $\mathcal{X}_0 = \emptyset \subset \mathcal{X}_1 \subset \dots \subset \mathcal{X}_T = \mathcal{X}$. A learned *flow* $F_t(s_{t+1} | s_t)$ must satisfy the *trajectory balance* identity so that marginalizing the trajectories recovers π_T . Writing $R(\sigma) = \exp[-E(\sigma)]$ and $Z = \sum_\sigma R(\sigma)$, the constraint reads

$$\prod_{t=0}^{T-1} F_t(s_{t+1} | s_t) = \frac{1}{Z} R(s_T) \prod_{t=1}^T F_{t-1}^\leftarrow(s_{t-1} | s_t), \quad (9.34)$$

where F^\leftarrow denotes the corresponding reverse flow. Eq. (9.34) — *Trajectory Balance* (TB) — generalizes the familiar *Detailed Balance* (DB) of Markov chains and replaces the more restrictive *Global Balance* (GB) condition historically used in Monte Carlo methods (see Sections 7.4, 7.5).

GFN training amounts to minimizing a TB-inspired loss such as

$$\left(\log \frac{Z_\theta}{R(s_T)} + \sum_{t=0}^{T-1} \log \frac{F_t(s_{t+1} | s_t; \theta)}{F_t^\leftarrow(s_t | s_{t+1}; \theta)} \right)^2,$$

over the parameterized flow F_θ toward the exact solution F^* satisfying (9.34). In practice, $\log F_\theta$ is typically parameterized by a neural network trained via stochastic optimization.

Significance for Decision Flow (DF) (coming next). GFN supplies a principled, discrete-time recipe for *auto-regressive sampling*. DF retains this construction but replaces the black-box network by a *linearly-solvable control law* — yielding analytic kernels, transparent explanation, and a link to Markov Decision Processes (MDPs) discussed earlier. In that sense, DF can be viewed as an *integrable* white-box extension of the GFN framework, also injecting diffusion modeling into the GFN formalism.

9.6.3 Decision Flow: an Integrable, Diffusion-like Extension of GFN

We continue our exploration of sampling from the energy-based distribution

$$\sigma \sim \pi_T(\sigma) \propto R(\sigma) = \exp(-E(\sigma)), \quad (9.35)$$

that is, sampling from a model known only up to a normalization constant. As with GFN, we avoid generating σ in one shot and instead construct it *sequentially* via Eq. (9.33). However, diverging from GFN, we assume access to a *prior* Markov kernel $p_t^{\text{prior}}(s_{t+1} | s_t)$ that guarantees accessibility of all finite-energy states. The objective is to learn a controlled kernel $p_t(\bullet | \bullet)$ that steers the growth process toward the Gibbs terminal distribution (9.35) while deviating as little as possible from the prior.

Linearly-Solvable MDP: Formulation. Decision Flow (DF) [67] casts this problem as a *linearly solvable Markov decision process* (LS-MDP) [98, 67]. Among all path measures with 1-step transitions $p_t(\bullet | \bullet)$ and marginals $\pi_t(\bullet)$, we minimize the Stochastic Optimal Control (SOC) functional:

$$\sum_{t=0}^{T-1} \mathbb{E}_{\pi_t(\bullet); p_t(\bullet | \bullet)} \left[\log \frac{p_t(s_{t+1} | s_t)}{p_t^{\text{prior}}(s_{t+1} | s_t)} \right] + \mathbb{E}_{\pi_T(\bullet)} [\Phi(s_T)], \quad (9.36)$$

over $p_t(\bullet | \bullet)$ and $\pi_t(\bullet)$, subject to the recursion

$$\pi_{t+1}(s_{t+1}) = \sum_{s_t} p_t(s_{t+1} | s_t) \pi_t(s_t), \quad (9.37)$$

and initialized by the empty state $s_0 = \emptyset$. The first term in Eq. (9.36) penalizes deviations from the prior, while the second term encourages agreement with the Gibbs target (9.35).

Closed-form Optimal Policy. Define the *backward Green's function*:

$$G_t(s_t | \sigma) := \sum_{s_{t+1:T-1}} \prod_{\tau=t}^{T-1} p_\tau^{\text{prior}}(s_{\tau+1} | s_\tau) \cdot \mathbf{1}\{s_T = \sigma\},$$

which satisfies $G_T(\sigma \mid \sigma') = \delta_{\sigma\sigma'}$. Define also the scalar potential:

$$u_t(s_t) := \sum_{\sigma} e^{-E(\sigma)} G_t(s_t \mid \sigma). \quad (9.38)$$

Remarkably, the optimal kernel minimizing Eq. (9.36) under constraints (9.35), (9.37) is:

$$p_t^*(s_{t+1} \mid s_t) = \frac{p_t^{\text{prior}}(s_{t+1} \mid s_t) u_{t+1}(s_{t+1})}{u_t(s_t)}. \quad (9.39)$$

Exercise 9.6.1. Linear Solvability. Augment the cost (9.36) with Lagrange multipliers enforcing (9.35), (9.37). Derive the stationarity conditions by varying over $p_t(\bullet \mid \bullet)$ and $\pi_t(\bullet)$, and verify that substituting (9.39) yields the following backward recurrence:

$$u_T(s_T) = e^{-E(s_T)}, \quad u_t(s_t) = \sum_{s_{t+1}} p_t^{\text{prior}}(s_{t+1} \mid s_t) u_{t+1}(s_{t+1}). \quad (9.40)$$

Hint: See Feature #2 below for the connection between $\Phi(\cdot)$ and $E(\cdot)$.

Notable Features of the Decision Flow Framework

Decision Flow possesses several important properties:

1. **Integrability:** The MDP formulation leads to Bellman-like equations that are linear and solvable in closed form, as demonstrated by Eq. (9.39). No numerical dynamic programming is needed. (Note: Eq. (9.39) is the discrete analog of the Hopf–Cole transformation familiar from Path Integral Control, see Section 9.5.)
2. **Relation to Optimal Transport:** By choosing the terminal potential $\Phi(\cdot)$ in Eq. (9.36) to match the prescribed Gibbs form (9.35), the DF formulation also solves the associated optimal transport problem — mapping the trivial initial distribution (a δ -function on the empty set) to the energy-based target.
3. **Algorithmic Recipe:** The DF sampler consists of:
 - (a) **Backward pass** (compute the potential): solve Eq. (9.40).
 - (b) **Forward pass** (generate a sample): draw $s_{t+1} \sim p_t^*(\cdot \mid s_t)$ via Eq. (9.39).

This yields an *exact* sampler once u is known, and empirical convergence is typically rapid due to linearity.

4. **Neural Implementation:** The scalar potential u_t can be parameterized by a NN to enable fast, regularized inference.
5. **Connections to Diffusions and GFNs:** DF unifies:
 - (i) Discrete-time, discrete-space linearly solvable control [98],
 - (ii) Continuous-time, continuous-space path integral control [97], and

- (iii) Generative Flow Networks [100].

The closed-form kernel (9.39) enables principled use of prior dynamics while ensuring exact energy-based sampling.

6. **State Space Enlargement, Markovianity, and Transformers:** While the physical state σ is non-Markovian, the extended trajectory $\{s_t\}$ forms a Markov chain. This mirrors the **auto-regressive** structure in transformers, where the expanding context renders the next-token distribution conditionally independent of the distant past.
7. **DF and Reinforcement Learning (RL):** DF’s construction explicitly formulates an MDP, directly connecting to RL — viewed as a data-driven instantiation. Beyond this, DF aligns with recent RL-based post-training and inference-time alignment methods in diffusion models, see [101] for review. These approaches modify generation to meet new downstream tasks. While RL post-training relies on standard RL algorithms, inference-time alignment resembles DF in that the adjustment is at least partially analytic.

Small Illustrative Example from [67]: Ising 3×3

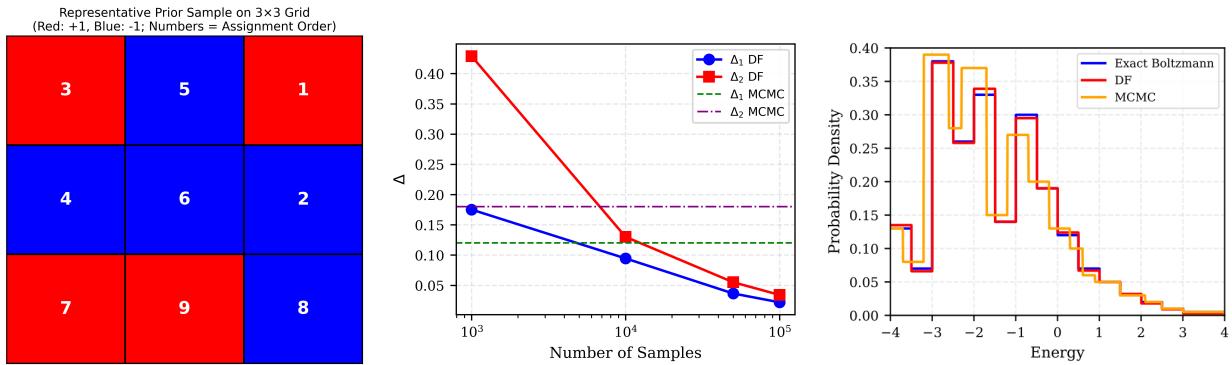


Figure 9.5: **Left:** Typical Decision Flow trajectories on a 3×3 Ising lattice at $\beta = 0.6$ (red/blue = ± 1). Numbers indicate insertion order. **Middle:** Performance of the DF algorithm on a 3×3 planar Ising model. Each bias h_a and interaction J_{ab} is independently drawn from Uniform $[-1, 1]$, and the number of prior samples K equals the number of posterior samples S . The plot displays the discrepancy metrics $\Delta_{1,2}$ using the exact reference, as a function of $S = K$. For comparison, dashed horizontal lines indicate results from Metropolis-Hastings MCMC, where the chain discards the first 2000 burn-in samples and then records every 10th sample from a total of 5000 samples. **Right:** Probability density function of the energy $E(\sigma)$ estimated from DF-generated samples. The density is computed from σ samples produced using the DF algorithm with $K = S = 5 \times 10^4$.

DF algorithm was tested on a “glassy” Ising model defined over a planar grid. Fig. 9.5 (left) illustrates a representative prior sample. Using K prior samples, the DF algorithm generates S posterior (target) samples. To assess sample quality, empirical estimates of the singleton

and pairwise correlations:

$$m_a^* = \frac{1}{S} \sum_{s=1}^S \sigma_{T;a}^{(s)}, \quad c_{ab}^* = \frac{1}{S} \sum_{s=1}^S \sigma_{T;a}^{(s)} \sigma_{T;b}^{(s)},$$

were computed and compared to reference values – obtained either exactly (as in the 3×3 example) or via MCMC. The integrated mismatch metrics are defined as

$$\Delta_1 = \sum_a \frac{\|m_a - m_a^{(\text{ref})}\|}{T \|m_a^{(\text{ref})}\|}, \quad \Delta_2 = \sum_{(a,b)} \frac{2\|c_{ab} - c_{ab}^{(\text{ref})}\|}{T(T-1) \|c_{ab}^{(\text{ref})}\|}.$$

The results shown in middle and right panels of Fig. 9.5 demonstrate that, with a sufficiently small sample size the DF algorithm outperforms MCMC. Specifically, see the middle panel for the Δ -test results and right panel for the energy probability density function of the generated samples.

Outlook

DF offers a unifying, analytically grounded framework combining *Diffusion*, *Stochastic Optimal Control*, and *Transformers*. Promising future directions include:

- *Scalability – Neural Green’s Functions*: Learn not only transition kernels but also reverse Green’s functions using NNs to enhance generalization.
- *Continuous-space and Hybrid DF*: PID [65] can be viewed as a continuous-time limit of DF with short and finite memory (not auto-regressive). Extending PID to growing state spaces including hybrid continuous-discrete DF for trans-dimensional problems (e.g., jump or birth-death dynamics) — is a natural next step.
- *Transformer-guided DF*: While the auto-regressive nature is already embedded in DF, further integration of transformer components — such as sparse attention — may yield promising new architectures.
- *Expert-informed DF*: The flexibility of DF allows incorporating domain constraints and physics-based priors, targeting rare or underrepresented modes of the distribution. Identifying tractable formulations for such inference-time alignment is an important challenge.

DF has broad application potential in science and engineering. For instance:

- **Molecular design**: Building molecules sequentially.
- **Physical systems**: Control of complex fluids.
- **Engineered systems**: Smart grids, robotics, swarms.
- **Social systems**: Simulation of viral spread, epidemic control.

In each case, the timeline naturally decomposes into auto-regressive episodes — precisely the regime DF is designed for.

9.7 Path Forward

Over the preceding sections of this synthesis chapter we have reviewed three paradigms that at first appear distinct yet are becoming increasingly intertwined in modern generative modeling and decision-making:

1. *Score-based diffusion models* (Sections 9.1–9.3), where samples are obtained by inverting a carefully designed stochastic process;
2. *Path-integral control and reinforcement learning* (Sections 9.4–9.5), whose sampling viewpoint emerges from *optimal control*;
3. *Generative Flow Networks (GFNs)* and *Decision Flow* (Section 9.6), which recast sampling as *decision-making*, i.e. optimization and control of transition probabilities on directed graphs.

A unifying mathematical language is now taking shape – rooted in stochastic optimal control, optimal transport, and the geometry of probability flows – that promises to bridge these paradigms. Constructing a rigorous “dictionary” that links their key objects – score functions, control fields, and flow potentials – remains an outstanding research goal.

9.7.1 Work in Progress (by the author) and Open Directions

While drafting this living book in Spring 2025 I mainly with Hamidreza Behjoo and, for Decision Flow, with Sungsoo Ahn—have continued to explore synergies among these models. A (partial) rainbow of ideas currently under active development is collected below so readers can trace the common thread.

- **U-Turn Diffusion** — building on [65], this minimal modification of score-based diffusion forces trajectories back to the data manifold, enabling faster reverse sampling and offering a dynamical phase-transition interpretation. Key questions:
 - How many phase transitions occur, and how are they linked to latent (possibly discrete) data labels?
 - Which order parameters best capture the transition structure?
 - Can the resulting insights be used to design self-supervised *self-classification* objectives or to regularise diffusion models?
- **Space-Time Bridge Diffusion** — extending [71], this framework employs a Doob transform to create finite-horizon diffusion bridges by *pinning* trajectories at a fixed time. A promising route is to let the forward process stay linear in the state space – to retain analytical tractability – while also become dependent/sensitive to the ground truth data or more generally on an information available about the probability distribution we are aiming to sample from. Outstanding tasks:
 - Identify tractable classes of data-conditioned linear processes (and may be non-linear too);

- Quantify the trade-off between expressivity and computational cost introduced by the pinning constraints.
- **Path–Integral Diffusion** — generalizing [65], this family of *integrable* stochastic-optimal-control (SOC) formulations reduces to a forward–backward pair of linear PDEs yet remains flexible enough to target arbitrary data distributions. Three nested research levels stand out:
 1. *Low-level integrability*: Beyond Harmonic PID case – with quadratic potentials (where integrability mirrors solving a Schrödinger equation in a quadratic well), can we build the scheme on richer families of integrable drift–diffusion processes?
 2. *Mid-level integrability*: Given samples, can neural surrogates (that is one represented by NNs) accurately approximate the reversed Green function and score functions?
 3. *High-level integrability*: How can drifts and vector potentials be tuned to encode additional scientific laws (e.g. conservation, equations, dependencies) or perceptual priors (e.g. aesthetics in images)?
- **Sampling Decisions** — introduced in [67], this framework unifies diffusion, reinforcement learning, and auto-regression. Next milestones:
 - *Domain-specific specialisation*: e.g. structural design in materials or real-time control of engineered systems;
 - *Neuralisation*: replace analytic transition kernels, score functions, and Green functions with neural surrogates while preserving decision-theoretic guarantees;
 - *Transformer enrichment*: weave more expressive transformer components into the decision flow framework, with an emphasis on (sparse) attention mechanisms adapted to the statistics of the data, the underlying energy landscape, or the control objectives. (At present the mathematics of transformers is far less developed than that of diffusion models or RL – but see [102, 103] for notable early progress.)

9.7.2 Further Ideas on a Grand Unification of Generative Models

A number of broad(er) challenges interconnect the topics above:

Scaling Laws: How many samples are needed to guarantee high-quality generation? How does this requirement scale with state-space dimensionality? Can we prove that AI-based samplers outperform classical tools such as MCMC for energy-based models?

Mixing Modalities: Which generative tools – among the expanding portfolio – are preferable for (a) discrete, continuous, or mixed spaces, and (b) energy-function, sample-based, or hybrid representations?

From Samples to Policies: How can Decision Flow be integrated with model-based RL to produce *policies*, not just trajectories or final samples? Viewed differently: can we

fuse Decision Flow with broader SOC/RL frameworks to create a *generative control network*?

Physics-informed Learning: How do we weave phenomenological equations, conservation laws, symmetries, and other domain knowledge directly into the sampling procedures discussed above? (See [104] for examples in turbulence.) The challenge is to embed such features without sacrificing generative flexibility.

Outlook. We hope that unifying these threads will yield a cohesive *calculus of decision flows and physics-informed generative controls*: a toolkit that moves fluidly between data-driven modeling, principled control, and decision-centered sampling.

9.7.3 Downstream Applications: Where the Mathematics of AI Meets the Real World

The preceding sections of the final chapter of this living book showed *how* we can derive, connect, and sometimes even *unify* diffusion, flow-matching, energy-based, and reinforcement-learning viewpoints. Natural final questions are:

What can we actually build with all this? What are the downstream tasks of generative AI?

These questions are particularly pressing today given the widening gap between major AI companies—focused on training massive foundation models—and the rest of the AI community, including smaller companies, startups, and academic groups. With limited resources, the latter naturally pivot toward opportunities in solving *downstream* problems: fine-tuning, composing, aligning, and deploying powerful models that already exist.

Below is a deliberately **non-technical postcard** of opportunities growing directly out of the ideas developed in this chapter. Each bullet can be seen as an invitation to a project, thesis, or startup.

1. Compositional Generation

- **Composition of Experts.** Building new generative models by combining existing ones: products of experts (PoE) for strong constraint satisfaction [105], classifier-free guidance and general conditioning [106], compositional text-to-image generation via product and mixture rules [107]. Also creative tapestry-style image generation by combining regions under different conditional experts [108].
- **Modality stitching.** Combining distinct modalities (e.g., audio with video, graphs with text) into a coherent generation framework via factor graph structures. A notable example is diffusion models for molecules + property captions [109], or 3D Computer-Aided-Design geometry + texture stitching using conditional latent diffusion [110]
- **Editable generation after the fact.** Score-based diffusion offers *time* as a controllable knob: one can rewind to an intermediate noise level, inject a constraint (e.g., a new text caption or region mask), and then roll forward again [111]. This enables rapid “what-if” loops for art, fashion, drug design, and content personalization.

2. Decision-Aware Generation—Merging Diffusion & Control

- **Planning as inference.** The path-integral view (Section 9.5) allows optimal trajectories to emerge as samples from a guided stochastic process. Applications: human-style motion planning in robotics [112]; planning trajectories through a maze (regions with exclusions) [113]; GenAI in supply chain and operations management [114].
- **Data-driven simulators.** Stochastic differential models learned from data can serve as differentiable surrogates for expensive PDE solvers: weather forecasting [115]; chip layout placement [116]; and aerodynamic shape design [117].
- **Risk-sensitive creativity.** Weighting diffusion paths by user-defined cost potentials (as in Sections 9.5 and 9.6) produces either conservative or exploratory behavior by tuning parameters like β . To the best of our knowledge these tools were not yet used but may be useful in portfolio risk management [118], safe reinforcement learning [119], and robust autonomy [120].

3. Scientific Discovery & Inverse Problems

- **Generative surrogates for Bayesian inversion.** Using a diffusion model as a prior and conditioning on partial observations enables fast, amortized Bayesian inversion – applied to MRI image reconstruction [121], seismic tomography [122], and gravitational wave source inference [123].
- **Symbolic hypothesis engines.** Decoding diffusion latents not into pixels but into symbolic quantitative models (expressed via equations) – offers a pathway toward *data-driven scientific discovery*. See [124] for physics-informed, approach and [125] for large language model approach – clearly more to be done with unified GenAI approaches.
- **Uncertainty-quantified simulators.** Diffusion-based generative models naturally produce full trajectory samples, providing built-in access to uncertainty estimates on outputs [126] and these approaches may be useful across a range of applications.

4. Societal Tooling

- **Alignment With Human Preferences.** Fine-tuning AI models to human preferences was first done in transformers and then the methodology was carried over to the diffusion models which also allow to make it based not only on the final sample but also on intermediate steps (see [127] and references there in) – thus suggesting that more opportunities, e.g. for the decision flow type of control, are available to meet human performance expectations.
- **Resource-aware AI.** Sparse score models, diffusion pruning, and low-precision samplers offer dramatic gains in energy efficiency, enabling deployment on edge devices and mobile systems [128]. Integration and further development of this technology within the generative and decision flow frameworks have a tremendous potential.

- **Personalization Platforms.** Personalization task in GenAI aims to capture and utilize concepts as generative conditions, which are not easily describable. We are learning that many subject driven generation methods are generalizable – see [129] for review – better understand the general trends and develop universal controls is another challenge of the decision flow type.

Take-away: The synthesis chapter argued that diffusion, probability flows, energy functions, and score-functions and controls are *different and synergistic lenses on the same challenge*. We re-framed that unified view as a **tool-kit** guidance for applications:

BIG model ready (trained) → Compose experts → Steer paths → Sample solutions, then deploy the resulting samplers across design, decision-making, science, and society. The mathematics you have learned is therefore not an end-point but a **passport** to whichever downstream application excites you most.

Bibliography

- [1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention Is All You Need,” Dec. 2017, arXiv:1706.03762 [cs]. [Online]. Available: <http://arxiv.org/abs/1706.03762>
- [2] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge University Press, 2004. [Online]. Available: <https://web.stanford.edu/~boyd/cvxbook/>
- [3] Y. Nesterov, *Introductory Lectures on Convex Optimization: A Basic Course*. Springer, 2004. [Online]. Available: <https://doi.org/10.1007/978-1-4419-8853-9>
- [4] L. Bottou, “Large-scale machine learning with stochastic gradient descent,” pp. 177–186, 2010. [Online]. Available: <https://leon.bottou.org/papers/bottou-2010>
- [5] J. Duchi, E. Hazan, and Y. Singer, “Adaptive subgradient methods for online learning and stochastic optimization,” in *Proceedings of the 24th Annual Conference on Learning Theory (COLT)*, 2011.
- [6] T. Tieleman and G. Hinton, “Lecture 6.5—rmsprop: Divide the gradient by a running average of its recent magnitude,” COURSERA: Neural Networks for Machine Learning, 2012, https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.
- [7] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” in *International Conference on Learning Representations (ICLR)*, 2015.
- [8] E. Candes, J. Romberg, and T. Tao, “Robust uncertainty principles: exact signal reconstruction from highly incomplete frequency information,” *IEEE Transactions on Information Theory*, vol. 52, no. 2, pp. 489–509, 2006.
- [9] M. Kurtz, J. Kopinsky, R. Gelashvili, A. Matveev, J. Carr, M. Goin, W. Leiserson, S. Moore, N. Shavit, and D. Alistarh, “Inducing and Exploiting Activation Sparsity for Fast Inference on Deep Neural Networks,” in *Proceedings of the 37th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, H. D. III and A. Singh, Eds., vol. 119. PMLR, Jul. 2020, pp. 5533–5543. [Online]. Available: <https://proceedings.mlr.press/v119/kurtz20a.html>
- [10] T. Hoefler, D. Alistarh, T. Ben-Nun, N. Dryden, and A. Peste, “Sparsity in Deep Learning: Pruning and growth for efficient inference and training in neural networks,” Jan. 2021, arXiv:2102.00554 [cs]. [Online]. Available: <http://arxiv.org/abs/2102.00554>

- [11] Z. Wang, “SparseDNN: Fast Sparse Deep Learning Inference on CPUs,” Jul. 2021, arXiv:2101.07948 [cs]. [Online]. Available: <http://arxiv.org/abs/2101.07948>
- [12] M. Grimaldi, D. C. Ganji, I. Lazarevich, and S. Sah, “Accelerating Deep Neural Networks via Semi-Structured Activation Sparsity,” Sep. 2023, arXiv:2309.06626 [cs]. [Online]. Available: <http://arxiv.org/abs/2309.06626>
- [13] DeepSeek-AI, A. Liu, B. Feng, B. Xue, B. Wang, B. Wu, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan, D. Dai, D. Guo, D. Yang, D. Chen, D. Ji, E. Li, F. Lin, F. Dai, F. Luo, G. Hao, G. Chen, G. Li, H. Zhang, H. Bao, H. Xu, H. Wang, H. Zhang, H. Ding, H. Xin, H. Gao, H. Li, H. Qu, J. L. Cai, J. Liang, J. Guo, J. Ni, J. Li, J. Wang, J. Chen, J. Chen, J. Yuan, J. Qiu, J. Li, J. Song, K. Dong, K. Hu, K. Gao, K. Guan, K. Huang, K. Yu, L. Wang, L. Zhang, L. Xu, L. Xia, L. Zhao, L. Wang, L. Zhang, M. Li, M. Wang, M. Zhang, M. Tang, M. Li, N. Tian, P. Huang, P. Wang, P. Zhang, Q. Wang, Q. Zhu, Q. Chen, Q. Du, R. J. Chen, R. L. Jin, R. Ge, R. Zhang, R. Pan, R. Wang, R. Xu, R. Zhang, R. Chen, S. S. Li, S. Lu, S. Zhou, S. Chen, S. Wu, S. Ye, S. Ye, S. Ma, S. Wang, S. Zhou, S. Yu, S. Zhou, S. Pan, T. Wang, T. Yun, T. Pei, T. Sun, W. L. Xiao, W. Zeng, W. Zhao, W. An, W. Liu, W. Liang, W. Gao, W. Yu, W. Zhang, X. Q. Li, X. Jin, X. Wang, X. Bi, X. Liu, X. Wang, X. Shen, X. Chen, X. Zhang, X. Chen, X. Nie, X. Sun, X. Wang, X. Cheng, X. Liu, X. Xie, X. Liu, X. Yu, X. Song, X. Shan, X. Zhou, X. Yang, X. Li, X. Su, X. Lin, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. X. Zhu, Y. Zhang, Y. Xu, Y. Xu, Y. Huang, Y. Li, Y. Zhao, Y. Sun, Y. Li, Y. Wang, Y. Yu, Y. Zheng, Y. Zhang, Y. Shi, Y. Xiong, Y. He, Y. Tang, Y. Piao, Y. Wang, Y. Tan, Y. Ma, Y. Liu, Y. Guo, Y. Wu, Y. Ou, Y. Zhu, Y. Wang, Y. Gong, Y. Zou, Y. He, Y. Zha, Y. Xiong, Y. Ma, Y. Yan, Y. Luo, Y. You, Y. Liu, Y. Zhou, Z. F. Wu, Z. Z. Ren, Z. Ren, Z. Sha, Z. Fu, Z. Xu, Z. Huang, Z. Zhang, Z. Xie, Z. Zhang, Z. Hao, Z. Gou, Z. Ma, Z. Yan, Z. Shao, Z. Xu, Z. Wu, Z. Zhang, Z. Li, Z. Gu, Z. Zhu, Z. Liu, Z. Li, Z. Xie, Z. Song, Z. Gao, and Z. Pan, “DeepSeek-V3 Technical Report,” Dec. 2024, arXiv:2412.19437 [cs]. [Online]. Available: <http://arxiv.org/abs/2412.19437>
- [14] A. Behrouz, P. Zhong, and V. Mirrokni, “Titans: Learning to Memorize at Test Time,” Dec. 2024, arXiv:2501.00663 [cs]. [Online]. Available: <http://arxiv.org/abs/2501.00663>
- [15] Q. Sun, E. Cetin, and Y. Tang, “\$\\text{Transformer}^2\$: Self-adaptive LLMs,” Jan. 2025, arXiv:2501.06252 [cs]. [Online]. Available: <http://arxiv.org/abs/2501.06252>
- [16] F. Rosenblatt, “The perceptron: A probabilistic model for information storage and organization in the brain,” *Psychological Review*, vol. 65, no. 6, pp. 386–408, 1958.
- [17] M. Minsky and S. Papert, *Perceptrons: An Introduction to Computational Geometry*, 1st ed. Cambridge, MA: MIT Press, 1969.
- [18] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998, publisher: IEEE.

- [19] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet Classification with Deep Convolutional Neural Networks,” in *Advances in Neural Information Processing Systems*, F. Pereira, C. J. Burges, L. Bottou, and K. Q. Weinberger, Eds., vol. 25. Curran Associates, Inc., 2012. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf
- [20] G. E. Hinton and R. R. Salakhutdinov, “Reducing the dimensionality of data with neural networks,” *Science*, vol. 313, no. 5786, pp. 504–507, 2006.
- [21] O. Ronneberger, P. Fischer, and T. Brox, “U-Net: Convolutional Networks for Biomedical Image Segmentation,” May 2015, arXiv:1505.04597 [cs]. [Online]. Available: <http://arxiv.org/abs/1505.04597>
- [22] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 770–778, 2016.
- [23] R. T. Q. Chen, Y. Rubanova, J. Bettencourt, and D. Duvenaud, “Neural ordinary differential equations,” in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 31, 2018. [Online]. Available: <https://arxiv.org/abs/1806.07366>
- [24] S. Jastrzebski, Z. Kenton, D. Arpit, N. Ballas, A. Fischer, Y. Bengio, and A. Storkey, “Finding Flatter Minima with SGD,” in *International Conference on Learning Representations (ICLR) Workshop*, 2018. [Online]. Available: <https://openreview.net/forum?id=r1VF9dCUG>
- [25] I. Garg, P. Panda, and K. Roy, “A Low Effort Approach to Structured CNN Design Using PCA,” *IEEE Access*, vol. 8, pp. 1347–1360, 2020, arXiv:1812.06224 [cs]. [Online]. Available: <http://arxiv.org/abs/1812.06224>
- [26] I. E. Lagaris, A. Likas, and D. I. Fotiadis, “Artificial neural networks for solving ordinary and partial differential equations,” *IEEE Transactions on Neural Networks*, vol. 9, no. 5, pp. 987–1000, 1998.
- [27] M. Raissi, P. Perdikaris, and G. Karniadakis, “Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations,” *Journal of Computational Physics*, vol. 378, pp. 686–707, 2019.
- [28] I. G. Kevrekidis, C. W. Gear, and G. Hummer, “Equation-free modeling: Coarse-grained computations for complex dynamical systems,” *SIAM Journal on Scientific Computing*, vol. 24, no. 2, pp. 409–432, 2003.
- [29] M. Schmidt and H. Lipson, “Distilling free-form natural laws from experimental data,” *Science*, vol. 324, no. 5923, pp. 81–85, 2009, available at <https://faculty.washington.edu/morgansn/pmwiki/uploads/Site/schmidt-science2009.pdf>.

- [30] S. L. Brunton, J. L. Proctor, and J. N. Kutz, “Discovering governing equations from data by sparse identification of nonlinear dynamical systems,” *Proceedings of the National Academy of Sciences (PNAS)*, vol. 113, no. 15, pp. 3932–3937, 2016. [Online]. Available: <https://doi.org/10.1073/pnas.1517384113>
- [31] L. Lu, P. Jin, and G. E. Karniadakis, “Learning nonlinear operators via deeponet based on the universal approximation theorem of operators,” *Nature Machine Intelligence*, vol. 3, no. 3, pp. 218–229, 2021.
- [32] Z. Li, N. Kovachki, K. Azizzadenesheli, K. Liu, K. Bhattacharya, A. M. Stuart, and A. Anandkumar, “Fourier neural operator for parametric partial differential equations,” *arXiv preprint arXiv:2010.08895*, 2020. [Online]. Available: <https://arxiv.org/abs/2010.08895>
- [33] M. Chertkov, *Principles and Methods of Applied Mathematics*. World Scientific Publishing Company, 2025. [Online]. Available: <https://www.worldscientific.com/worldscibooks/10.1142/14184>
- [34] D. Williams, *Probability with Martingales*. Cambridge University Press, 1991.
- [35] N. Tishby, F. C. Pereira, and W. Bialek, “The information bottleneck method,” in *Proceedings of the 37th annual Allerton conference on communication, control, and computing*, 2000, pp. 368–377.
- [36] O. Ronneberger, P. Fischer, and T. Brox, “U-net: Convolutional networks for biomedical image segmentation,” *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, pp. 234–241, 2015.
- [37] J. J. Hopfield, “Neural networks and physical systems with emergent collective computational abilities,” *Proceedings of the National Academy of Sciences*, vol. 79, no. 8, pp. 2554–2558, 1982.
- [38] D. O. Hebb, *The Organization of Behavior: A Neuropsychological Theory*. Wiley, 1949.
- [39] D. Krotov and J. J. Hopfield, “Dense associative memory for pattern recognition,” *Advances in Neural Information Processing Systems*, vol. 29, pp. 1172–1180, 2016.
- [40] ——, “Hierarchical associative memory: Molecules of neuronal computation,” *Philosophical Transactions of the Royal Society B*, vol. 376, no. 1820, p. 20200136, 2021.
- [41] R. Y. Rubinstein, “Optimization of computer simulation models with rare events,” *European Journal of Operational Research*, vol. 99, no. 1, pp. 89–112, 1997.
- [42] R. Y. Rubinstein and D. P. Kroese, *The Cross-Entropy Method: A Unified Approach to Combinatorial Optimization, Monte-Carlo Simulation, and Machine Learning*, ser. Information Science and Statistics. New York: Springer, 2004.

- [43] N. Wiener, “Differential-space,” *Journal of Mathematical Physics*, vol. 2, no. 1, pp. 131–174, 1923.
- [44] ——, “The average value of a functional of brownian motion,” *Proceedings of the National Academy of Sciences*, vol. 10, no. 7, pp. 253–260, 1924.
- [45] M. Kac, “On distributions of certain wiener functionals,” *Transactions of the American Mathematical Society*, vol. 65, no. 1, pp. 1–13, 1949.
- [46] R. P. Feynman and A. R. Hibbs, *Quantum Mechanics and Path Integrals*. New York: McGraw-Hill, 1965, reprinted by Dover Publications, 2010.
- [47] L. Ornstein and G. E. Uhlenbeck, “On the theory of the brownian motion,” *Physical Review*, vol. 36, no. 5, pp. 823–841, 1930.
- [48] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction (2nd Edition)*. MIT Press, 2018.
- [49] A. Karpathy, “minGPT: A Minimal PyTorch Re-implementation of GPT,” 2020. [Online]. Available: <https://github.com/karpathy/minGPT>
- [50] M. Chertkov, “Principles and Methods of Applied Mathematics (Living Book),” 2024. [Online]. Available: <https://sites.google.com/site/mchertkov/research/living-books>
- [51] ——, “INFERLO: Inference, Learning and Optimization with Graphical Models (Living Book, <https://sites.google.com/site/mchertkov/research/living-books>),” 2024. [Online]. Available: <https://sites.google.com/site/mchertkov/research/living-books>
- [52] M. Mezard and A. Montanari, *Information, Physics, and Computation*. Oxford University Press, 2009.
- [53] V. G. Satorras and M. Welling, “Neural Enhanced Belief Propagation on Factor Graphs,” Mar. 2021, arXiv:2003.01998 [cs]. [Online]. Available: <http://arxiv.org/abs/2003.01998>
- [54] D. P. Kingma and M. Welling, “Auto-Encoding Variational Bayes,” Dec. 2022, arXiv:1312.6114 [cs, stat]. [Online]. Available: <http://arxiv.org/abs/1312.6114>
- [55] A. Hyvärinen, “Estimation of Non-Normalized Statistical Models by Score Matching,” *Journal of Machine Learning Research*, vol. 6, no. 24, pp. 695–709, 2005. [Online]. Available: <http://jmlr.org/papers/v6/hyvarinen05a.html>
- [56] J. Sohl-Dickstein, E. A. Weiss, N. Maheswaranathan, and S. Ganguli, “Deep Unsupervised Learning using Nonequilibrium Thermodynamics,” Nov. 2015, arXiv:1503.03585 [cond-mat, q-bio, stat]. [Online]. Available: <http://arxiv.org/abs/1503.03585>
- [57] Y. Song, J. Sohl-Dickstein, D. P. Kingma, A. Kumar, S. Ermon, and B. Poole, “Score-Based Generative Modeling through Stochastic Differential Equations,” Feb. 2021, arXiv:2011.13456 [cs, stat]. [Online]. Available: <http://arxiv.org/abs/2011.13456>

- [58] M. Vuffray, S. Misra, A. Lokhov, and M. Chertkov, “Interaction Screening: Efficient and Sample-Optimal Learning of Ising Models,” in *Advances in Neural Information Processing Systems*, D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, Eds., vol. 29. Curran Associates, Inc., 2016. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2016/file/861dc9bd7f4e7dd3cccd534d0ae2a2e9-Paper.pdf
- [59] A. Y. Lokhov, M. Vuffray, S. Misra, and M. Chertkov, “Optimal structure and parameter learning of Ising models,” *Science Advances*, vol. 4, no. 3, p. e1700791, Mar. 2018. [Online]. Available: <https://www.science.org/doi/10.1126/sciadv.1700791>
- [60] G. E. Hinton, “Training Products of Experts by Minimizing Contrastive Divergence,” *Neural Computation*, vol. 14, no. 8, pp. 1711–1800, 2002, publisher: MIT Press.
- [61] T. N. Kipf and M. Welling, “Semi-Supervised Classification with Graph Convolutional Networks,” Feb. 2017, arXiv:1609.02907 [cs]. [Online]. Available: <http://arxiv.org/abs/1609.02907>
- [62] Z. Li, N. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. Stuart, and A. Anandkumar, “Fourier Neural Operator for Parametric Partial Differential Equations,” May 2021, arXiv:2010.08895 [cs]. [Online]. Available: <http://arxiv.org/abs/2010.08895>
- [63] B. Amos, L. Xu, and J. Z. Kolter, “Input Convex Neural Networks,” Jun. 2017, arXiv:1609.07152 [cs]. [Online]. Available: <http://arxiv.org/abs/1609.07152>
- [64] H. Behjoo and M. Chertkov, “U-Turn Diffusion,” *Entropy*, vol. 27, no. 4, 2025. [Online]. Available: <https://www.mdpi.com/1099-4300/27/4/343>
- [65] ——, “Harmonic Path Integral Diffusion,” *IEEE Access*, vol. 13, pp. 42 196–42 213, 2025. [Online]. Available: <https://ieeexplore.ieee.org/document/10910146/>
- [66] E. Bengio, M. Jain, M. Korablyov, D. Precup, and Y. Bengio, “Flow Network based Generative Models for Non-Iterative Diverse Candidate Generation,” Nov. 2021, arXiv:2106.04399 [cs]. [Online]. Available: <http://arxiv.org/abs/2106.04399>
- [67] M. Chertkov, S. Ahn, and H. Behjoo, “Sampling Decisions,” Mar. 2025, arXiv:2503.14549 [cs]. [Online]. Available: <http://arxiv.org/abs/2503.14549>
- [68] Y. Song and S. Ermon, “Generative Modeling by Estimating Gradients of the Data Distribution,” Oct. 2020, arXiv:1907.05600 [cs, stat]. [Online]. Available: <http://arxiv.org/abs/1907.05600>
- [69] J. Ho, A. Jain, and P. Abbeel, “Denoising Diffusion Probabilistic Models,” Dec. 2020, arXiv:2006.11239 [cs, stat]. [Online]. Available: <http://arxiv.org/abs/2006.11239>
- [70] B. D. Anderson, “Reverse-time diffusion equation models,” *Stochastic Processes and their Applications*, vol. 12, no. 3, pp. 313–326, May 1982. [Online]. Available: <https://linkinghub.elsevier.com/retrieve/pii/0304414982900515>

- [71] H. Behjoo and M. M. Chertkov, “Space-Time Diffusion Bridge,” *IFAC-PapersOnLine*, vol. 58, no. 17, pp. 274–279, 2024. [Online]. Available: <https://linkinghub.elsevier.com/retrieve/pii/S2405896324019360>
- [72] S. Särkkä and A. Solin, *Applied Stochastic Differential Equations*, 1st ed. Cambridge University Press, Apr. 2019. [Online]. Available: <https://www.cambridge.org/core/product/identifier/9781108186735/type/book>
- [73] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, “Generative Adversarial Networks,” Jun. 2014, arXiv:1406.2661 [cs, stat]. [Online]. Available: <http://arxiv.org/abs/1406.2661>
- [74] G. Alain and Y. Bengio, “What Regularized Auto-Encoders Learn from the Data Generating Distribution,” Aug. 2014, arXiv:1211.4246 [cs]. [Online]. Available: <http://arxiv.org/abs/1211.4246>
- [75] M. S. Albergo, N. M. Boffi, and E. Vanden-Eijnden, “Stochastic Interpolants: A Unifying Framework for Flows and Diffusions,” Nov. 2023, arXiv:2303.08797 [cond-mat]. [Online]. Available: <http://arxiv.org/abs/2303.08797>
- [76] D. P. Kingma and M. Welling, “An Introduction to Variational Autoencoders,” *Foundations and Trends® in Machine Learning*, vol. 12, no. 4, pp. 307–392, 2019, arXiv:1906.02691 [cs]. [Online]. Available: <http://arxiv.org/abs/1906.02691>
- [77] C. K. Sønderby, T. Raiko, L. Maaløe, S. K. Sønderby, and O. Winther, “Ladder Variational Autoencoders,” May 2016, arXiv:1602.02282 [stat]. [Online]. Available: <http://arxiv.org/abs/1602.02282>
- [78] G. Biroli and M. Mézard, “Generative diffusion in very large dimensions,” *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2023, no. 9, p. 093402, Sep. 2023. [Online]. Available: <https://iopscience.iop.org/article/10.1088/1742-5468/acf8ba>
- [79] G. Biroli, T. Bonnaire, V. de Bortoli, and M. Mézard, “Dynamical Regimes of Diffusion Models,” Feb. 2024, arXiv:2402.18491 [cond-mat]. [Online]. Available: <http://arxiv.org/abs/2402.18491>
- [80] D. P. Bertsekas, *Dynamic Programming and Optimal Control, Vol. I and II*. Athena Scientific, 1995.
- [81] R. Bellman, *Dynamic Programming*. Princeton University Press, 1957.
- [82] L. V. Kantorovich, “Mathematical methods of organizing and planning production,” Leningrad State University Press, 1939, in Russian; partial translations appeared later in various economic journals.
- [83] L. V. Kantorovich and A. G. Zel'dovich, *Mathematical Methods of Organizing and Planning Production*. Moscow: State Publishing House of Physical and Mathematical Literature, 1960, in Russian; later English translation by Daniel, M.

- [84] L. P. Kaelbling, M. L. Littman, and A. W. Moore, “Reinforcement learning: A survey,” *Journal of Artificial Intelligence Research*, vol. 4, pp. 237–285, 1996.
- [85] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, “Deep reinforcement learning: A brief survey,” *IEEE Signal Processing Magazine*, vol. 34, no. 6, pp. 26–38, 2017.
- [86] J. Li, W. Monroe, T. Shi, S. Jean, A. Ritter, and D. Jurafsky, “Deep reinforcement learning for dialogue generation,” in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2016, pp. 1192–1202.
- [87] D. M. Ziegler, N. Stiennon, J. Wu, T. B. Brown, A. Radford, D. Amodei, P. Christiano, and G. Irving, “Fine-tuning language models from human preferences,” *arXiv:1909.08593*, 2019.
- [88] C. Koh, L. Pagnier, and M. Chertkov, “Physics-guided actor-critic reinforcement learning for swimming in turbulence,” *Physical Review Research*, vol. 7, no. 1, p. 013121, Jan. 2025. [Online]. Available: <https://link.aps.org/doi/10.1103/PhysRevResearch.7.013121>
- [89] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. The MIT Press, 2018. [Online]. Available: <http://incompleteideas.net/book/the-book-2nd.html>
- [90] K. Black, M. Janner, Y. Du, I. Kostrikov, and S. Levine, “Training Diffusion Models with Reinforcement Learning,” Jan. 2024, arXiv:2305.13301 [cs]. [Online]. Available: <http://arxiv.org/abs/2305.13301>
- [91] Y. Zhang, E. Tzeng, Y. Du, and D. Kislyuk, “Large-scale Reinforcement Learning for Diffusion Models,” Jan. 2024, arXiv:2401.12244 [cs]. [Online]. Available: <http://arxiv.org/abs/2401.12244>
- [92] S. Shekhar and T. Zhang, “ROCM: RLHF on consistency models,” Mar. 2025, arXiv:2503.06171 [cs]. [Online]. Available: <http://arxiv.org/abs/2503.06171>
- [93] K. Yang, J. Tao, J. Lyu, C. Ge, J. Chen, Q. Li, W. Shen, X. Zhu, and X. Li, “Using Human Feedback to Fine-tune Diffusion Models without Any Reward Model,” Mar. 2024, arXiv:2311.13231 [cs]. [Online]. Available: <http://arxiv.org/abs/2311.13231>
- [94] Z. Zhu, H. Zhao, H. He, Y. Zhong, S. Zhang, H. Guo, T. Chen, and W. Zhang, “Diffusion Models for Reinforcement Learning: A Survey,” Feb. 2024, arXiv:2311.01223 [cs]. [Online]. Available: <http://arxiv.org/abs/2311.01223>
- [95] W. H. Fleming, “Exit probabilities and optimal stochastic control,” *Applied Mathematics and Optimization*, vol. 4, no. 1, pp. 329–346, 1977.

- [96] S. K. Mitter, “Non-linear filtering and stochastic mechanics,” in *NATO Advanced Study Institutes Series. Stochastic Systems: The Mathematics of Filtering and Identification and Applications*, vol. 78, 1981, pp. 479–503.
- [97] H. J. Kappen, “Path integrals and symmetry breaking for optimal control theory,” *Journal of Statistical Mechanics: Theory and Experiment*, p. P11011, 2005.
- [98] E. Todorov, “Linearly-solvable Markov decision problems,” in *Advances in neural information processing systems*, vol. 19. MIT Press, 2007, pp. 1369–1376.
- [99] B. Tzen and M. Raginsky, “Theoretical guarantees for sampling and inference in generative models with latent diffusions,” May 2019, arXiv:1903.01608 [cs, math, stat]. [Online]. Available: <http://arxiv.org/abs/1903.01608>
- [100] Y. Bengio, S. Lahlou, T. Deleu, E. J. Hu, M. Tiwari, and E. Bengio, “GFlowNet Foundations,” Jul. 2023, arXiv:2111.09266 [cs]. [Online]. Available: <http://arxiv.org/abs/2111.09266>
- [101] M. Uehara, Y. Zhao, C. Wang, X. Li, A. Regev, S. Levine, and T. Biancalani, “Inference-Time Alignment in Diffusion Models with Reward-Guided Generation: Tutorial and Review,” Jan. 2025, arXiv:2501.09685 [cs]. [Online]. Available: <http://arxiv.org/abs/2501.09685>
- [102] B. Geshkovski, C. Letrouit, Y. Polyanskiy, and P. Rigollet, “The emergence of clusters in self-attention dynamics,” May 2023, arXiv:2305.05465 [cs, math, stat]. [Online]. Available: <http://arxiv.org/abs/2305.05465>
- [103] ——, “A mathematical perspective on Transformers,” Feb. 2024, arXiv:2312.10794 [cs, math]. [Online]. Available: <http://arxiv.org/abs/2312.10794>
- [104] M. M. Chertkov, “Mixing artificial and natural intelligence: from statistical mechanics to AI and back to turbulence,” *Journal of Physics A: Mathematical and Theoretical*, vol. 57, no. 33, p. 333001, Sep. 2024. [Online]. Available: <https://iopscience.iop.org/article/10.1088/1751-8121/ad67bb>
- [105] Y. Du and L. Kaelbling, “Compositional Generative Modeling: A Single Model is Not All You Need,” Jun. 2024, arXiv:2402.01103 [cs]. [Online]. Available: <http://arxiv.org/abs/2402.01103>
- [106] J. Ho and T. Salimans, “Classifier-Free Diffusion Guidance,” Jul. 2022, arXiv:2207.12598 [cs]. [Online]. Available: <http://arxiv.org/abs/2207.12598>
- [107] M. Skreta, T. Akhound-Sadegh, V. Ohanesian, R. Bondesan, A. Aspuru-Guzik, A. Doucet, R. Brekelmans, A. Tong, and K. Neklyudov, “Feynman-Kac Correctors in Diffusion: Annealing, Guidance, and Product of Experts,” Mar. 2025, arXiv:2503.02819 [cs]. [Online]. Available: <http://arxiv.org/abs/2503.02819>

- [108] Y. Du, C. Durkan, R. Strudel, J. B. Tenenbaum, S. Dieleman, R. Fergus, J. Sohl-Dickstein, A. Doucet, and W. Grathwohl, “Reduce, Reuse, Recycle: Compositional Generation with Energy-Based Diffusion Models and MCMC,” Sep. 2024, arXiv:2302.11552 [cs]. [Online]. Available: <http://arxiv.org/abs/2302.11552>
- [109] H. Zhu, T. Xiao, and V. G. Honavar, “3M-Diffusion: Latent Multi-Modal Diffusion for Language-Guided Molecular Structure Generation,” Oct. 2024, arXiv:2403.07179 [cs]. [Online]. Available: <http://arxiv.org/abs/2403.07179>
- [110] R. Rombach, A. Blattmann, D. Lorenz, P. Esser, and B. Ommer, “High-Resolution Image Synthesis with Latent Diffusion Models,” Apr. 2022, arXiv:2112.10752 [cs]. [Online]. Available: <http://arxiv.org/abs/2112.10752>
- [111] C. Meng, Y. He, Y. Song, J. Song, J. Wu, J.-Y. Zhu, and S. Ermon, “SDEdit: Guided Image Synthesis and Editing with Stochastic Differential Equations,” Jan. 2022, arXiv:2108.01073 [cs]. [Online]. Available: <http://arxiv.org/abs/2108.01073>
- [112] X. Li, T. Zhao, X. Zhu, J. Wang, T. Pang, and K. Fang, “Planning-Guided Diffusion Policy Learning for Generalizable Contact-Rich Bimanual Manipulation,” Feb. 2025, arXiv:2412.02676 [cs]. [Online]. Available: <http://arxiv.org/abs/2412.02676>
- [113] M. Janner, Y. Du, J. B. Tenenbaum, and S. Levine, “Planning with Diffusion for Flexible Behavior Synthesis,” Dec. 2022, arXiv:2205.09991 [cs]. [Online]. Available: <http://arxiv.org/abs/2205.09991>
- [114] I. Jackson, D. Ivanov, A. Dolgui, and J. Namdar, “Generative artificial intelligence in supply chain and operations management: a capability-based framework for analysis and implementation,” *International Journal of Production Research*, vol. 62, no. 17, pp. 6120–6145, Sep. 2024. [Online]. Available: <https://www.tandfonline.com/doi/full/10.1080/00207543.2024.2309309>
- [115] C. K. Sønderby, L. Espeholt, J. Heek, M. Dehghani, A. Oliver, T. Salimans, S. Agrawal, J. Hickey, and N. Kalchbrenner, “MetNet: A Neural Weather Model for Precipitation Forecasting,” Mar. 2020, arXiv:2003.12140 [cs]. [Online]. Available: <http://arxiv.org/abs/2003.12140>
- [116] V. Lee, M. Nguyen, L. Elzeiny, C. Deng, P. Abbeel, and J. Wawrzynek, “Chip Placement with Diffusion Models,” Mar. 2025, arXiv:2407.12282 [cs]. [Online]. Available: <http://arxiv.org/abs/2407.12282>
- [117] T. Wagenaar, S. Mancini, and A. Mateo-Gabín, “Generative Aerodynamic Design with Diffusion Probabilistic Models,” Sep. 2024, arXiv:2409.13328 [cs]. [Online]. Available: <http://arxiv.org/abs/2409.13328>
- [118] M. Wang and H. Ku, “Risk-sensitive policies for portfolio management,” *Expert Systems with Applications*, vol. 198, p. 116807, Jul. 2022. [Online]. Available: <https://linkinghub.elsevier.com/retrieve/pii/S0957417422002640>

- [119] S. Gu, L. Yang, Y. Du, G. Chen, F. Walter, J. Wang, and A. Knoll, “A Review of Safe Reinforcement Learning: Methods, Theory and Applications,” May 2024, arXiv:2205.10330 [cs]. [Online]. Available: <http://arxiv.org/abs/2205.10330>
- [120] L. Kunze, N. Hawes, T. Duckett, M. Hanheide, and T. Krajiník, “Artificial Intelligence for Long-Term Robot Autonomy: A Survey,” Jul. 2018, arXiv:1807.05196 [cs]. [Online]. Available: <http://arxiv.org/abs/1807.05196>
- [121] A. Jalal, M. Arvinte, G. Daras, E. Price, A. G. Dimakis, and J. I. Tamir, “Robust Compressed Sensing MRI with Deep Generative Priors,” Dec. 2021, arXiv:2108.01368 [cs]. [Online]. Available: <http://arxiv.org/abs/2108.01368>
- [122] U. b. Waheed, T. Alkhalfah, E. Haghigat, C. Song, and J. Virieux, “PINNtomo: Seismic tomography using physics-informed neural networks,” Apr. 2021, arXiv:2104.01588 [physics]. [Online]. Available: <http://arxiv.org/abs/2104.01588>
- [123] M. Mould, D. Gerosa, and S. R. Taylor, “Deep learning and Bayesian inference of gravitational-wave populations: Hierarchical black-hole mergers,” *Physical Review D*, vol. 106, no. 10, p. 103013, Nov. 2022. [Online]. Available: <https://link.aps.org/doi/10.1103/PhysRevD.106.103013>
- [124] M. D. Cranmer, R. Xu, P. Battaglia, and S. Ho, “Learning Symbolic Physics with Graph Networks,” Nov. 2019, arXiv:1909.05862 [cs]. [Online]. Available: <http://arxiv.org/abs/1909.05862>
- [125] L. Pan, A. Albalak, X. Wang, and W. Y. Wang, “Logic-LM: Empowering Large Language Models with Symbolic Solvers for Faithful Logical Reasoning,” Oct. 2023, arXiv:2305.12295 [cs]. [Online]. Available: <http://arxiv.org/abs/2305.12295>
- [126] D. Shu and A. B. Farimani, “Zero-Shot Uncertainty Quantification using Diffusion Probabilistic Models,” Aug. 2024, arXiv:2408.04718 [cs]. [Online]. Available: <http://arxiv.org/abs/2408.04718>
- [127] J. Ren, Y. Zhang, D. Liu, X. Zhang, and Q. Tian, “Refining Alignment Framework for Diffusion Models with Intermediate-Step Preference Ranking,” Feb. 2025, arXiv:2502.01667 [cs]. [Online]. Available: <http://arxiv.org/abs/2502.01667>
- [128] X. Yang, D. Zhou, J. Feng, and X. Wang, “Diffusion Probabilistic Model Made Slim,” Nov. 2022, arXiv:2211.17106 [cs]. [Online]. Available: <http://arxiv.org/abs/2211.17106>
- [129] P. Cao, F. Zhou, Q. Song, and L. Yang, “Controllable Generation with Text-to-Image Diffusion Models: A Survey,” Mar. 2024, arXiv:2403.04279 [cs]. [Online]. Available: <http://arxiv.org/abs/2403.04279>